Journal of Neutron Research 24 (2022) 33–72
DOI 10.3233/JNR-220002
IOS Press

Guidelines for collaborative development of sustainable data treatment software

Joachim Wuttke [a,*], Stephen Cottrell [b], Miguel A. Gonzalez [c], Anders Kaestner [d], Anders Markvardsen [b], Thomas H. Rod [e], Piotr Rozyczko [e] and Gagik Vardanyan [c]

[a] *Forschungszentrum Jülich GmbH, Jülich Centre for Neutron Science at Heinz Maier Leibnitz-Zentrum, Lichtenbergstraße 1, 85748 Garching, Germany*
[b] *ISIS Neutron and Muon Source, STFC Rutherford Appleton Laboratory, Didcot OX11 0QX, United Kingdom*
[c] *Institut Laue-Langevin, 71 avenue des Martyrs, CS 20156, 38042 Grenoble Cedex 9, France*
[d] *Paul Scherrer Institute, Forschungsstrasse 111, CH-5232 Villigen PSI, Switzerland*
[e] *European Spallation Source ERIC, PO BOX 176, SE-221 00 Lund, Sweden*

Abstract. Software development for data reduction and analysis at large research facilities is increasingly professionalized, and internationally coordinated. To foster software quality and sustainability, and to facilitate collaboration, representatives from software groups of European neutron and muon facilities have agreed on a set of guidelines for development practices, infrastructure, and functional and non-functional product properties. These guidelines have been derived from actual practices in software projects from the EU funded consortium 'Science and Innovation with Neutrons in Europe in 2020' (SINE2020), and have been enriched through extensive literature review. Besides guiding the work of the professional software engineers in our computing groups, we hope to influence scientists who are willing to contribute their own data treatment software to our community. Moreover, this work may also provide inspiration to scientific software development beyond the neutron and muon field.

Keywords: Software development, scientific software, scientific computing, research software, software engineering, sustainable software, data treatment, reproducible research

1. Introduction

1.1. Software needs at user facilities

Large research facilities, such as neutron, x-ray or muon sources, provide researchers across a range of disciplines (including physics, chemistry, biology and engineering) with unique opportunities to explore properties of materials. To achieve this, they operate a broad range of highly specialised instruments, each targeting a particular area of investigation. The structure of the generated datasets reflects the specific operation modes of the instruments, and therefore requires bespoke data treatment software.

In most cases, the data treatment is conceptually and practically divided in two stages: *reduction* and *analysis*. Typical *data reduction* steps are background subtraction, normalization to incident flux, detector efficiency correction, axis calibration, unit conversions and histogram rebinning. In a good first approximation, the outcome is a function such as the static structure factor $S(q)$, the dynamic structure factor $S(q, \omega)$, the reflectivity $R(q)$ or the diffraction peak intensity $I(hkl)$, which characterizes the sample with little remaining dependence on the instrument details. In the subsequent *data analysis*, this function is interpreted, and often fitted with mathematical models, in order to improve our understanding of the scattering target (sample).

*Corresponding author. E-mail: j.wuttke@fz-juelich.de.

The major sources in our field are *user facilities*: They operate instruments primarily for externals users, with different degrees of in-house research. In the early days many users had a special interest in the experimental method, and analysed their data with their own tools. Into the 1990s, it was quite normal for a PhD thesis to consist of a few scattering experiments on a single sample and a detailed analysis supported by some new software. With increasing maturity of the experimental techniques, users are less interested in methodological questions; they rightfully expect an instrument to just work, with the results obtained often contributing to a wider study. This is reflected in more recent PhD theses and journal publications, where increasingly experiments at large facilities are just one of several methods that are used to gain a complete understanding of the system under study.

Laboratory instruments bought from specialized manufacturers, such as the popular physical property measurement systems used for magnetic, thermal and electrical characterisation of samples, come with fully integrated software for experiment control and, to a varying degree, for data analysis. Well designed graphical interfaces minimize training needs. When developing software for large facilities we should strive for no lesser quality standard – even if experimentalists are exceptionally forbearing with unstable software, counter-intuitive interfaces and idiosyncratic commands, as long as they somehow succeed in treating their precious data.

In the past, much of the data treatment software in our field has been contributed by engaged scientists, who developed it primarily for their own needs, then shared or published it. In particular, many instrument scientists have increased the utility of their instrument by providing software tailored for the needs of their users. While this software contains invaluable application-domain expertise, it often raises concerns about its long-term maintenance. There have already been cases where outdated computers had to be kept in operation because source code was lost, or could not be recompiled. Therefore the development and maintenance of data treatment software needs to be professionalised.

Most instruments have a lifetime of decades, with occasional refurbishments or upgrades. Therefore software development is as a long-term endeavor. Solid engineering should facilitate maintenance and future updates. Most instruments embody a basic design that is shared with a number of similar instruments at different facilities. This opens opportunities to share software between similar instruments and across facilities. By sharing the development, better use is made of limited resources, bugs are found earlier, the confidence in the correctness grows and the scientific end users are better served.

1.2. From computer programming to software engineering

Software engineering has been defined as the "systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software to optimize its production, support, and quality" [145]. Software engineering goes beyond mere computer programming in that it is concerned with size, time, and cost [321].

Size concerns the code base as well as the data to be processed. In either respect, programming techniques and development methods that are fully sufficient at small scale may prove inadequate at large scale. With a growing code base comes the need for developer documentation, version control and continuous integration. With big data comes the need for optimization and parallelization. With involvement of more than one developer comes the need for coordination and shared technical choices.

Time refers to both the initial development and the total life span of a software. No engineering overhead is needed for code that is written in a few hours and run only once. However, for projects where the code will be used for a considerable period, one needs to consider what kinds of changes shall be sustained. To support incremental development, version control may already pay off after a couple of days. To react to changed requirements without breaking extant functionality, good tests are essential. To keep a software running for years and decades, one must cope with changes in external dependencies like libraries, compilers, and hardware [133].

Cost as the limiting factor of any large-scale or long-term effort means that we have to live with compromises, and constantly decide about trade-offs [190, p. 15].

To become a good programmer and software engineer, one needs theoretical knowledge and understanding as well as experience. Any programming course therefore involves exercises. Academic exercises, however, cannot be

scaled to the size, complexity and longevity that are constitutive for engineering tasks. Engineering training therefore is mostly done on the job, but needs to be supported by continuous reading and other forms of self-education [10, rule 2.2]. Professionalisation of scientific software development is now a strong movement, expressed for instance in the take up of the job designation "research software engineer" (RSE), coined by the UK Software Sustainability Institute [273] and adopted by a growing number of RSE associations [156].

1.3. The case for shared engineering guidelines

In engineering, established rules and available technologies still leave vast freedom for the practitioner to do things in different ways. However, freedom in secondary questions can be more distracting than empowering. Arbitrary, seemingly inconsequential, choices can become the root of long-standing incompatibilities. Any organisation, therefore, needs to complement universal rules of the profession with more detailed in-house guidance [260]. In our field, the minimum organisational level for agreeing on software engineering rules is either the software project[1] or the developers' group. Groups for scientific software, or specifically for data reduction and analysis software, have been created at most large facilities. As some projects involve several computing groups and some developers contribute to several projects, it is desirable to agree on basic guidelines across groups *and* across projects.

Software groups from five major European neutron and muon facilities [73,125,143,144,212] came together in the SINE2020 consortium [267] to define a set of guidelines for collaborative development of data treatment software. This collaboration is continuing, including through the League of advanced European Neutron Sources (LENS) [167]. The authors of this paper include software engineers and project managers at the aforementioned five sources. The guidelines presented in the following are based on our combined experience. They improve and update on material collected in SINE2020 [181].

1.4. Scope of these guidelines

This paper has been compiled with a broad audience in mind. We hope that our guidelines, beyond governing work in our own groups, will also be taken into consideration by instrument responsibles and other scientists who are contributing to data treatment software, as this would facilitate future collaborations. Aspects of our work may also be of interest to RSEs from other application fields and to managers who want to get a better understanding for the scope of software engineering and the challenges of long-term maintenance.

Our guidelines are meant not only for new projects but also for the maintenance, renovation, and extension of extant code. In any case, with years, the distinction between new and old projects blurs. In our field, where projects have a lifetime of decades, most of us are working most of the time on software initiated by somebody else. Conversion to the tools and practices recommended here needs not to be sudden and complete, but can be done in steps, and in parallel with work on new functionality.

This paper is at the same time descriptive and prescriptive. It documents our current best practices, gives mandatory guidance to our own staff, explains our choices to external collaborators and invites independent developers to consider certain tools and procedures. In the following, each subsection starts with a brief guideline in a text box, which is then discussed in detail. Section 3 describes the procedures and infrastructure that should be setup by a software group, Section 4 addresses the functionality that is expected from data treatment software, and Section 5 gives recommendations for non-functional choices.

[1] As customary in open-source or agile context [250], we use the term *software project* (or *project* for short) in a wide sense that includes open-ended undertakings, contrary to the standard definition of a project as "a temporary endeavor" [233].

1.5. Reproducible computation, FAIR principles and workflow preservation

While many of the points discussed in this paper hold true for software engineering or data treatment in general, one concern is particularly pertinent for research code: reproducibility. Computational reproducibility is one of several requirements for replication, and "replication is the ultimate standard by which scientific claims are judged" [214].

In reasearch with scattering methods, there is no tradition of explicit replication studies. However, breakthrough experiments on novel materials or discoveries of unusual material behavior motivate many follow-up studies on similar samples. Erroneous conclusions would soon lead to inconsistencies, which would provoke further experiments and deeper analysis until full clarification is reached. Potential sources of error include sample preparation, hardware functioning, instrument control, and data treatment. Disentangling these is much easier if experimental data are shared and data treatment is made reproducible. As experiments are quite expensive, replicating and validating data reduction and analysis should be among the first steps in scrutinizing unusual results.

Open *data* is encouraged, and increasingly requested, by research organizations. Broad consensus has been reached that research data should be findable, accessible, interoperable, and reusable (FAIR) [17,45,316]. Software is data as well, albeit of a very special kind. The FAIR principles, suitably adapated, can provide guidance how to share and preserve software [150,157,166]. Functional aspects of software, on the other hand, are outside the scope of these principles.

To make software findable and accessible, we recommend version control (Section 3.2.1), numbered releases and digital object identifiers (Section 3.2.6), provenance information in treated data (Section 4.3), rich online documentation (Section 5.9), and possibly the deployment in containers (Section 5.10.4). For interoperability, we recommend versatile interfaces (Section 4.1), standard data formats (Section 4.2) and code modularization (Section 5.4.1). To make code reusable, we emphasize the need for a clear and standard open-source license (Section 5.1), and recommend the use of standard programming languages (Section 5.2.2) and of plain American English (Section 5.3.2).

In the long term, the biggest risk to software reuse, and thereby to computational reproducibility, comes from interface breaking changes in lower software or hardware layers [133]. To give a recent example, numeric code that relied on *long double* numbers to be implemented with 80 bits no longer works on new Macintosh machines as Apple replaced the x86 processor family by the arm64 architecture, with compiler backends falling back to the minimum length of 64 bits required by the C and C++ standards. Strategies to reduce the risk of such *software collapse* [133] include: Keep your software simple and modular (Section 5.4.1); wherever possible, use standard components (Section 5.4.2); ideally, depend on external tools or libraries that have several independent embodiments (in our example, long double never had more than 64 bits under the Visual Studio compiler, which should have warned us not to rely on a non-standard extension); run nightly tests to detect defects as soon as possible (Section 5.8.2). Backward compatibility can help to replicate old data analyses even if old software versions no longer work (Section 5.10.2).

To replicate a data treatment, one needs to retrieve the data, the software, and the full parameterization of the workflow (Section 4.1, Fig. 2). We must therefore enable our users to preserve enough machine readable information so that entire data reduction and analysis workflows can be replayed. While this is relatively straightforward for script-based programs (Sections 4.1.2–4.1.5), extra functionality must be provided if software is run through a graphical user interface (Sections 4.1.1, 4.3).

2. How we wrote this paper

To get an overview of received practices in our field, we started this work with a questionnaire targeted at five neutron software packages that were under active development and were partly funded by SINE2020. The packages considered were *Mantid* [9,178], *ImagingSuite* [32,154,155], *McStas* [188,317,318], *BornAgain* [23,228] and *SasView* [254]. Questions and answers are fully documented in a project report [181], which also contains some

statistics and a summary of the commonalities. The empirical guidelines that emerged from this study became a deliverable of work package 10 of the European Union sponsored consortium SINE2020 (Science & Innovation with Neutrons in Europe) [267]. For the present publication we updated and extended the guidelines, based on evidence from research and practice.

For each section of this paper, we searched for literature, using both Google Scholar and standard Google search to discover academic papers as well as less formal work. We also (re)read a number of books [40,76,84,127,134,183,191,246,248,290,323] and style guides [115,286,297], and looked through many years of the "Scientific Programming" department in the journal *Computing in Science & Engineering* [46]. Other formative influences include the question and answer site Stackoverflow [278], mailing lists or discussion forums of important libraries (e.g. *Scipy* or *Qt*), conferences (or videorecorded conference talks) on software engineering or on single programming languages, initiatives such as CodeRefinery [44] and the carpentries of the Software Sustainability Institute [273]. Insights from these sources are reflected in numerous places below. References attest that our views and recommendations are mainstream. Much good advice has been found in blogs, and in books that risk strong opinions. Most of these sources are about software engineering in general, not specifically about research software.

Each guideline in the final text has been approved by each of us. Even so, we do not claim that we are already following all guidelines in all our own software efforts. During the years that led to the compilation of this paper we learned a lot from each other and from the literature, and we continue to bring the gained insights to our own work.

3. Procedures and infrastructure

3.1. Development methodology

3.1.1. Ideas from agile

> Adapt agile methodology to your needs.

Over the past twenty years, the discourse on software development methodology has been dominated by ideas from the *Agile Manifesto* [20,196]. Subsets of these ideas have been combined and expanded into fully fledged methods like *Extreme Programming* (XP) [19], *Scrum* [258,259], *Lean* [227], *Kanban* [6], *Crystal* [42] and hybrids of these like *Scrumban* [165] and *Lean-Kanban* [49]. For a critical review, we recommend the book by Bertrand Meyer [191].

Many organizations that claim to be *agile* do not strictly adhere to one of the more formal methods, but have adopted and adapted some agile ideas. This is also the case for our facilities: We all embrace agile but do not consider it to be a panacea. To varying extent, our groups practice concepts like short daily meetings, retrospectives, pair programming, user stories. Refactoring (Section 3.1.3), iterative workflows and continous integration (Section 3.2.3) have become second nature to our teams. Indeed, good ideas from agile permeate much of this paper.

3.1.2. Project management and stakeholder involvement

> Involve user representatives.

Scrum introduced the notion of a *product owner* who represents the customers, "who champions the product, who facilitates the product decisions, and who has the final say about the product" [222]. At our facilities, we have no dedicated full-time person for this role. Regardless, it is an important insight that developers need a counterpart who forcefully represents the interests of end users.

To identify such representatives in our application domain, the distinction of data reduction versus analysis software (Section 1.1) comes into play. For data *reduction*, the instrument responsibles are the natural mediators

between the user community at large and the software developers; they have full knowledge of the instrument physics, and they get direct feedback from their users when there are any problems with the software. Indeed, we have had excellent results with instrument scientists guiding our work and testing our software prototypes. Some facilities also have staff (e.g. instrument data scientists) whose job includes the task to liaise with users, instrument teams, and developers. For a multi-facility software like Mantid, formal governance bodies are required [179]. While instrument scientists can help with the customization to their respective instruments, they may have quite different requirements for the common parts of the software; these divergent wishes need to be bundled, reviewed, and reconciled.

In contrast, many software projects in the realm of data *analysis* are steered by a scientist who acts as product owner and chief maintainer in one person. This is essentially the "benevolent dictator" management model [92, 142,247], which has proven viable for large and important open-source projects like Linux, Emacs, Python, and many more [313]. SasView, governed by a committee [255], seems a rare exception in our field. Many owners of analysis software are scientists who originally wrote the software for their own research needs, then started to support a growing user community. If such a creator retires, and maintenance is taken over by software engineers without domain knowledge, then the loss of application-domain knowledge must somehow be compensated from within the user community. The community must be empowered to articulate their needs, for instance through online forums, user meetings, webinars, not to mention essentials like an issue tracker and a contact email. In addition, users can participate in formal or de facto steering committees or user reference groups.

3.1.3. Refactoring

Keep code clean through test-covered refactoring.

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [84]. We refactor before and after changing external behavior, both to lay the groundwork and to clean up the code. Continual refactoring prevents the accumulation of technical debt and keeps the code readable, modular and maintainable. Through refactoring, code from different sources is made more homogeneous, and brought in line with received design principles (Section 5.4.3). Typical targets of refactoring are known as *code smell* [84,183,288], which include inaccurate names, code duplication, overly large classes, overlong functions, more than three or four arguments to a function, chains of getter calls, etc.

Even trivial refactorings, like the renaming of variables, are error-prone. Automatic editing tools can help to avoid certain errors, at the risk of introducing errors of other kinds. Each single refactoring should be separately committed to the version control (Section 3.2.1) so that it can easily be reverted if needed. Good test coverage is a precondition; often, it is appropriate to add a new test to the regression suite (Section 5.8.1) before undertaking a specific refactoring step.

In private enterprises, it is often hard for developers to assert the need for refactoring while management presses for new functionality. In our non-profit setting, we must be no less wary of the opposite danger of procrastinating in refactoring without adding value to our product.

At times, the complexity of a software feels so overwhelming that developers will propose to rewrite big parts or all of the "legacy" code [183, p. 5]. We have had bad experiences with such initiatives. Legacy code, for all its ugliness and complexity, "often differs from its suggested alternative by actually working and scaling" [285]. There is considerable risk, well-known as the *second-system effect* [29], that the new project, in response to all lessons learned from the previous one, is so heavily overengineered that it will never be finished. On the other hand, if there is need for architectural improvements, then piecemeal refactoring without sufficient upfront design is also "terrible advice, belonging to the 'ugly' part of agile" [191]. We therefore recommend that major changes to a software framework be proposed in writing, and reviewed, before substantial time is invested into implementation. Innovative ideas may require development of a prototype or a minimum viable product, or some other proof of concept.

For significant renovations, or for writing code from scratch, work should be broken down into packages and iterations such that tangible results are obtained every few months at most, and not much work is lost in case a developer departs. For critical work, more than one developer should be assigned from the outset if possible.

3.2. Development workflow and toolchain

3.2.1. Version control

Put code and related artifacts under Git version control.

Source code, and all other human-created artifacts that belong to a software project, should be put under version control [215,275]. Version control (also called revision control or source code management) is supported by dedicated software that preserves the full change history of a file *repository*. With each *commit*, the timestamp, the author name and a message are logged. Any previous state of any file can be restored. This is useful when some modification of the code either did not meet expectations or broke functionality. For particularly difficult bug hunts, bisection on the history can help.

Working habits can and should change once version control is ensured. One needs no longer to care about safety copies. One can take more risks, such as using automatic editing tools as it is easy to revert failed operations. Since deleted code can easily be restored, there is hardly a good reason left for outcommenting used code or for maintaining unused files [172]. And most importantly, version control makes it safe and easy for developers to work in parallel on *branches* of the code. Conflicting edits have to be resolved before a branch is merged back into the common trunk.

We recommend the version control software *Git* [35,43,95], written by Linux founder Linus Torvalds [298], which is the de facto standard for new projects in many domains, including ours. The dominance of Git is cemented by repository managers like GitHub and GitLab (Section 3.2.2). For a curated list of Git resources, see [13].

3.2.2. Repository manager

Use Git-based repository management.

It is straightforward to install and run a Git server on any internet facing computer. The non-trivial restriction is "internet facing", which requires a permanent URL and appropriate firewall settings. For collaborative software development it is convenient to use a web-based repository manager that combines Git with an issue tracker and additional functionality for work planning, continuous integration (CI, Section 3.2.3), code review (Section 3.2.5), and deployment (Section 3.2.6).

The best known of these services is GitHub [96]. As for many other cloud services, it has a "freemium" business model, offering basic services for free, especially for open-source projects, and charging money for closed projects or when additional features are required. An interesting alternative is GitLab which, beyond offering freemium hosting [100], also releases their own core software under a freemium model. New features first appear in an Enterprise edition, but after some time they all go into an open-source release [101]. Some important creators of open-source software like Kitware (CMake, ParaView, VTK [107]) are running their own GitLab instance, and so do several of our institutions [105,106,108]. For collaborative projects it may be argued that hosting on a "neutral ground" cloud service is preferable for avoiding perceived ownership by a single institution.

In spite of some differences in terminology and in recommended work flows, GitHub and GitLab are very similar. Each of them is used through a terse, no-nonsense web interface that appeals to power users but presents a certain barrier to newcomers [98,102]. As external users are more likely to be familiar with GitHub, it may help to mirror GitLab repositories at GitHub [103].

3.2.3. Workflow; continuous integration, delivery, and deployment

Choose a workflow that fits well with Continuous Integration, Delivery or/and Deployment.

In a small or very stable software project, where no edit conflicts are to be expected, all development may take place in one single *main* (or *master*) branch. If more than one developer is contributing at the same time, then work

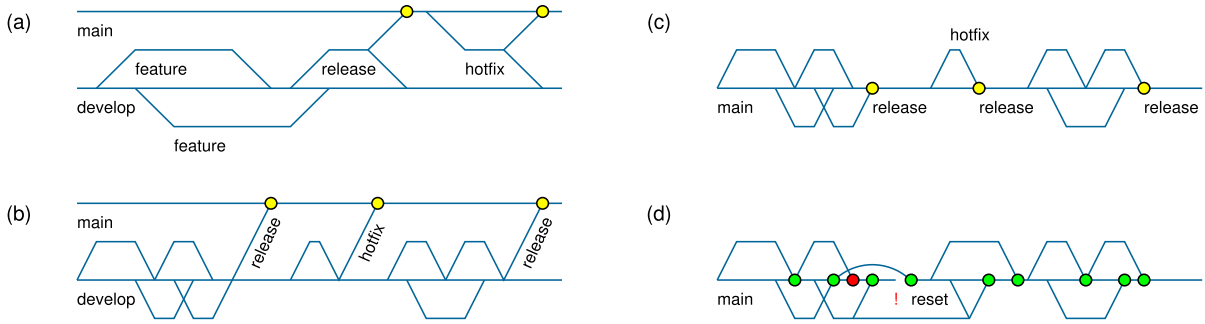

Fig. 1. (a) Development workflow according to the *Git Flow* model [65,186]. Feature branches are merged into *develop*. At some point, a *release* branch is started, which is frozen for new features. The release (yellow disk) is published after the *release* branch is merged into *main*. Occasionally, a *hotfix* is applied to *main*, and merged back into *develop*. (b) Development workflow with Continuous Integration (shortlived feature branches, frequent mergers) and Continuous Delivery (frequent releases, taken directly from the *develop* branch). (c) Ditto, but without *develop* branch; all changes go directly into *main*. This is promoted under the name *Trunk Based Development* [120]. (d) Workflow with Continuous Deployment, where the application is deployed after each successful merger (colored disks). If a critical bug is detected (red exclamation mark), and attributed to a recent merger (red disk), then it is easy to reset *main* to the last unaffected software state, and deploy that one.

is usually done in distinct *feature branches* of the code. At some point, successful developments are *merged* back into a common branch. Before the merger is enacted, possible edit conflicts must be resolved, integration tests must pass, and the changes should be reviewed.

The project specific *workflow* regulates the details of this. For quite a while, best practice was a workflow proposed in 2010 [65] and later termed *Git Flow* [186], summarized in Fig. 1a. This workflow is still appropriate if there is need to freeze a *release branch* during a period of manual testing.

More recently, the trend has gone towards shorter lived side branches, and more frequent integration into the common trunk. This is called *Continuous Integration* (CI) [224]. It heavily depends on fast and automated build and test procedures, discussed in the next Section 3.2.4. If CI procedures and developer discipline ensure that the common trunk is at any time in usable state, then it takes little additional automation to build all executables after each merger, including installers, and other deliverables that would be needed to deliver a new software version to the users. This is called *Continuous Delivery* (CD) [224], sketched in Fig. 1b. Automation of release procedures should also extend to the documentation, though human written content still needs to be kept up to date by humans.

With CI/CD, the old open-source advice “release early, release often” [246] is no longer thwarted by tedious manual release procedures, and we can “deploy early and often” [22]. In contrast, our end users who have to install the data analysis software on their own computers may perceive frequent releases more as an annoyance than as a helpful service. Each project needs to find the right balance.

When there are no more feature freezes and release branches, then there is no strong reason left to maintain separate *develop* and *main* branches. Rather, all changes can be merged directly into *main*, as sketched in Fig. 1c. This is promoted under the name *Trunk Based Development* [120].

Confusingly, the abbreviation CD is also used for *Continuous Deployment* [224], which goes beyond Continuous Delivery in that there are no releases anymore. Instead, new software versions are automatically deployed with high frequency, possibly after each single successful merger into the common trunk, and with no human intervention. One advantage is the ability to react swiftly and minimize damage after a critical bug is discovered (Fig. 1d). Arguably, Continuous Deployment is the best choice whenever deployment is under our own control. This is the case for cloud services (Section 5.2.1), or when data reduction software is only run on local machines.

3.2.4. Build and test automation

Use the repository manager to automatize tests and builds.

Continuous Integration (Section 3.2.3) depends on automation of build and test procedures. To control these procedures, some projects still use dedicated CI software like Jenkins. Nowadays, essential CI control is also provided by repository management software (RMS, Section 3.2.2). When a developer submits a Merge Request (GitLab) or Pull Request (GitHub), then the RMS (or other CI controller) launches build-and-test processes on one or several *integration servers*. Only if each of them has terminated successfully will the submitted change set be passed on for code review (Section 3.2.5) and then for merger into the common trunk (*develop* or *main*, Section 3.2.3).

Usually, the modified software has to be built and tested under all targeted operating systems, using as many integration servers, either on distinct hardware or in containers (e.g. *Docker* containers [63]). On each integration server a light background process runs, called the *runner* [97,104]. The runner frequently queries the RMS for instructions. Upon a Merge (Pull) Request, the RMS finds an available runner for each target platform and sends each of these runners the instruction to launch a new job. This then causes the runners to fetch the submitted change set, compile and build the software, run tests and possibly perform further actions according to a customized script. The progress log is continuously sent back to the RMS. Upon completion of the job, the runner sends the RMS a green or red flag that allows or vetoes the requested merger.

Groups or projects need to decide whether to run CI on their own hardware or in the cloud. For small projects the workstations under our desks may suffice. If there are several developers in a project, more fail-safe hardware is in order; if builds or tests are taking more than a few minutes, it is desirable to use a powerful multi-processor machine.

Alternatively, we could run our CI/CD processes on preconfigured cloud servers that are typically offered as freemium services [312]. Specifically, at the time of writing this paper, both GitHub and GitLab offer a practically unlimited number of build/integration/deployment jobs to be run on their free accounts. This may save us from buying and maintaining hardware, but at the expense of having less flexibility and occasional processing delays as free services are queued with low priority when computing centers are busy with payed jobs. Whether local or cloud based CI is less onerous to set up and keep running is an important question, which we must leave open.

Since the tests are an important part of a project's code base, we discuss them in more detail below, in the section on software requirements (Section 5.8).

3.2.5. Change review

Organize routine review of code changes.

Projects should define rules for reviewing proposed changes. Typically, a review is required for each Merge (Pull) Request. It takes place after all tests passed and before the merger is enacted. Repository management software (RMS, Section 3.2.2) provides good support for annotations and discussions.

Usually, the invitation to review a changeset goes to the other members of the team. The main purpose of code review [14,253] is quality assurance: prevent accidents, discover misunderstandings, maintain standards, improve idiom, keep the code readable and consistent. In some cases, the reviewer should build and run the changed code to check its validity, performance or other technical aspects.

Reviewers must be mindful not to hurt human feelings, the more so as all comments in our open-source repository management systems will remain worldwide readable indefinitely. What language is appropriate is very much culture dependent, and should be attuned to the reviewee. Improper bluntness can start undesirable power games [60]. Questions are better than judgements [122]. As soon as general principles, esthetic concerns or emotions get involved, it is advisable to change the medium, talk with each other, and possibly ask other developers or superiors for pertinent rules or specific guidance.

Besides quality assurance and education of the submitter, another purpose of code review is knowledge transfer from the submitter to the reviewer who is to be kept informed about changes in the code.

The RMS can be configured such that explicit approval from authorized parties is needed before a merger can be enacted. In this respect, habits and opinions vary across our groups and projects. Some enforce a strict four-eyes principle, either through RMS configuration or convention, while others entrust experienced developers the discretion to merge trivial or recurring changes without waiting for formal approval.

3.2.6. Releases

> Strive for regular releases.

A well-known adage from the early days of the open-source movement recommends "Release early, release often." [232,246]. If a software stagnates without fresh releases for more than a year, users will wonder whether the project is still alive. In contrast, frequent releases can be inconvenient for users who can end up spending considerable time reinstalling the software and adapting to its changes. Typically, our projects are aiming for two to four releases per year. The frequency might increase if software is deployed as a cloud service (Section 3.2.3). Release previews may be sent out to special users for extended manual testing.

Avoid changes that break other people's software or analysis pipelines [132,133,232]. If such changes become inevitable, discuss them in advance with your users, communicate them clearly, bundle them in one single release, and mark them by an incremented major version number [231]. If backward compatibility cannot be maintained it might still be possible to provide conversion scripts for user inputs.

It is good practice for a project to have a low-traffic moderated mailing list. In particular, this list should be used for announcing new releases. The announcement should contain an accurate but concise description of changes. The same information should also be aggregated in a change log (e.g. a file called "CHANGELOG" in the top-level directory). It "should be easily understandable by the end user, and highlight changes of importance, including changes in behavior and default settings, deprecated and new parameters, etc. It should not include changes that, while important, do not affect the end user, such as internal architectural changes. In other words, this is not simply the annotated output of *git log*" [118].

To facilitate citation, it is advisable to mint a digital object identifier (DOI) for each release. This can be accomplished using a free cloud service [77]. The most frequently used service, Zenodo [330], can be automatically triggered from GitHub.

4. Functional requirements

Functional requirements specify how the software shall transform input into output [281]. Obviously, these requirements are different for each application. Nonetheless, some fairly generic requirements can be formulated with respect to the more formal aspects of software control, data formats, logging, and plotting.

4.1. User interfaces

From the standpoint of theoretical computer science there is no fundamental difference between software, data, and parameters, they are all input to a Turing machine [134, Section 3.2], as outlined in Fig. 2. In practice, however, it matters that code, data, and parameters have a different origin and variability. Software for data reduction and analysis is ideally provided by the instrument operators, and is long-term stable with occasional updates. Data sets are immutable once generated; their format is defined by the instrument operators, their content comes from an experiment that is steered by an instrument user. Parameterization is chosen, often iteratively, by the investigators who thereby adapt the generic software to their specific needs. There are different ways to let users control the software, notably graphical user interface (GUI), shell commands, command-line interface (CLI), application programming interface (API) for scripting, and configuration files, to be discussed in the following subsections.

4.1.1. Graphical user interface

> A GUI facilitates interactive work. However, do not underestimate the development effort.

A well-designed GUI gently guides a user through a workflow, organizes options in a clear hierarchy of menus, and provides a natural embedding for data visualization and parameter control. A GUI tends to have a flatter

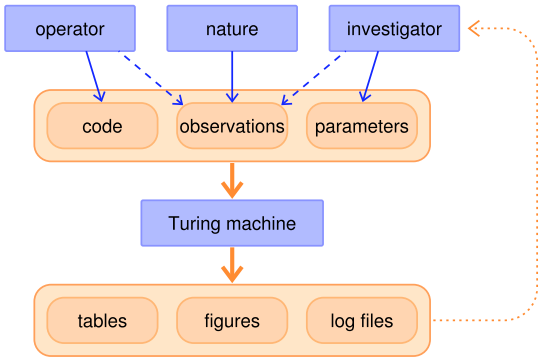

Fig. 2. Agents (blue) and data (orange) in experiment analysis. While code, experimental data, and parameters are exchangeable from the standpoint of theoretical computer science, they have different lifetime and are under control of different agents (instrument operator vs instrument user). The dotted feedback arrow indicates that the investigator may refine the parameterization iteratively in response to the output.

learning curve than a text-based interface, to the point that users familiarize themselves with a GUI through trial and error rather than by reading documentation.

These advantages are so strong that almost all modern consumer software is steered through a GUI. Most industry built laboratory instruments are also provided with some kind of GUI. This conditions the expectations of our users, who tend to regard a text-based interface as outdated and unpractical.

On the downside, serial data processing through a GUI can be tedious. Developers should be prepared to meet user demands for automating repetitive workflows. When there is need for high flexibility and for programmatic constructs like loops and conditionals, then it is not advisable to stretch the limits of what a GUI can do; rather we should provide a scripting interface – either instead of a GUI, or complementary to it.[2]

Be warned that a GUI is difficult and time-consuming to develop and maintain, the more so if developers from a science background are not trained for this kind of work. One hard problem is designing views and workflows to provide a good user experience, another one is getting the software architecture right. As several data representations (views) may need to change in response to user actions or external events, a naive use of the signalling mechanism provided by a GUI toolkit like Qt (Section 5.4.2) can result in hopeless spaghetti [268]. Cyclic dependences can stall or crash the GUI in ways that are difficult to debug. GUI tests take extra effort to automatize (Section 5.8.6).

Therefore, we recommend thinking carefully before allocating scarce resources to a GUI. A GUI is clearly a useful tool for supporting interactive, heuristic work, especially if the user is to interact with visualized data, e.g. by setting masks in histograms. For more involved visualization tasks, however, it may be best to refer users to some of the outstanding dedicated tools that exist already, especially so for imaging applications.

4.1.2. Shell commands

> Non-interactive tools should be launched through shell commands.

If a program for inspecting or transforming data does not require interactive control, then it should be packaged as an independent application that can be launched from the operating system's command prompt. Parameters, including paths of input and output files, can be supplied as command-line arguments.

Use cases include tabulating metadata or statistics from a set of raw data, summarizing contents of a binary file and transforming data between file formats. A number of such tools are available for the HDF5 binary format [124], and more specifically for the NeXus format [304, Ch. 7].

[2]Several of our software projects (BornAgain, Mantid, SasView) have both a GUI and a script API. In BornAgain, users can set up a simulation model in the GUI, and then export it to a Python script.

4.1.3. Command-line interface

> The traditional command-line has few remaining use cases.

An application with command-line interface (CLI) is associated with a terminal window and prompts the user for keyboard input. It can take the form of a dialogue where the user has to answer questions, or it provides a single prompt where users enter commands defined by the software. Hybrid forms also exist. A CLI is much easier to develop than a GUI, and therefore is appropriate for prototypes and niche applications.

A CLI in dialogue form (and equivalently, a GUI based dialogue with multiple popup windows) is acceptable for configuration work that rarely needs to be repeated. As a simpler alternative, however, consider using instead a configuration file that can be modified in a text editor (Section 4.1.5).

A CLI with menu prompt can be a powerful tool that gives users full control through concise commands. The disadvantages are a steep learning curve and the risk of feature creep: To save users from repetitious inputs, the concise commands tend to evolve into a Turing-complete ad-hoc language "that has all the architectural elegance of a shanty town" [201, p. 161]. It is therefore preferable to base a CLI on an established general-purpose scripting language, as discussed in the next subsection.

4.1.4. Python interface

> A Python interface is better than a self-invented command-line language.

Python is the most widely known programming language in large areas of science [216]. It is taught in many universities, and comes with lots of libraries and modules for scientific applications (Section 5.4.2). Python scripts can also be run from the popular *Jupyter* notebook [75,116,217]. Such a notebook is able to display a large variety of graphical elements like charts, plots and tables. Notebooks are usually accessed through a web browser, and are therefore natural candidates for "software as a service" (Section 5.2.1). As many data treatment programs already support Python, consistent choice of this language favors interoperability so that data acquisition, data reduction, analysis and visualization can be entirely controlled through Python.

For these reasons, it is a good choice to equip data treatment software with an interface that can be called from a Python script, interpreter, or notebook. To this end, we package our software (potentially written in another, lower-level language like C, C++ or Fortran) as a module, which can be imported by Python. The functionality of our software is exposed through class and function calls that form an application programming interface (API). Technically, our users take the role of application programmers. To get them started, it is advisable to provide tutorials with lots of example scripts.

4.1.5. Configuration files

> Support simple configuration files for user or site specific parameterization that changes rarely.

Parameterizing software through configuration files is cheap for developers and convenient for users, provided it is restricted to parameters that change only rarely. Good application cases are the customization of a software to local needs or personal taste.

An initial configuration should be provided by the software. It should be a text file in a simple format like INI or TOML, where each parameter is specified by a key/value pair. It is important that the format allows for ample comments for each parameter. Examples for configuration files with rich comments include the *Doxyfile* that steers the documentation extractor *Doxygen* [64], and many of the files in the */etc* directory of a Unix system.

4.2. Data formats and metadata

> Liaise with upstream stakeholders to foster standard data formats and full metadata capture.

Interoperable data formats are a long-standing concern in our community. Considerable effort has gone into the *NeXus* standard [162], used for raw data storage at many neutron, x-ray, and muon instruments. While NeXus is built on top of the binary HDF5 file format [123], some subcommunities prefer human-readable text formats. Reflectometry experts, for example, have recently agreed on a hybrid format for reduced data [203] that combines metadata in YAML [326] with a traditional plain table of q, I, ΔI lines.

As developers of downstream data treatment software, we can adapt our loader functions to whatever raw data format is being used. Our concern is not so much the file format, but the completeness, unambiguousness, and correctness of metadata. If a data treatment step needs metadata that are not contained in the input data, then we must prompt the user to manually supply the missing information. This is tedious and error-prone. Therefore we should liaise with the developers of upstream instrument control and data acquisition software to ensure that all necessary metadata are provided.

Occasionally, automatically acquired metadata are incomplete or incorrect, perhaps because some state parameters were not electronically captured, or were captured incorrectly. Should data treatment programs provide means to correct such errors? We rather recommend that metadata correction and enrichment be implemented as a separate step that precedes data reduction and analysis. Our rationale is that data correction is a one-time action, whereas data reduction and analysis software may be run repeatedly as users are working towards reliable fits and informative plots; it would be inconvenient for them to enter the same raw data corrections again and again when rerunning reduction and analysis. Furthermore, if experimental data are to be published under FAIR principles (Section 1.5), then both the automatically acquired and the corrected or enriched metadata need to be findable.

File formats should have a version tag so that downstream software recognizes when the input data structure has changed. We may need version-dependent switches to support all input of any age. Modern self-documenting formats make it possible to add new data or metadata fields without breaking loaders. However, once these new entries are expected by our software, we need a fallback mechanism for older files.

4.3. Log files vs provenance and warnings

> Write provenance information to output artifacts. Consider using log files to store information that does not belong to regular output artifacts. Generate warnings if physics or numerics are questionable.

Decades ago, it was quite standard for batch processing software to deliver not only numeric results but also a file that logged the actions of the software and associated output during data treatment. With modern GUI or scripting interfaces, writing these log files has somewhat fallen into disuse, and their original purpose is now frequently better served by other mechanisms.

As discussed in connection with Fig. 2, data treatment is conditioned by software, data, and parameters. These inputs make up the *provenance* [211] of the output data. To document provenance efficiently, parameters ought to be stored verbatim, whereas software and data should be pointed to by persistent identifiers. Instead of writing provenance information to a log file, it is far better to store it as metadata in each output artifact. Downstream software reads these metadata along with data input, adds further provenance information, and stores the enriched metadata with its output. Thereby FAIR archival is possible after each data processing step.

Alternatively or additionally, provenance information can also be preserved in form of a script that allows the entire data treatment to be replayed. Such a mechanism has been successfully implemented in *Mantid*, in *BornAgain*, and in the tomography reconstruction softwares *MuhRec* [154] and *Savu* [310]. Maintaining a step-by-step history of user actions can also help to implement *undo/redo* functionality, which is an important requirement for a user-friendly GUI.

Data treatment usually involves physical assumptions and mathematical approximations that are justified for certain ranges of control parameters and certain types of experimental data. Hopefully, code functionality will have been validated for a representative suite of input data, but it should be recognised that it may fail when applied to data that were not anticipated by the developers. This is a particularly serious problem in a research

context where versatile instruments are used to investigate new physical systems outside of the domain envisaged by the designer. As long as a data treatment software executes seamlessly its output will be believed by most users. Software authors therefore bear a heavy responsibility. We need to make our software robust in the hands of basic or intermediate users [168]. We need to detect invalid or questionable computations and warn users of the problem. Warnings written to a log file or a verbose monitor window are easily overlooked. We recommend clear and strong signalling, for instance by green-yellow-red flags, to mark output in a GUI as safe, questionable, or invalid.

Finally, the log files meantioned earlier can be the dump for background information that does not belong to the regular output artifacts. A good example is the log file written by the text processor TeX. Normally, it is safe to ignore such logs; however, users can consult them if some problem needs closer investigation.

4.4. Plots

> If a software includes data visualization, then support export to vector graphics and to data tables.

Whenever a software includes some means of data visualization, there will be user demand to preserve plots for internal documentation or for publication. If this is not properly supported by software, users will fall back on desperate means such as taking a screen copy, which is likely to be a crudely rasterized image that will deteriorate in subsequent image processing. We therefore recommend that all software provide an export mechanism for every displayed plot. The export options should include a standard vector graphics format to enable later rescaling or editing. Also consider providing an option to export displayed data as number to a simple tabular text file so that users can replot the data using a program of their own choice. Another export option could be a Python script that draws a figure by sending commands to a plot API such as MatPlotLib [140], Bokeh [151], or Plotly [280]; users can then edit the script to adapt the figure to their needs.

Certain software products boast of their ability to export "publication-grade graphics." Given the high variability of graphics quality in actual publications, this is a rather empty promise. Users who strive for above-average quality [300] may request (among other things) the following capabilities: free selection of plot symbols and colors [163]; free selection of aspect ratios [2]; sublinear scaling of supporting typefaces with image size; labels with Greek and mathematical symbols, subscripts and superscripts; free placement of a legend and other annotations in a plot; various ways to combine several plots into one figure. Clearly, it would be wasteful to implement all this functionality for each specialized piece of data reduction and analysis software. Instead, it is better to link our specialized software with a generic plotting library (possibly one of the aforementioned [140,151,280]) that is flexible enough to meet all the above demands and more.

5. Non-functional requirements

5.1. License

> Put all source code under a widely used, OSI approved, open-source license.

Choosing a license should be among the very first steps in a software project. Changing the license after the first code preview has gone public would seed confusion if not conflict. Changing the license after the first external contribution has been merged into the code can prove outright impossible. An example of extreme precaution is given by the Python Software Foundation [238], where new contributors need to sign a license agreement and to prove their identity before their patches are considered. While this would seem exaggerated in our field, it gives nonetheless a correct idea how essential it is for the juridical safety of a software that all contributions be made under the same, clearly communicated licensing terms [171].

We take for granted that all institutionally supported projects in our field will be published as free and open source software (FOSS), as the advantages for the facilities, the users, the scientific community, and the developers, are overwhelming [16,93,141,150,198,206,232]. Licenses must not include popular "academic" clauses like "non profit use" or "for inspection only", which nullify "the many significant benefits of community contribution, collaboration, and increased adoption that come with open source licensing" [198]. Choose the license among those approved by the Open Source Initiative (OSI) [204]. Only OSI-approved licenses will give free access to certain services [307]. Moreover, to minimize interpretive risks [248, Ch. 19.4] and to facilitate future reuse of software components by other projects, avoid the more exotic ones of the many OSI approved licenses. Preferably choose a license that is already widely used in our field. We have no common position on the long-standing debate as to whether or not the *GNU General Public License* [109] with its "viral" [90,283] precautions against non-free re-use is a good choice. Working group 4 of LENS [167] and UK Research and Innovation [302] are recommending the BSD 3-clause licence [205]. For textual documentation (as opposed to executable computer code), the *Creative Commons* licenses [54,283] are widely used.

The software license should be clearly communicated in the documentation and in a file called "COPYING" or "LICENSE" in the top-level source directory. Additionally, some software projects insist that each source file have a standardized header with metadata such as copyright, license and homepage.

5.2. Technology choices

5.2.1. Cross-platform development vs software as a service

> Develop from the onset for the different operating systems prefered by the end users, or consider deployment to the cloud.

Before starting code development, decide which operating system(s) shall be supported, since this has influence upon the choice of programming languages, tools and libraries. The decision depends on the expected workflow. For some instruments, data reduction is always carried out in the facility, during or immediately after the experiment. In this case, it suffices to support just a single platform preferred by the facility. Otherwise, and especially for later data analysis, users will run the software either on their own equipment or remotely on a server of the experimental facility. To support installation on user computers, we usually need to develop *cross-platform*, and provide installers or packages for Windows, MacOS and Linux alike (Sections 5.10.3, 5.10.4).

However, the global trend in research, as in other fields, goes towards cloud computing. In the "software as a service" (SaaS) model, a user runs a "thin client", most often just a web browser, to access an application that is executed in the host facility or a third-party server farm. Where raw data sets become too large to be taken home by the instrument users, SaaS is the only viable alternative to on-site data reduction. Once set up, SaaS will facilitate the data treatment in several ways, with users no longer needing to install the software locally or to download their raw data. In this configuration, software versions are controlled and updated centrally, and we no longer need to assist users in debugging unresolved dependencies and incompatible system libraries. Computations can be made faster by using powerful shared servers with a fast connection to the data source.

Until a few years ago, the user experience with SaaS for data analysis suffered from random delays in interactive visualization. Thanks to faster internet connections and to optimised client-side JavaScript libraries [136,177], this difficulty is essentially solved. A remaining difficulty is the administrative overhead for server operation and load balancing, for authentication and authorisation, and for managing access rights to input and output data. These topics are, for example, being addressed by the European Open Science Cloud [72] and by the joint photon and neutron projects ExPaNDS [74] and PaNOSC [210].

5.2.2. Programming languages

> Choose a programming language that has wide support in the community.

After deciding about the target platforms, and before starting to code, one needs to choose the programming language(s) to be used for the new software. In making this decision, intrinsic qualities of a language such as expressive power or ease of use play a relatively minor role. More important is the interoperability with extant products, social compatibility with the developer and user communities and the ease of recruiting.

Frequently used implementation languages in our field are C, C++, Fortran, Matlab, IDL and Python. Computing groups may need to support all of these, and are reluctant to add any new language to this portfolio. For new projects, the choice is even narrower as most of us would exclude Matlab and IDL for not being open source. The preferred scripting language in our community is Python. Hence, choosing Python as scripting language will facilitate collaboration and interoperablity as already discussed in Section 4.1.4 with regard to a user API.

When it comes to the choice between C, C++, and Fortran as languages for speed-critical core computations, our preference goes to C++ mostly because as an object oriented general-purpose language it also is an excellent platform for managing complex data hierarchies and for writing a GUI. For small libraries of general interest, plain C is still worth considering.[3]

To balance the conservatism of this guideline, we need to stay open to emerging alternative languages. Promising candidates include Rust, to deliver us from the technical debt and memory ownership issues of C++ [219], and Julia, which addresses various shortcomings of Python [66,218,329].

5.2.3. Keeping up with language evolution

Keep up as programming languages evolve.

Programming languages evolve in time. Compatiblity-breaking disruptions like the transition from Python2 to Python3 are very rare. More typical are revisions that fix unclear points in the language standard, introduce new constructs and provide new functionality in the standard library.

Developers need to keep up with this evolution. Projects that are under active development should encourage the use of new features once a language revision is supported on all target platforms. In the short term, it may seem wiser to freeze mature code at a certain language version; however, in the longer term, this may impair productivity and will be unattractive for developers. Inevitably then, any long-standing code base will be stylistically heterogeneous. Modularisation and encapsulation can help to keep different strata apart, and as long as a class or a library provides stable functionality through a well documented interface there is no need to modernize the internals.

5.3. Coding style

Here and in the next two sections we collect guidelines that apply to both C++ and Python, and possibly to other languages. Language specific guidelines follow in Sections 5.6 and 5.7.

5.3.1. Code formatting

Use automatic tools to ensure uniform code formatting.

Code formatting refers to layout decisions that affect the visual appearance of source files, but not the outcome of compilation. In most programming languages, the main degree of freedom is the insertion of white space (blank character and newline) between lexical tokens. White space governs indentation, line lengths, vertical alignment, and the placement of parentheses and braces. A coding style is a set of layout rules for a specific programming language. In C and C++, different combinations of rules for various syntax elements yield a huge number of possible coding styles, some of which have been popularised by books, corporate style guides and important open

[3] A long-standing problem with C under Windows, the non-compliance with standards C99 or later, and in particular the missing support for complex numbers, has been fixed in Visual Studio 2019 [193]. The advantage of C is that it can easily be called from, or wrapped by, almost any other programming language. This argument, however, is weakening as automatic wrapper generators for C++ keep improving.

source projects. In Python, early adoption of the authoritative style guide PEP8 [306] eliminated some variability of layout, but left many finer points open.

Within a project, a binding coding style should be adopted in order to minimize distraction in reading and writing, and to prevent friction between developers. It should be chosen early, and modified only for very good reasons, in order to keep the git history clean. The formatting can and should [260,305] be done automatically by tools like *clang-format* [38] for C and C++, or *yapf* [327] for Python, with project-wide parameterization in the top-level files *.clang-format* and *.style.yapf*, respectively. To ensure that code changes respect the chosen style, the formatting tools should be run as part of the test suite or from a pre-commit hook [35, Ch. 8.3].

5.3.2. English style

> Use plain American English.

For ease of international collaboration, all our writing should be in English. This concerns user-facing artifacts (such as graphical interface, scripting commands, log files and error messages, manual and website) as well as the code base (such as filenames, identifier names and comments) and internal documentation. As our users and developers need reading proficiency in technical English anyway, we would rather not spend scarce resources on translations into other languages. For consistency with external resources (such as operating system, libraries and literature) the spelling should be American [314]. Avoid slang and puns. Allusions to popular and high culture alike are not understood by everybody [286, NL.4].

5.3.3. File and identifier names

> Invest effort in finding expressive identifier and file names. Avoid abbreviations. Make consistent use of capital letters and underscores.

Expressive file and identifier names are of prime importance for the readability of large code bases. Use a dictionary and thesaurus to find accurate English designations. Employ grammatical categories consistently, as suggested in this recommendation: “Variables and classes should be nouns or noun phrases; Class names are like collective nouns; Variable names are like proper nouns; Procedure names should be verbs or verb phrases; Methods used to return a value should be nouns or noun phrases” [117].

It happens that a name was initially correct, but intention, scope or function of the designated object have drifted. Code review and refactoring rounds are occasions to get names right. Discussing naming rules and problematic cases can be a rewarding team exercise.

Camel case or connecting underscores can help to keep composite names readable. It is good practice to use typography for distinguishing categories of identifiers. For C++ and Python alike, we recommend the wide-spread convention that class names are in camel case (*SpecularScan*), and variables, including function arguments, in snake case (*scan_duration*) [115,301,306]. For function names, there is no consensus; projects have to choose between snake case and dromedar case (*executeScan*). Use these typographic devices with moderation. Don’t split words that may be written together in regular text: Prefer *wavenumber* and *Formfactor* over *wave_number* and *FormFactor*. Keep things simple: *Rawfile* is better than *RawDataFile*, *Sofa* better than *MultiButtSupporter* [48]. While Hungarian Notation (type designation by prefixes) is now generally discouraged [286, NL.5], [183,194,297], a few prefixes (or postfixes) may be helpful (e.g. *m_* for C++ class member variables [301]).

Directory and namespace hierarchies can help to avoid overlong names. Short names are in order for variables with local scope [286, NL.7], [297]: In a loop, *i* and *j* are better than *horizontal_pixel_index* and *vertical_pixel_index*. Consider names in context; if names are too similar then they will be confused in cursory reading [286, NL.19], [71,152].

General advice goes against abbreviations [115]: *createRawHistogramFile* is better than *crtRwHstgrFil*. However, some domain-specific abbreviations for recurrent complex terms are fine: Prefer *SLD* and *EISF* over numerous repetitions of *ScatteringLengthDensity* or *ElasticIncoherentStructureFactor*.

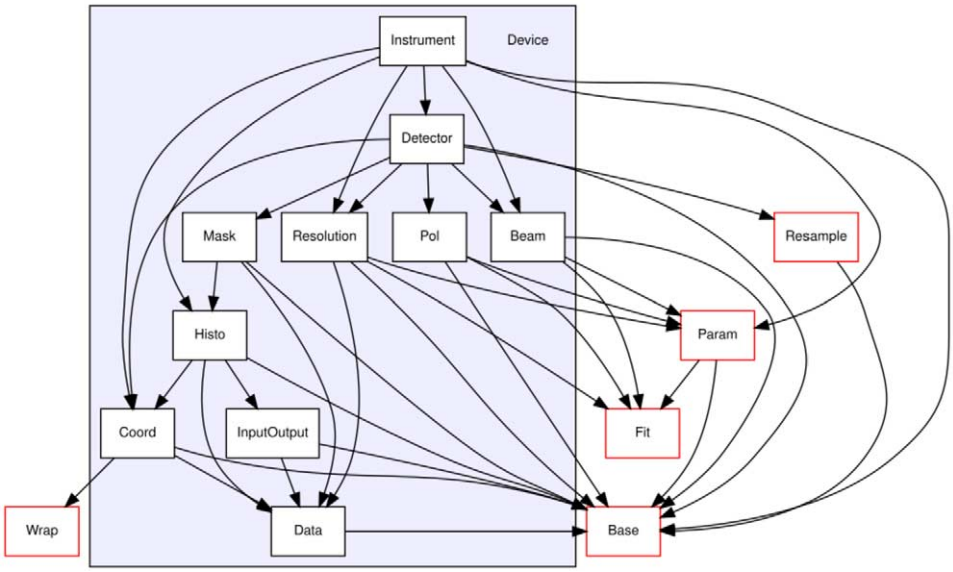

Fig. 3. Include dependencies between some library components (red frames, blue box) and subdirectories (within the blue box) of BornAgain. Such images, automatically generated by Doxygen [64] and therefore always up to date, provide documentation of a software's architecture. They are most useful if care is taken to organize directory dependencies as a directed graphs.

5.3.4. Source comments and developer documentation

> Keep source comments concise. Explain intent rather than how things are done. Where possible, generate documentation out of the code.

Documentation is difficult, and hardly ever completely satisfactory. This holds true even more so for the developer oriented than for the user oriented material. In the short term, developers neglect documentation because it does not help them in coding. In the long run, developer documentation is notoriously unreliable because there are no automatic tests to enforce that it stays in tune with the executable code. This well known fundamental problem [274] has no single comprehensive solution. We can however recommend the following strategies.

To keep documentation and code synchronized, generate documentation out of the code [190, p. 54]. To document an API, be it user or developer facing, use a standard tool like Doxygen for C++ [64] and Sphinx for Python [26].

Strive for the classic documentation style of Unix manual pages: telegraphic but complete. Equally avoid incomprehensibility and condescension. Assume an active reader who is able to deduce obvious consequences [248, Ch. 18.2]. Most importantly, don't say in comments what can be said in code [286, NL.1], [115,128].

General advice is that comments should explain the *intent* of code segments rather than their internal workings [286, NL.2]. If the code is structured into files, classes and functions that have *one* clear purpose (single responsibility principle, Section 5.4.3), and have consistent, accurate and expressive names (Section 5.3.3), then no comment may be needed. The section on comments in the Google C++ style guide [115] gives valuable hints as for which information should be provided in function and class comments.

Developers who are new to a project should seize this opportunity to improve the documentation while familiarizing themselves with extant code [184].

For projects of a certain complexity, a software architecture document is helpful or even indispensable. It "is a map of the software. We use it to see, at a glance, how the software is structured. It helps you understand the software's modules and components without digging into the code. It's a tool to communicate with others – developers and non-developers – about the software" [309]. It needs to stay high-level. "It will quickly lose value if

it's too detailed (comprehensive). It won't be up to date, it won't be trusted, and it won't be used" [309]. Any more detailed view of the software architecture should be automatically generated, for instance in form of the various diagrams produced by Doxygen, like the one shown in Fig. 3.

5.4. Architecture

5.4.1. Modularization

> Organize the code as a hierarchy of modules with clear scope. Keep GUI and physics code separated. Avoid cyclic dependencies.

"Decomposition driven by abstraction is the key to managing complexity. A decomposed, modular computer program is easier to write, debug, maintain and manage. A program consisting of modules that exhibit high internal cohesion and low coupling between each other is considered superior to a monolithic one" [294]. If a component has limited scope and complexity, chances increase that users are able to extend its functionality [206].

Large projects should organize their code into several object libraries (or Python modules). A good library structure may reduce the time spent on recompiling and linking, and thereby accelerate development cycles. Functions and classes that handle file, database or network access should be isolated from application specific code by, for instance, putting them into a separate library [170].

As already mentioned in Section 4.1.1, GUI code has its own difficulties. Choose an established design pattern like Model-View-Presenter (MVP) [230] (used by Mantid [180]) or one of its variations (MVC, MVVM [56,271]). These patterns describe a layered architecture where the screen "view" forms a thin top layer with minimal logic. The "model" component must not be confused with the data model of the underlaying physics code. GUI and "business logic" (physics) should be strictly separated. The "model" component of the GUI design patterns is only meant as a thin layer that exposes the data structures and functions from the GUI-independent physics module to the "view".

There should be no cyclic dependencies [170]; this holds equally for linking libraries [185, Ch. 14], for including C/C++ headers from different source directories,[4] and for Python module imports [251].

Sometimes, part of our code solves a specific scientific problem and therefore may also be of interest for reuse in a concurrent project or in other application domains. In this case, we recommend configuring the code as a separate library (or Python module), moving it to a separate repository and publishing it as a separate project. For code that includes time-consuming tests this has the additional advantage that it saves testing time in dependent projects.

5.4.2. External components

> Wherever possible, use standard software components.

Wherever possible, we reuse existing code. Preferentially, we choose solutions from well established, widely used components that can be treated as *trusted modules* [311, Section 8.4] for which we need not write our own unit tests. For compiled languages, external components are typically used in form of libraries that must be linked with our own application; the equivalent in Python are modules that are included through the *import* statement. In either case a dependency is created that adds to the burden of maintenance (Section 5.10.1) and deployment (Section 5.10.3).

For "dependency hygiene", it "can be better to copy a little code than to pull in a big library for one function" [223]. Header-only C++ libraries may seem good compromise in that they create a dependency only for maintenance, not for deployment, but they come at an expense in compile time.

Standard C or C++ libraries for numeric computations include BLAS [315] and Eigen [70] for linear algebra, FFTW for Fourier transform [86,87], and GSL (GNU Scientific Library) [82,110] for quadrature, root finding,

[4] Directory include graphs can be generated with the documentation tool Doxygen [64].

fitting and more. In Python, we consequentially use NumPy [121,199] for array data structures and related fast numerical routines. The library SciPy [262], built on top of NumPy, provides algorithms "for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations and many other classes of problems" [308].

As for a GUI library, the de facto standard in our community is Qt [243], which started as a widget toolkit and has become a huge framework. It is written in C++, but wrappers are available for a number of other languages, among them Python [244], Go [291] and Rust [252]. Qt, as well as all libraries mentioned in the previous library, is open source. It can be freely used in free and open source projects. As commercial licenses are also available, a broad industrial user base, especially from the automotive sector, will sustain Qt for many years to come.

5.4.3. Design principles

Learn about object-oriented design principles.

The object-oriented programming paradigm suggests ways of organizing data and algorithms in hierarchies of *classes*. As object-oriented languages came into wider use, it was learned that additional design principles are needed to achieve *encapsulation*, *cohesion* and *loose coupling*, and thereby arrive at a clear and maintainable code architecture [57,182,185,190,293]. In the C++ community, some principles became known by the mnemonic SOLID [185]:

- single responsibility: a class should have one and only one reason to change;
- open/close: classes should be open for extension, closed for change;
- Liskov substitution: class method contracts must also be fulfilled in derived classes;
- interface segregation: classes should be coupled through thin interfaces;
- dependency inversion: high-level modules and low-level implementation should both depend on abstract interfaces.

Other often quoted rules include:

- law of Demeter [169,183]: Class methods should only call methods that are closely associated with their own class or with their parameters;
- prefer composition over inheritance [187, Section 6.3]; the latter should only be used for "is-a" [25] relationships.

However simple and plausible these short summaries may seem, a true understanding requires considerable experience. How these principles should be applied in refactoring can be a good topic for code review and group discussions.

5.4.4. Design patterns

Know design patterns and use them with judgement.

Patterns "are simple and elegant formulations on how to solve recurring problems in different contexts" [296]. A catalog of 23 design patterns for object-oriented programming has been established in the widely known "gang-of-four book" [91]. They go by names like *adapter, command, decorator, factory, observer, singleton, strategy and visitor pattern*. Developers should know these patterns sufficiently well to recognize when they can be used. When a pattern is used this should be made transparent in the code by, for instance, choosing class names that contain the pattern name.

Design patterns are not always the optimal choice because they are meant to lay ground for later generalization. We have seen code bloated by premature generalization for hypothetical use cases that never came into being. Design patterns, therefore, should not be used just because they *can* be used; they should be preferred over equivalent, self-invented, solutions, but not over obviously simpler designs [11,12,30,126,295].

5.4.5. Error handling

> Choose consistent ways of error handling, usually using exceptions.

A policy for handling errors and non-numbers (Section 5.5.1) should be laid out early in the development or renovation of a software module, as it can be hard to revise this later in the project [286, E.1].[5] When integrating modules that have different policies, it may be necessary to wrap each cross-module function call to catch errors or non-numbers. This applies in particular to calls to the C standard library and to other mathematical libraries of C or Fortran legacy.

There are different ways to signal that a function cannot perform its assigned task:

- return a non-number,
- set a global error variable (*errno* in C [174], *ifail* in Fortran [226]),
- return status information or an error code,
- invoke an error handler [31],
- raise an exception.

Among these, exceptions are the natural choice [286, E.2] if supported by the programming language, as is the case for C++ and Python. By default, exceptions are passed silently from caller to caller. Occasionally, it may be in order for a caller to catch and re-throw an exception to enrich the error message.

The ultimate reaction to an exception depends on the control mode (Section 4.1). Batch processing programs should terminate upon error. In interactive sessions, immediate termination should be avoided if it is possible to return to the main menu without risk of data corruption. This gives users the opportunity to continue their work or at least to save some intermediate results before quitting the program.

To avoid data corruption when a computation terminates abnormally, subroutines should transform data out-of-place rather than in-place. We note, however, that in-place transformation can be simpler and faster when very large data blocks (e.g. image volumes of tens of GBytes) are handled.

5.4.6. Assertions

> Use assertions to detect programming errors early in runtime.

In our code we use assertions to check invariants and other implicit conditions that need to be true for a program to work. A failed assertion always reveals a bug. Function arguments can be checked by assertions if the caller is responsible for preventing invalid arguments. Invalid user input must be handled by other mechanisms. Assertions help to detect bugs early on, and to prevent software from delivering incorrect results. Furthermore, assertions are valuable as self-testing documentation [84,320]: *Assert(x>0)* is much better than a comment that says *x is expected and required to be a nonzero positive number*. Assertions are particularly useful for checking whether input and output values satisfy the implicit contract between a function and its callers [67]. In languages that lack strong typing, assertions can be used to check and document constraints on the supported type of function arguments; in Python, however, the standard approach for this is to use native *type hints* combined with one of the many tools for static or dynamic checking [50,94].

There is a tradition of disabling assertions unless compiling for debug [173]. This makes sense if crashed runs can easily be replayed in debug mode. That condition, however, is not fulfilled if users are running our software on computers and with inputs of their own. To analyse bug reports, instead of exactly replaying a user's interactive session, we often depend on analysing error messages. If this message comes from an assertion, then it is closer to the bug, and therefore more informative, than any downstream consequence of the violated contract could

[5] The Google C++ Style Guide [115] forbids the use of exceptions but goes on explaining that this is mostly for compatibility with extant code: "Things would probably be different if we had to do it all over again from scratch".

be. Therefore we recommend keeping all assertions active in production code, except perhaps in time-critical innermost loops.

In the spirit of Fig. 2, bugs are not fundamentally different from errors in input data or user commands. Therefore, exceptions raised by failed assertions should be handled in the same way as any other exception, except for an error message that invites the user to submit a bug report. This can imply that we have to override the native *assert* command or function of a programming language or standard library.

5.5. Computation

Having looked at the textual aspects of the code in Section 5.3 and at architecture in Section 5.4, we now propose a few more guidelines that specifically address computations.

5.5.1. Non-numbers

> Choose consistent ways of handling non-numbers.

The non-numbers *NaN* and *Inf* are part of the IEEE floating-point arithmetic that is implemented in all our target hardware [80,114]. Non-numbers can arise from input parsing, from overflow, from division by zero, and from domain errors in functions like *sqrt* or *log*. Project policy ought to regulate whether non-numbers are to be caught early, or whether they are allowed to propagate through further data treatment. If non-numbers are allowed to appear in output files this must clearly be communicated to users. Tests should cover all possible combinations of numbers and non-numbers in input and output.

5.5.2. Physical units

> Choose fixed units for the internal representation of dimensioned quantities. Clearly communicate units in programming interfaces and in input or output operations.

Missing or incorrect conversion of physical units is a recurrent source of errors in technical software.[6] We are fortunate enough to have no imperial units in our field, but we do have radians and degrees for angles, ångström and nm for neutron wavelengths and atomistic lengths, μeV, meV, GHz, THz and s^{-1} for energy transfers, mm, cm, m and dimensionless detector pixel indices for instrument coordinates. Therefore, in software use and development, we need to be aware of units.

Software extensions have been devised to make dimensioned quantities typesafe [83,146,234,256,303]. They provide a safeguard against unit errors *within* a code, but not in input and output. In our field, a C++ *Units* library [303] is used for all variables in the new array framework *scipp* [261]. In other projects, we rely on convention for the consistent use of units within one software module (library or application). For instance, *BornAgain* core has angles in rad, atomistic lengths in nm, macroscopic lengths in mm, while the GUI has angles in degrees, and also supports lengths in ångström.

To prevent conversion errors in input or output, consider the following measures:

- To guard wall clock time against misunderstandings of time zones and discontinuities from changes between summer and winter time, internally use UTC. Use the standard software modules *std::chrono* in C++ and *datetime* in Python.
- In the API documentation, indicate the dimension for all function arguments and return values. Alternatively or additionally, consider including the unit in the name of the function or the function argument, like *wavelength_nm* or *twotheta_rad*.

[6] Most famous perhaps is the loss of the Mars Climate Orbiter, which had for root cause a miscommunication between two computer programs: A navigation software wrote impulse data in pound-force seconds to a file, which was then used by a trajectory control software that assumed, in accord with written specification, that impulses were in Newton seconds [282].

- In a GUI, and in data plots, along with any dimensioned label show its unit.
- Users of the BornAgain Python API are advised to write *set_wavelength(0.62*nm)* for the sake of self-documentation, even if the constant *nm* is just 1.
- For data exchange, prefer self-documenting formats that have obligatory fields for physical units.

5.5.3. Profiling and performance optimization

> Avoid premature optimization. Measure performance. Weigh performance gains against code complexity.

When processing huge data sets, fitting complex models, or interactively visualizing multidimensional arrays, execution speed becomes a major concern. Perceptible latency impairs the user experience. Execution times of hours or days make even batch processing difficult. For these reasons we may wish to *optimize* our code for execution speed.

A task is said to be "embarrassingly parallel" if it lends itself to parallelization with little extra effort. If this is the case, then we may want to plan from the onset for parallel processing.

In general, however, "premature optimization is the root of all evil" [131,160]: It makes the code more complicated and less readable, and often for no perceptible gain in speed. Even if developers have perfect overview of a software's architecture, they will often guess wrong where the bottlenecks are. Empirical *profiling* is needed to find out where most execution time is spent. Possible causes of performance bottlenecks [55,135] include algorithmic complexity, memory (RAM) shortage (resulting in swapping), inefficient caching, input/output congestion and threading deadlocks.

Profiling has its own pitfalls [68,89]: Some profilers add heavy instrumentation to the code, making the executable so slow that it cannot be run with realistic input. If CPU time instead of wall time statistics is collected, bottlenecks in input or output operations may be overlooked. If only the program counter is analyzed, one will see how much time is spent in low-level system routines, but not from where they are called. The best generic profiling technique is sampling the call stack at fixed or random time intervals. Under C++, this can be done with "very little runtime overhead" [159] with the CPU profiler of *gperftools* [119]. Under Python, recommended [328] sampling profilers include *Pyinstrument* and *Py-spy*; *Palanteer* even works for Python wrapped C++ code.

The accuracy of such measurements is limited by the usual $\sqrt{N}$ sampling error. More importantly, results may vary with input data and parameterization. It is therefore essential to develop and maintain a representative benchmark suite for use in profiling, in regression tests (Section 5.8.7) and in assessing the success of optimization steps. Also investigate how performance scales with data size. If code has become more complicated, but not appreciably faster for relevant kinds of input, then better revert the attempted optmization.

Optimizing strategies include:

- If the bottleneck is algorithmic complexity, can we replace an algorithm of our own by an optimized solution from an established library? Think of the linear algebra routines from BLAS, which are thoroughly optimized for efficient caching. Or, conversely, should we replace a generic algorithm by a more specific solution of our own?
- When working with Python make sure that all inner loops are executed within NumPy, or be prepared to write time-critical parts as C or C++ modules [200,213].
- Execute computations in parallel, using the full power of modern multi-core processors (vectorization and parallelization) [34,161], and possibly also of GPUs (graphics cards) [27,207].
- Avoid repetition of time-consuming computations or input/output operations by using lookup tables, caches, or memoization [138,192].

See Section 5.6.2 for admonition against pointless manual microoptimization in compiled languages.

5.5.4. Numeric approximations

> Choose consistent numerical accuracy requirements and cutoffs.

While our raw data are detector counts, hence integers, after normalization they become floating-point numbers. Most scattering data have less than 8 significant digits and therefore fit into IEEE single precision. Nonetheless, in our software we usually process them with double precision to prevent cancellation errors. Only if memory is the bottleneck and cancellation is not of concern it may be worth going back to single precision.

Some numeric computations can be accelerated by relaxing accuracy requirements. Many iterative approximations take a parameter that specifies the required accuracy. Functions can be cut off at arguments where return values become negligible. Choose these parameters such that the impact on any output is by one or two orders of magnitude smaller than the uncertainty due to input error bars.

All procedures involving numeric approximations should be covered by characterization tests (Section 5.8.5). These tests need to admit a certain tolerance of output values, which must be consistent with the intended numeric accuracy.

5.6. C++ guidelines

5.6.1. Compiler warnings and static code analysis

> Do not tolerate compiler warnings. Use static code analysis.

Compiler warnings hint at language constructs that are not forbidden by the standard but for some reason are considered problematic. Warnings can indicate violations of fundamental design principles (code smell), possible runtime errors (undefined behavior, overflow, invalid downcasts), use of deprecated language or library features, name shadowing or name conflicts, issues with code style and readability, unused variables, unused code, and other problems [265].

Our advice is to adopt zero tolerance for warnings [28] because compiler output cluttered with warnings is distracting, makes a bad impression, and entails the risk that important information is overlooked. This policy can be enforced with a compiler option (*-Werror* for gcc and clang) that treats warnings as errors.

Start by enabling all warnings (*-Wall* or even *-Wpedantic*), then deselect those that are ignorable by project policy. Warnings from external dependencies should not be deselected globally but be silenced by applying a *#pragma* to the include. Once this is settled, it is straightforward to keep the build clean of warnings [28]. New warnings may appear when the compiler is upgraded.

The diagnostic capabilities of compilers are restricted to warnings that can be detected in passing, without impairing the speed of compilation. Deeper checks can be done with dedicated tools, called static code analyzers or linters. For C++, there exist the powerful open-source linters *Cppcheck* [53] and *clang-tidy* [39]. The latter supports hundreds of different rules from various guidelines. Just as for compiler warnings, some work must be invested to deselect rules that make no sense for us. Once this is done, zero tolerance should be enforced by nightly CI runs.

5.6.2. Rely on compiler optimization

> Recognize the amazing power of compilers and avoid pointless manual optimization.

Many developers in our field have received little or no formal training in computer science and numerical mathematics, and know little about compilers and processors. They tend to waste time on putative optimization steps that are completely pointless because they are automatically done by any optimizing compiler (draw constants out of a loop, eliminate local variables, inline functions, replace division by 2 by a bit shift). To find out what amazing transformations modern compilers can do, read Refs. [81,112], and visit the online *Compiler explorer* [113] to gain hands-on experience. Try *link time optimization* so that the compiler can see optimization opportunities that involve code from external libraries [112].

5.6.3. Build system

> Configure C++ projects with CMake.

C++ source files would be incomplete without configuration files that steer the building, testing and packaging of the software. These configuration files are in a domain-specific language defined by a build automation tool. We recommend *CMake* [41], which is widely used and can deploy to both Windows and Unix-like platforms.

Due to a long history and almost complete backward compatibility, many deprecated commands continue to be supported by CMake. CMake scripts in extant open-source projects often mix different styles and use outdated constructs. Beware of taking such code as template. When starting a new project, invest some effort in learning modern CMake [257,264,289] and write clean scripts from scratch.

CMake should be run from a dedicated build directory. This is called out-of-source build, in opposition to in-source, which pollutes the source directory up with compilation output. Since in-source builds are more likely to be accidental than intended, we recommend configuring CMake such that they are forbidden [24].

CMake can be used together with the compiler cache *CCache* [33]. CCache wraps compiler calls, caches inputs and skips recompilation of sources that have not changed. This can substantially speed up development cycles.

5.7. Python guidelines

5.7.1. Pythonic style

> Learn idiomatic Python.

Thanks to its simple syntax, Python has a reputation of being easy to learn. This simplicity, however, can be deceptive for experienced programmers who are coming from other languages. They are at risk of only using a subset of Python that falls into their comfort zone, per the old joke "the determined Real Programmer can write FORTRAN programs in any language" [229]. To make full and idiomatic ("Pythonic") use of the language, it is necessary to learn iterators, generators, list comprehensions, positional and keyword arguments, Python's flavor of object orientation, metaprogramming with decorators and introspection. Good sources of information are advanced textbooks [242] and best-practice code examples [18,249].

5.7.2. Static code analysis

> Static code analysis is particularly important in Python.

Static code analysis is even more important in Python than in C++ because Python has neither strong typing nor a compiler, so that faulty statements in many cases cannot be detected before they are reached during execution. A detailed comparison of different Python code analyzers has been provided in a recent Bachelor's thesis [158]. The free and open-source tools Pylint [236] and flake8 [78] are excellent so that there is little incentive to try out commercial alternatives [5]. Similar to the C++ linter *clang-tidy* (Section 5.6.1), they support lots of different rules. Some team effort should be invested in selecting a set of rules that are appropriate for a given project. After that, a zero tolerance policy is easiest to sustain, for the same reasons as given above (Section 5.6.1).

5.8. Tests

5.8.1. Regression test suite

> Accumulate tests in a regression test suite.

All but the most trivial software should include tests. Tests should accumulate, with new tests being included each time new functionality is added or a bug resolved. There is rarely a good reason to remove a test. Accumulated tests have been called *regression* tests because they prevent the software from falling back, from losing functionality that is covered by these tests. Since there are no tests in our code bases that do *not* belong into this category, we consider the attribute "regression" redundant and rarely use it.

Comprehensive test coverage is a precondition for safe and efficient refactoring and, in turn, for maintenance and evolution. This is how "legacy" came to be defined as software without tests [76]. For these reasons, we are discussing tests under the heading "product requirements", whereas the infrastructure for running the tests has found its place among the procedural guidelines (Section 3.2.4).

In a typical source tree, tests are organised in a top-level directory called *tests* or similar. There should be a simple way to run all automatized tests, or some selection thereof, like the command *ctest* povided by the build system CMake [52] for C++ and *pytest* or *unittest* for Python [237].

Which parts of code are covered by tests should occasionally be investigated with a *coverage* tool [3]. Beware that coverage only measures invocation, not verification, of code lines, and, as for any metric per Goodhart's law, this must not become a goal in itself [21, p. 222]. In fact, empirical evidence suggests that high unit-test coverage is no tangible help in producing defect-free software [7]. However, a helpful side effect of determining coverage, is that it can draw attention to code that is no longer used anywhere and should be removed.

Colleagues experienced in running software carpentries have argued that the "breathtaking diversity of scientific code" makes it difficult to give any generic advice on how to write meaningful tests [319]. Nonetheless, we could agree on the following guidelines for our more homogeneous application domain.

5.8.2. Functional tests overview

Run fast tests before merging code, slow tests overnight.

The standard taxonomy of functional tests is based on *scope* (or *granularity* [292]): *unit, integration, system* tests (to keep things simple, we do not distinguish *system*, *acceptance* or *end-to-end* tests). However, when the focus is shifted from writing to running tests, then a more important distinction is made by *size*, i.e. by execution speed [21]. We recommend distinguishing three *sizes* of test:

Small tests are so fast that they can be run by each developer after every recompilation. Typically they run in a single process and are not allowed to sleep, to perform I/O operations, or to make any other blocking call [21, p. 216]. Altogether, they should take less than a minute. They should include enough unit tests to cover the bulk of functionality [21, p. 220]. Additionally, there should be some representative end-to-end tests, possibly from tutorials (Section 5.9.3), tweaked for fast execution.

Medium tests have to pass on our CI servers before a merge request (GitLab) or pull request (GitHub) is accepted into the shared code base. Altogether, they should take not much more than 10 minutes on the dedicated CI machines, as otherwise, in our experience, the collaborative workflow starts to suffer. To accelerate these tests, it is often advisable to reduce array dimensions.

Large tests are needed to check the treatment of real-life data, or to verify numerical computations in multidimensional parameter space. Altogether, they may take several hours, and are typically run overnight. Of course, specific large tests should be run before acceptance of a merge/pull request if they are explicitly covering parts of the code that have been changed.

The coding can be simplified and standardized by using a test framework. Under C++, this could be *Google Test*, the more lightweight *Catch2* [36,287], or *QTest* [245] with functionality for mouse and keyboard simulation to support basic GUI testing. Under Python, either *Pytest* or *PyUnit* (module name *unittest*) [175,189] could be used.

5.8.3. Unit tests

To write good unit tests, learn to write testable code.

Many interesting discussions can be found in the web regarding the question what should be covered by unit tests. For instance, should we test getters and setters? Kent Beck, the creator of "extreme programming", has given this answer: "It is impossible to test absolutely everything, without the tests being as complicated and error-prone as the code... You should test things that might break. If code is so simple that it can't possibly break, and you measure that the code in question doesn't actually break in practice, then you shouldn't write a test for it" [19]. And elsewhere: "I get paid for code that works, not for tests... If I don't typically make a kind of mistake (like setting the wrong variables in a constructor), I don't test for it... I'm extra careful when I have logic with complicated conditionals. When coding on a team, I modify my strategy to carefully test code that we, collectively, tend to get wrong" [279].

A more formal approach refers to the *cyclomatic complexity* of function, that is the number of different possible pathways due to conditional statements. Basically, there should be as many different test cases as there are pathways [266]. However, we make an exception for a complexity of 1; if a function contains neither a branch nor performs any other complicated operation, then we advise against unit code coverage. Such functions are better tested indirectly, through their callers, in integration or system tests. No unit tests thus for plain getters and setters, nor for constructors or destructors that perform no extra work.

This last clause brings us to the question how to unit test a function or class that is tightly coupled to other classes. Part of the answer is that to write good unit tests we need to adapt our coding style, we need to "write testable code" [129]. Specifically, constructors and destructors should do no real work, call chains should not walk long object graphs, modifiable singletons should be avoided, classes should not be too large [130].

Not all interdependencies can be factored away though. For meaningful unit tests, we still need to allow functions and classes under test to depend on other functions, classes, and external data. If these external dependencies are too heavy for small (fast) tests, then we need to replace them with a *test double*. See Ref. [299] for a detailed discussion of this technique, and especially of the pitfalls of overused *mocking* frameworks.

5.8.4. Numerics tests

Numeric computations need broad test coverage.

Numeric computations, unless performed by trusted modules (Section 5.4.2), need comprehensive unit tests. To do better than black-box testing, we may instrument the code so that tests can determine parameter values where the algorithm in use changes. In this way, continuity and smoothness can be checked at the points where failure is most likely [324]. Such numeric code should then be packed as a separate library and in a separate project so that the time consuming tests must not be rerun over and again in dependent applications.

5.8.5. Characterization tests

Use characterization tests to prevent regression of numeric results.

To validate our software end to end we compare results with other programs, consider special parameter values, investigate how the output scales with input data and parameters, and so on. We note that this type of testing relies on our thorough understanding of the application domain. Some of these manual checks can and should be converted into automatized system tests that are preserved in our regression test suite. Others, like the comparison with third-party software, are more difficult to automatize, or would make the test suite too complicated or too slow. Instead, we suggest performing these tests manually, documenting them, and then ensuring by other tests that future changes of our software do not corrupt what has once been validated.

In the absence of an independent *oracle* [284, Ch. 4.1], the best end-to-end validation is through *characterization* tests [76, Chapter 13], also called golden master, output comparison, or snapshot tests [276]. To build a characterization test, we characterize the present state of our software by running it for different inputs and collecting the outputs. These outputs are added as reference data to our source repository. A characterization test runs the current version of our software and asserts that the output still agrees with the pertinent reference output. If results

disagree, then it is up to the developer to investigate whether the latest changes in the code have introduced a bug or, on the contrary, have fixed a bug in the reference version. In the latter case, the reference data are replaced by the current output.

When the output consists of text and integer numbers, one can simply check for literal agreement with the reference data. However, when it comes to floating-point numbers exact agreement should not be expected as numeric results may change under legitimate refactoring operations, and even the same software version may yield different numeric results under different compilers or on different hardware [62]. Therefore, characterization tests that check the invariance of floating-point outputs need to admit a reasonable margin for relative deviations. If a code involves random numbers (e.g. for Monte-Carlo integration), seeds must be set to fixed values.

5.8.6. GUI tests

> GUIs can and should be covered by automatic tests.

Automating GUI testing is difficult but important, and therefore a subject of intense research and development (e.g. [15,197] and references therein). Tests need to sample the input space, which consists of external data, parameterization, and user actions like mouse movements, clicks, and keyboard inputs.

For automated testing, the application must be fed with emulated user actions. They are either generated by automatic methods that treat the software under test as a black box, or they are constructed to represent realistic sessions. Often, they are captured from interactive input. Of all the tools available for capture & replay testing, the most promising for our needs are probably the free *GNU Xnee* [111] for the X11 environment and the commercial *Squish* [88] for the Qt GUI framework. However, we do not have enough experience in their use to recommend one or other of these tools. For the data reduction software *Steca* we are experimenting with a wrapper library *libQCR* [325] that equips Qt widgets with functionality for scripting, capture, and replay; however, it is not yet clear how this approach would scale to larger projects.

Even without emulating widget interactions, a substantial degree of test coverage can be achieved if the software adheres to a design pattern like Model-View-Presenter (Section 5.4.1) where the widgets form a thin top layer with minimal logic. As the other layers of the GUI code are unaware of the widget toolkit and its screen representation, they can be covered by conventionally coded tests [85].

5.8.7. Performance tests

> Maintain meaningful performance tests to detect regression of execution speed.

Among non-functional tests, most relevant for us are *performance tests*. They are closely related to the optimization work discussed in Section 5.5.3. Performance tests are benchmarks together with additional control scripts that safeguard against performance regression. They may reuse code from functional tests. They may have unit, integration or system scope. Unit and integration tests detect, for instance, inefficiencies in numerical computations, in caching, threading, I/O or network access. System tests mimic actual software usage. Performance tests concentrate on what is key to user satisfaction, namely functionality that is time critical and frequently used.

5.9. Web presence and user documentation

The user documentation should consist of a home page, tutorials, references and a comprehensive overview in form of a peer-reviewed publication. Tutorials and reference manuals should be available online and be version controlled. This facilitates maintenance, helps users who are working with an old software version, and is indispensable for the sake of reproducible science.

5.9.1. Home page

> Each project should have an informative homepage in the World Wide Web.

The landing page should give quick access to

- essential metadata: authors, maintainers, license, recommended citation, maintenance status;
- download location and installation instructions;
- user documentation;
- news;
- source repository, issue tracker etc;
- contact mail address, contact form, and/or discussion forum.

For GitHub and GitLab repositories, the simplest way to set up a web presence is through MarkDown files in the source tree. For small libraries, one single "README" page may suffice. However, for user-oriented software it is worth investing some effort in a visually attractive landing page. Further content is best converted from a markdown language to HTML by a static site generator like *Hugo* [139] or *Jekyll* [149]. For deployment, many cloud services offer free hosting; for instance, GitHub offers web space through GitHub Pages [99]. Local hosting is not difficult either.

5.9.2. Peer-reviewed high-level description

> Present the software in a peer-reviewed overview paper.

A peer-reviewed overview paper is the established way to present a scientific software to the academic community and to give confidence that the software can be relied upon in downstream research. For data reduction software, a natural choice is to publish the paper in a methods oriented journal; for data analysis, a journal related to the application domain may be more appropriate.

For the *Journal of Applied Crystallography* (JAC) to consider a software paper, "the utility of the program and adequacy of the documentation should normally have been proved by the successful use of the program by at least two independent users" [153]. This is an excellent rule, since there is little public interest in papers on software that is not yet production ready.

For further advice, we continue to cite the notes for JAC authors: "A description of the purpose, strategy, computer language, machine requirements, input requirements and type of results obtained should be included. A Computer Program article should make clear the novelty of the program and its scientific context. Articles on significant updates to existing programs will also be considered ..." To reduce the need for updates, a journal paper should concentrate on fundamental ideas and avoid details that are better addressed by a user manual.

Complementary to this type of scientific publication, authors should also consider documenting the engineering aspects of their software in technical outlet such as the *Journal of Open Research Software* [147] or the *Journal of Open Source Software* [148].

5.9.3. Tutorials and references

> Provide tutorials for linear reading and references for random access.

From their experience with consumer applications, users are accustomed to explore a new GUI by trial and error. However, to master a complex, and perhaps not so well designed, scientific GUI, they may need some guidance. Textual tutorials, lengthy and full of screenshots, are not ideal as they are cumbersome to read and to maintain. A better alternative could be tooltips and other help facilities that are tightly integrated in the GUI; this approach is consequentially pursued in the *easydiffraction* GUI [69]. Another alternative, for software that is reasonably stable,

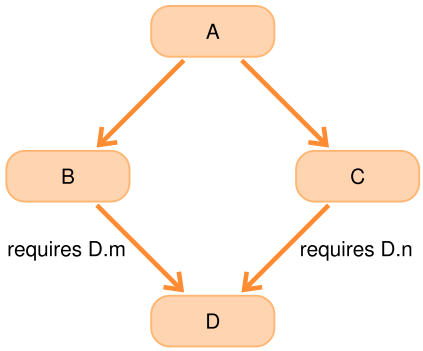

Fig. 4. Let A, B, C, D represent four software components. The arrows stand for "depends on" or "requires" relations. There is a *diamond problem* if A, through intermediary of B and C, depends on different versions of D (m or n) that are incompatible with each other. This situation arises if some, but not all dependent components are updated after a low-level interface has changed.

may be video tutorials and other e-learning formats. The Photon and Neutron E-learning platform *PaN-learning* [208] provides an e-learning module [209] on how to make videos with *Open Broadcaster Software* [202].

For a command-line or scripting interface, tutorial chapters should be built around actual scripts. As our software undergoes revisions, we need to make sure that all scripts still work, and still produce the same answer as promised by the documentation. This is best accomplished by covering all scripts with regression tests. Some tricks may be in order to accelerate scripts when run from routine tests, for instance by reducing the number of output pixels.

Command references, just as developer documentation of low-level APIs (Section 5.3.4), should wherever possible be generated out of the code, using tools like *Doxygen* [64] or *Sphinx* [26].

5.10. Deployment

5.10.1. Dependency problems

Keep up with changing external dependencies.

Any nontrivial application depends on external libraries. A few of those may be written and maintained by ourselves, most are not. As installed libraries may be updated by the operating system, package managers, or installers of other software, it is quite possible that our software stops working because of broken dependencies. This is discussed under the name *software rot* or *software collapse* [133]. It poses severe problems for the deployment of software, and for the reproducibility of work done with a specific software version.

Additional difficulties arise if dependencies between software components do not form a tree, but instead are more complex with some components being connected through more than one path. The paradigm for this is the *diamond dependence*, explained in Fig. 4. For a large system that involves many components, the diamond problem easily explodes into what is known as *dependency hell* [59].

A natural reaction is to revert changes that broke the system, and to freeze dependencies at the latest consistent state. This, however, can only be a short-term remedy, and in the longer term (for essentially the same reasons as in Section 5.2.3) one has no choice but to keep up with the evolution of the infrastructure. We therefore tend to use the latest versions of operating systems, compilers, tools and libraries, and upgrade our software as suggested or required by the external dependencies [322].

5.10.2. Reproducibility vs backward compatibility

Maintain backward compatibility regarding input data and fit models.

In the literature, reproducibility in computational science [133] is mostly considered from the standpoint of a scientist who integrates third-party code to set up a simulation or to process data for one specific research project. Even if the scientist's own code and protocols are preserved and well documented it is not guaranteed that they still work, and work correctly, some years later on newer hardware and with newer system libraries [220].

Here we are concerned with data treatment software that cannot be frozen, as data reduction programs need to be updated when instruments are modified, sample environment equipment added or data formats changed; therefore, data analysis software needs frequent updates to meet ever new user demands. It would require an unrealistic amount of effort to keep all previous version of such software operable while library dependencies and operating systems are evolving. Instead, we should strive for *backward compatibility*, with new versions of our software still being able to process old raw data and to run old fit models (Section 4.2).

5.10.3. Deploying compiled applications

Use installers or package managers to deploy compiled software.

Whatever our internal policy about dependency handling, the bigger challenge is to deploy our software to end users so that *they* do not have to fight dependency hell. Unless the software is uniquely run on servers of our own (SaaS, Section 5.2.1), we need to enable users to install it on their own machines. This means we have to support a carefully chosen range of operating systems, and their respective versions, with the software needing to coexist with an unknown combination of libraries and other applications already installed on the computer.

Since we want to produce truly open-source software, we also need to provide instructions for building from source. However, users will increasingly expect a binary distribution of the software to be provided for each of the popular operating systems, removing the need for them to spend time learning how to build the software.

For very simple programs it may be adequate to distribute just one binary executable per target operating system, with all library dependencies statically linked. This precludes all problems with finding external libraries.

For larger programs, the present state of the art requires dynamic linking, where libraries are not contained in the application binary, but are loaded at runtime (shared libraries under Unix, dynamic link libraries (DLL) under Windows, modules in Python). Dynamic linking reduces network, disk and memory consumption, but has also some disadvantages [61], most importantly the need to keep the right library version available and findable on each target system.

Under Windows and MacOS, the standard method of software deployment is to provide an *installer* that bundles a binary executable with its libraries dependencies, and takes care of installation, upgrades or removal of the software components. Installers are preferred over direct download of executables because of concerns about file size, user convenience, compatibility with other installed software, elevated privileges and special services [277]. In CMake based projects, the installers can be created automatically through the program *cpack* [51]. An interesting alternative to native Windows or MacOS installers could be the cross-platform installer *Zero Install* [331].

When running our installer, users may need to acknowledge a warning that they are executing software from an untrusted source. To get rid of this warning, a facility has to purchase signing certificates from Microsoft and Apple, and a CI step for *code signing* has to be added. We did so for *SasView* [254] and *Mantid* [9,178].

Instead of an installer, we may also deploy our software as a *package*, and thereby leave the handling of dependencies to a package manager. This is the standard way of binary distribution under Linux. Unfortunately, different Linux distributions are based on different package formats. Given our limited resources, it may suffice to provide a *Debian* package since most users are running a Debian based distribution (Ubuntu, MX, Mint, ...), and there are also ways to install Debian packages on rpm based and other distributions. Thankfully, there is a Debian Photons And Neutrons Team [58] at Soleil Synchrotron that builds packages for a large number of scientific applications, and feeds them into the official Debian distribution.

Package managers and repositories also exist for Windows [195,270] and Mac [269]. Some are free and open source, and have GitHub based repositories, which facilitates the submission of packages. *Homebrew* [137] and *MacPorts* [176] are widely known among Mac users, and are supported by active communities. For Windows, *Scoop* [263] and Chocolatey [37] look promising, but we cannot base a recommendation on practical experience of our own.

5.10.4. Deploying Python applications

Use pip or higher-level methods to deploy Python applications.

As the Python maintainers acknowledge, "packaging in Python has a bit of a reputation for being a bumpy ride. This impression is mostly a byproduct of Python's versatility" [240]. They distinguish no less than seven different layers of dependencies that may be included on top of the Python code that is to be deployed. From raw code to SaaS, the different deployment methods are:

- Provide the Python code, which users will then run in their own Python environment. Include files *setup.py*, *setup.cfg*, *MANIFEST.in*, *pyproject.toml* to configure the project and assemble a "source distribution" and a "built distribution" (*wheel*) [241]. Upload the distributions to PyPI, the Python Packaging Index [239]. Packages from PyPI can be installed with the Python utility *pip*.
- Use PEX [221] to combine the Python code with all its module dependencies in a .pex file, which is a specially configured zip archive. This is probably the least common method, and is not supported under Windows.
- Instead of pip, use the cross platform package and environment manager *Conda* [47]. Contribute the package to the cross-language repository Anaconda [4]. In this way, other scientific code from Anaconda can be integrated in a distribution. To resolve performance issues with Conda's dependency resolution [1], consider *Poetry* [225].
- Use PyInstaller [235] or equivalent tools [240] to freeze the Python application into a stand-alone executable.

Pushing the integration further, the remaining deployment methods are no longer specific to Python:

- Include system libraries to build a binary *image*, using e.g. *AppImage* [8], *Flatpak* [79], or *Snap* [272] to generate distribution-independent Linux packages.
- Use *Docker* [63] or *Singularity* [164] to put a binary image into a sandboxed *container*.
- Use a *virtual machine* to run an image or a container under whatever operating system.
- Deploy only to your own hardware, and run your software as a service (Section 5.2.1).

These different levels of packaging all have their own caveats and there is no "one size fits all" solution to be recommended at the time of writing.

6. Conclusion

In this work we presented engineering guidelines for data treatment software, to foster sustainability and collaboration. The set of guidelines emerged from a questionnaire in the SINE2020 project [181] and was augmented to cover all aspects of the software development process. In lieu of a summary, we refer back to the boxed guidelines at the beginning of subsections. In the text that follows each guideline we give more detailed advice and explain our choices, with ample references to the technical literature.

Perhaps the most surprising aspect of this paper is its length, which reflects the depth of our technology stack and the breadth of engineering knowledge that is needed for sustainable software development. Our guidelines show how much more there is to professional software engineering than just computer programming. With professionalization comes division of work: While software engineers help to get the software right, we need scientists to continue contributing their application-domain expertise so that the right software is built.

Many engineering techniques and tools have become commonplace among software developers in general, or in the open-source or research software communities in particular. Therefore, for most points we discuss, we could easily reach a consensus. However, we also identified a few questions that need to be answered individually for each project or in each group: licensing (Section 5.1), error handling (Section 5.4.5), deployment (Sections 5.10.3–5.10.4).

Inevitably, given the rapid development of the area, some of the technical advice given here will be outdated in a few years. The participating groups now continue their cooperation in working group 4 of the League of advanced European Neutron Sources (LENS) [167], and we will continue to monitor and assess emerging alternatives, updating our recommendations as required.

Acknowledgements

We thank Robert Applin, Ed Beard, Chiara Carminata, Paul Erhart, Andrew McCluskey, Anthony Lim, Ammar Nejati, Daniel Nixon, Eric Pellegrini, Gennady Pospelov, Wojciech Potrzebowski, Emmanouela Rantziou, Dimitar Tasev, Walter Van Herck, Peter Willendrup, and two anonymous reviewers for valuable input at different stages of this work.

This work was funded by the Horizon 2020 Framework Programme of the European Union under project number 654000 [267].

References

[1] E. Alizadeh, A Guide to Python Environment, Dependency and Package Management: Conda + Poetry, 2021. https://bit.ly/3B35qzV.
[2] E.A. Allen and E.B. Erhardt, Visualizing scientific data, in: *Handbook of Psychophysiology*, 4th edn, J.T. Cacioppo et al., eds, 2016.
[3] A. Altvater, The Ultimate List of Code Coverage Tools: 25 Code Coverage Tools for C, C++, Java, .NET, and More, 2017. https://stackify.com/code-coverage-tools.
[4] Anaconda — Installers and Packages. https://repo.anaconda.com/.
[5] Analysis tools [curated list of static code analysis tools for all programming languages]. https://bit.ly/3ndvZxd.
[6] D.J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010.
[7] V. Antinyan, J. Derehag, A. Sandberg and M. Staron, Mythical unit test coverage, *IEEE Software* **35** (2018), 73–79. doi:10.1109/MS.2017.3281318.
[8] AppImage — Linux apps that run anywhere. https://appimage.org.
[9] O. Arnold et al., Mantid — Data analysis and visualization package for neutron scattering and μSR experiments, *Nucl Instr Meth A* **764** (2014), 156–166. doi:10.1016/j.nima.2014.07.029.
[10] Association for Computing Machinery, ACM Code of Ethics and Professional Conduct. https://www.acm.org/code-of-ethics.
[11] J. Atwood, Coding Horror: Head First Design Patterns, 2005. https://bit.ly/3nBtrKV.
[12] J. Atwood, Rethinking Design Patterns, 2007. https://bit.ly/3HDSpBl.
[13] Awesome Git — A curated list of amazingly awesome Git tools, resources and shiny things. https://github.com/dictcp/awesome-git.
[14] A. Bacchelli and C. Bird, Expectations, outcomes, and challenges of modern code review, in: *Proceedings of the 2013 International Conference on Software Engineering*, pp. 712–721. https://bit.ly/3Ji0TOV.
[15] I. Banerjee, B. Nguyen, V. Garousi and A. Memon, Graphical user interface (GUI) testing: Systematic mapping and repository, *Inform Software Tech* **55** (2013), 1679–1694. doi:10.1016/j.infsof.2013.03.004.
[16] N. Barnes, Publish your computer code: It is good enough, *Nature* **467** (2010), 753. doi:10.1038/467753a.
[17] R. Bast, A FAIRer future, *Nature Phys* **15** (2019), 728–730. doi:10.1038/s41567-019-0624-3.
[18] D. Beazley and B.K. Jones, *Python Cookbook*, 3rd edn, O'Reilly, 2013.
[19] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000. Meyer [191, p. 137] prefers this first edition over the second edition from 2005, where some "extreme" advice has been attenuated.
[20] K. Beck et al., Manifesto for Agile Software Development, 2001. https://agilemanifesto.org.
[21] A. Bender, Testing Overview, Chapter 11 in Ref. [323], 2020.
[22] S. Berczuk, Deploy Early and Often, Chapter 20 in Ref. [127], 2010.
[23] BornAgain — Software for simulating and fitting X-ray and neutron small-angle scattering at grazing incidence. http://www.bornagainproject.org.
[24] BornAgain, configuration file PreventInSourceBuilds.cmake. https://bit.ly/3u2AMFn.
[25] R.J. Brachman, What is-a is and isn't: An analysis of taxonomic links in semantic networks, *Computer* **16** (1983), 30–36. doi:10.1109/MC.1983.1654194.
[26] G. Brandl, Sphinx — Python Documentation Generator. https://www.sphinx-doc.org.
[27] A.R. Brodtkorb, T.R. Hagen and M.L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, *J Parallel Distr Com* **1** (2013), 4–13. doi:10.1016/j.jpdc.2012.04.003.
[28] J. Brodwall, Keep the Build Clean, Chapter 42 in Ref. [127], 2010.

[29] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975.
[30] D. Budgen, Design patterns: Magic or myth?, *IEEE Software* **30** (2013), 87–90. doi:10.1109/MS.2013.26.
[31] C++ reference, C, Error handling. https://en.cppreference.com/w/c/error.
[32] C. Carminati, M. Strobl and A. Kaestner, KipTool, a general purpose processing tool for neutron imaging data, *SoftwareX* **10** (2019), 100279. doi:10.1016/j.softx.2019.100279.
[33] Ccache — A fast C/C++ compiler cache. https://ccache.dev.
[34] V. Cesare, I. Colonnelli and M. Aldinucci, Practical parallelization of scientific applications, in: *28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE Computer Society Conference Publishing Services, 2020.
[35] S. Chacon and B. Straub, *Pro Git*, 2nd edn, Apress (Springer), 2014. Full text available online https://git-scm.com/book/en/v2.
[36] M.G. Chajdas, From Google Test to Catch, 2016. https://bit.ly/3AY7Xet.
[37] Chocolatey — The package manager for Windows. https://chocolatey.org.
[38] clang-format — A tool to format C/C++/Java/JavaScript/JSON/Objective-C/Protobuf/C# code. https://bit.ly/3ndiGgu.
[39] clang-tidy — A clang-based C++ "linter" tool. https://clang.llvm.org/extra/clang-tidy.
[40] P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley Professional, 2010.
[41] CMake — An open-source, cross-platform family of tools designed to build, test and package software. https://cmake.org.
[42] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*, Adison-Wesley, 2004.
[43] Codecademy, Learn Git. https://www.codecademy.com/learn/learn-git.
[44] CodeRefinery — Training and e-Infrastructure for Research Software Development. https://coderefinery.org.
[45] S. Collins et al., *Turning FAIR into reality. Final Report and Action Plan from the European Commission Expert Group on FAIR Data*, Publications Office of the European Union, 2018. doi:10.2777/1524.
[46] Computing in Science & Engineering. The "Scientific Programming" department is currently edited by A. Dubey and K. Hinsen. https://www.computer.org/csdl/magazine/cs.
[47] Conda — Package, dependency and environment management for any language. https://conda.io.
[48] E. Cornet, The world seen by an "object-oriented" programmer, 2011. https://bonkersworld.net/object-world.
[49] E. Corona and F.E. Pani, A review of Lean–Kanban approaches in the software development, *WSEAS Trans Information Sci Appl* **10**(1) (2013), 1–13.
[50] B.H. Cottman, 20 Type Hinting Techniques and Tools for Better Python Code, 2020. https://bit.ly/31K5wka.
[51] cpack — The CMake packaging program. https://cmake.org/cmake/help/latest/manual/cpack.1.html.
[52] ctest — the CMake test driver program. https://cmake.org/cmake/help/latest/manual/ctest.1.html.
[53] Cppcheck — A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io.
[54] Creative Commons. https://creativecommons.org.
[55] A.T. Dang, These Bottlenecks Are Killing Your Code, 2021. https://bit.ly/3FLRCvO.
[56] A.T. Dang, MVC vs MVP vs MVVM. What's different between them? 2020. https://bit.ly/3FVHrVS.
[57] D. de Champeaux, D. Lea and P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.
[58] Debian Photons And Neutrons Team. https://salsa.debian.org/pan-team.
[59] S. Dick and D. Volmar, DLL hell: Software dependencies, failure, and the maintenance of Microsoft Windows, *IEEE Ann Hist Comput* **40** (2018), 28–51. doi:10.1109/MAHC.2018.2877913.
[60] E. Dietrich, Politeness or Bluntness in Code Review? Settling the Matter Once and for All. https://bit.ly/3aVJvA3.
[61] W. Dietz and V. Adve, Software multiplexing: Share your libraries and statically link them too, *Proc ACM Progr Lang* **2** (2018), 154. doi:10.1145/3276524.
[62] Differences among IEEE 754 Implementations [Section in an online reprint of Ref. [114], added by an anonymous editor]. https://bit.ly/2ZeYti1.
[63] Docker — Accelerate how you build, share, and run modern applications. https://www.docker.com.
[64] Doxygen — Generate documentation from source code. http://doxygen.nl.
[65] V. Driessen, A successful Git branching model, 2010. https://nvie.com/posts/a-successful-git-branching-model.
[66] T. Driscoll, Matlab vs. Julia vs. Python, 2021. https://tobydriscoll.net/blog/matlab-vs.-julia-vs.-python.
[67] P.F. Dubois, Maintaining correctness in scientific programs, *Comput Sci Eng* **7** (2005), 80–85. doi:10.1109/MCSE.2005.54.
[68] M.R. Dunlavey, A Case Study of Performance Tuning, 2013. https://bit.ly/3nIardj.
[69] easydiffraction — Making diffraction data analysis and modelling easy. https://easydiffraction.org.
[70] Eigen — A C++ template library for linear algebra. https://eigen.tuxfamily.org.
[71] E. Engheim, The Case Against Descriptive Variable Names, 2017. https://bit.ly/3prIMyT.
[72] European Open Science Cloud (EOSC). https://bit.ly/3aYOQXm.
[73] European Spallation Source ERIC (Denmark & Sweden). https://ess.eu.
[74] ExPaNDS, the European Open Science Cloud Photon and Neutron Data Services. https://www.expands.eu.
[75] H. Fangohr et al., Data exploration and analysis with Jupyter notebooks, in: *17th Int. Conf. on Acc. and Large Exp. Physics Control Systems, ICALEPCS*, JACoW Publishing. doi:10.18429/JACoW-ICALEPCS2019-TUCPR02.
[76] M. Feathers, *Working Effectively with Legacy Code*, Prentice Hall, 2005.
[77] M. Fenner, DOI Registrations for Software, 2018. https://bit.ly/3469O6G.

[78] Flake8 — Your Tool For Style Guide Enforcement. https://flake8.pycqa.org/en/latest.
[79] Flatpak — A next-generation technology for building and distributing desktop applications on Linux. https://flatpak.org.
[80] A. Fog, Floating point exception tracking through NAN propagation, 2020. https://www.agner.org/optimize.
[81] A. Fog, Optimizing software in C++. An optimization guide for Windows, Linux, and Mac platforms, 2021. https://www.agner.org/optimize.
[82] L. Fortunato and M. Galassi, The case for free and open source software in research and scholarship [with reference to the GNU Scientific Library as a case study], *Phil Trans R Soc A* **379** (2021), 20200079. doi:10.1098/rsta.2020.0079.
[83] M. Foster and S. Tregeagle, Physical-type correctness in scientific Python, arXiv preprint, 2018. arXiv:1807.07643.
[84] M. Fowler, *Refactoring. Improving the Design of Existing Code*, Addison-Wesley, 1999.
[85] M. Fowler, GUI Architectures, 2006. https://bit.ly/3z3qo20.
[86] M. Frigo and S.G. Johnson, FFTW — A C subroutine library for computing the discrete Fourier transform in one or more dimensions. https://www.fftw.org.
[87] M. Frigo and S.G. Johnson, The design and implementation of FFTW3, *Proc IEEE* **93** (2005), 216–231. doi:10.1109/JPROC.2004.840301.
[88] FrogLogic GmbH, Squish — GUI Test Automation Tool. https://www.froglogic.com/squish.
[89] N. Froyd, J. Mellor-Crummey and R. Fowler, A Sample-Driven Call Stack Profiler (2004). https://hdl.handle.net/1911/96328.
[90] M.S. Gal, Viral open source: Competition vs synergy, *J Compet Law Econ* **8** (2012), 469–506. doi:10.1093/joclec/nhs013.
[91] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[92] R. Gardler and G. Hanganu, Benevolent dictator governance model, 2010/2013. https://bit.ly/3FleimN.
[93] R.C. Gentleman et al., Bioconductor: Open software development for computational biology and bioinformatics, *Genome Biol* **5** (2004), R80. doi:10.1186/gb-2004-5-10-r80.
[94] J. Giacomelli, Python Type Checking, 2021. https://bit.ly/3GkaxPv.
[95] Git — A free and open source distributed version control system. https://git-scm.com.
[96] GitHub — Where the world builds software. https://github.com/about.
[97] GitHub Actions Runner. https://github.com/actions/runner.
[98] GitHub Learning Lab. https://lab.github.com.
[99] GitHub Pages. https://pages.github.com.
[100] GitLab, Hosted projects. https://gitlab.com/explore.
[101] GitLab, Install self-managed GitLab. https://about.gitlab.com/install.
[102] GitLab, Learn. https://about.gitlab.com/learn.
[103] GitLab, Repository mirroring [GitLab to GitHub]. https://bit.ly/2ZdW5s4.
[104] GitLab, Runner. https://docs.gitlab.com/runner.
[105] GitLab instance of Forschungszentrum Jülich, subdirectory of the JCNS Scientific Computing Group. https://jugit.fz-juelich.de/mlz.
[106] GitLab instance of Institut Laue-Langevin. https://code.ill.fr/explore/groups.
[107] GitLab instance of Kitware Inc. https://gitlab.kitware.com/explore/groups.
[108] GitLab instance of Paul Scherrer Institut. https://gitlab.psi.ch.
[109] GNU General Public License (GPL) v3.0. https://www.gnu.org/licenses/gpl-3.0.en.html.
[110] GNU Scientific Library (GSL). https://www.gnu.org/software/gsl.
[111] GNU Xnee — Recorder/replay for GNU/Linux and other X11 based systems. https://xnee.wordpress.com.
[112] M. Godbolt, Optimizations in C++ compilers, *Comm ACM* **63** (2020), 41–49. doi:10.1145/3369754.
[113] M. Godbolt et al., Compiler explorer, 2012. https://godbolt.org.
[114] D. Goldberg, What every computer scientist should know about floating-point arithmetic, *ACM Comput Surv* **23** (1991), 5–48. doi:10.1145/103162.103163.
[115] Google, C++ Style Guide. https://google.github.io/styleguide/cppguide.html.
[116] B.E. Granger and F. Pérez, Jupyter: Thinking and storytelling with code and data, *Comp Sci Eng* **23** (2021), 7–14. doi:10.1109/MCSE.2021.3059263.
[117] R. Green and H. Ledgard, Coding guidelines: Finding the art in the science, *Commun ACM* **54** (2011), 57–63. doi:10.1145/2043174.2043191.
[118] B.A. Grüning, S. Lampa, M. Vaudel and D. Blankenberg, Software engineering for scientific big data analysis, *GigaScience* **8** (2019), 1–6. doi:10.1093/gigascience/giz054.
[119] gperftools (originally Google Performance Tools) — A collection of a high-performance multi-threaded malloc() implementation, plus some pretty nifty performance analysis tools. https://github.com/gperftools/gperftools.
[120] P. Hammant, Trunk Based Development, 2017. https://trunkbaseddevelopment.com.
[121] C.R. Harris et al., Array programming with NumPy, *Nature* **585** (2020), 357–362. doi:10.1038/s41586-020-2649-2.
[122] P. Hauer, Code Review Guidelines for Humans, 2018. https://phauer.com/2018/code-review-guidelines.
[123] The HDF Group, HDF5 — A data model, library, and file format for storing and managing data. https://portal.hdfgroup.org/display/HDF5/HDF5.

[124] The HDF Group, Other Tools. https://bit.ly/3nzEvb2.
[125] Heinz Maier-Leibnitz Zentrum (Garching, Germany). https://mlz-garching.de.
[126] T. Helvick, Design Patterns: Genius or Overengineered?, 2018. https://bit.ly/3bnGGI7.
[127] K. Henney (ed.), *97 Things Every Programmer Should Know: Collective Wisdom from the Experts*, O'Reilly, 2010.
[128] K. Henney, Comment Only What the Code Cannot Say, Chapter 17 in Ref. [127], 2010.
[129] M. Hevery, The Clean Code Talks — Unit Testing, 2008. https://youtu.be/wEhu57pih5w.
[130] M. Hevery, Guide: Writing Testable Code, 2008. http://misko.hevery.com/code-reviewers-guide.
[131] K. Hinsen, Technical debt in computational science, *Comp Sci Eng* **17** (2015), 103–107. doi:10.1109/MCSE.2015.113.
[132] K. Hinsen, A plea for stability in the SciPy ecosystem, 2017. https://bit.ly/3po4F14.
[133] K. Hinsen, Dealing with software collapse, *Comp Sci Eng* **21** (2019), 104–108. doi:10.1109/MCSE.2019.2900945.
[134] K. Hinsen, *Computation in Science. From Concepts to Practice*, 2nd edn, IOP Publishing, 2020.
[135] T. Hoff, Big List of 20 Common Bottlenecks, 2012. https://bit.ly/3tJOONm.
[136] B. Holland, 11 JavaScript frameworks for creating graphics, 2020. https://bit.ly/2XxRDDN.
[137] Homebrew — The Missing Package Manager for macOS (or Linux). https://brew.sh.
[138] J. Hughes, Lazy Memo-functions, in: *Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud, ed., Springer, 1975.
[139] Hugo — The world's fastest framework for building websites. https://gohugo.io.
[140] J.D. Hunter, Matplotlib: A 2D graphics environment, *Comp Sci Eng* **9** (2007), 90–95. doi:10.1109/MCSE.2007.55.
[141] D.G. Ince, L. Hatton and J. Graham-Cumming, The case for open computer programs, *Nature* **482** (2012), 485–488. doi:10.1038/nature10836.
[142] H. Ingo, Open Life: The Philosophy of Open Source, 2006. https://www.openlife.cc.
[143] Institut Laue-Langevin (Grenoble, France). https://www.ill.eu.
[144] ISIS Neutron and Muon Source (Didcot, UK). https://www.isis.stfc.ac.uk.
[145] ISO/IEC 2382:2015(en) Information technology — Vocabulary, 2015. https://bit.ly/3wNCY4A.
[146] ISO/IEC, JTC1/SC22/WG5, Working document N2113: Units of measure for numerical quantities, 2016. https://wg5-fortran.org/documents.html.
[147] Journal of Open Research Software. https://openresearchsoftware.metajnl.com.
[148] The Journal of Open Source Software. https://joss.theoj.org.
[149] Jekyll — Transform your plain text into static websites and blogs. https://jekyllrb.com.
[150] R.C. Jiménez et al., Four simple recommendations to encourage best practices in research software, *F1000Research* **6** (2017), 876. doi:10.12688/f1000research.11407.1.
[151] K. Jolly, *Hands-On Data Visualization with Bokeh: Interactive web plotting for Python using Bokeh*, Packt, 2018.
[152] D.M. Jones, The New C Standard (Identifiers). An Economic and Cultural Commentary, 2008. https://bit.ly/3aVkXa7.
[153] Journal of Applied Crystallography, Notes for authors. https://bit.ly/3vDXAvp.
[154] A.P. Kaestner, MuhRec — A new tomography reconstructor, *Nucl Instr Meth A* **651** (2011), 156–160. doi:10.1016/j.nima.2011.01.129.
[155] A.P Kaestner et al., Imaging suite — Software for neutron imaging. https://github.com/neutronimaging/imagingsuite. Documentation at https://neutronimaging.github.io.
[156] D.S. Katz et al., Introducing the International Council of RSE Associations. https://bit.ly/3aTYSJg.
[157] D.S. Katz, M. Gruenpeter and T. Honeyman, Taking a fresh look at FAIR for research software, *Patterns* **2** (2021), 1–3.
[158] J. Kiska, Static analysis of Python code, Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2021. https://bit.ly/3CgY0up.
[159] G. Klinger, gprof, Valgrind and gperftools — an evaluation of some tools for application level CPU profiling on Linux, 2015. https://bit.ly/3IohIqE.
[160] D.E. Knuth, Structured programming with go to statements, *ACM Comput Surv* **6** (1974), 261–301. doi:10.1145/356635.356640.
[161] S. Knuth and T. Hauser, Introduction to High-Performance and Parallel Computing [free online course]. https://bit.ly/3B3AEqm.
[162] M. Könnecke et al., The NeXus data format, *J Appl Cryst* **48** (2015), 301–305. doi:10.1107/S1600576714027575.
[163] M. Krzywinski and B. Wong, Points of view: Plotting symbols, *Nature Methods* **10** (2013), 451. doi:10.1038/nmeth.2490.
[164] G.M. Kurtzer, V. Sochat and M.W. Bauer, Singularity: Scientific containers for mobility of compute, *PLOS ONE* **12** (2017), e0177459. doi:10.1371/journal.pone.0177459.
[165] C. Ladas, *Scrumban — Essays on Kanban Systems for Lean Software Development*, Modus Cooperandi Press, 2008.
[166] A.-L. Lamprecht et al., Towards FAIR principles for research software, *Data Sci* **3** (2020), 37–59. doi:10.3233/DS-190026.
[167] League of advanced European Neutron Sources (LENS), Working groups. https://www.lens-initiative.org/working-groups.
[168] J.T. Leek and R.D. Peng, Reproducible research can still be wrong: Adopting a prevention approach, *Proc Nat Acad Sci USA* **112** (2015), 1645–1646. doi:10.1073/pnas.1421412111.
[169] K.J. Lieberherr and I.M. Holland, Assuring good style for object-oriented programs, *IEEE Software* **6** (1989), 38–48. doi:10.1109/52.35588.
[170] C. Lilienthal, *Sustainable Software Architecture. Analyze and Reduce Technical Debt*, 2nd edn, dpunkt.verlag, 2017.
[171] V. Lindberg, *Intellectual Property and Open Source*, O'Reilly, 2008.
[172] B. Linders, Dead Code Must Be Removed, 2017. https://www.infoq.com/news/2017/02/dead-code.

[173] Linux manual, assert(3). https://man7.org/linux/man-pages/man3/assert.3.html.
[174] Linux manual, errno(3). https://man7.org/linux/man-pages/man3/errno.3.html.
[175] S. Lyubinsky, Top 8 Python Testing Frameworks in 2021, 2020. https://bit.ly/3B1g4qU.
[176] MacPorts — An open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading [...] open-source software on the Mac operating system. https://www.macports.org.
[177] J. Majorek, 14 JavaScript Data Visualization Libraries in 2021. https://bit.ly/3pkHXIf.
[178] Mantid — Manipulation and Analysis Toolkit for Instrument Data. doi:10.5286/software/mantid.
[179] Mantid, Mantid Governance. https://bit.ly/3I2pkyD.
[180] Mantid, MVP Design. https://bit.ly/3JdRfNL.
[181] A. Markvardsen et al., EU SINE2020 WP 10: Report on guidelines and standards for data treatment software, Technical Report, RAL-TR-2019-008, Science and Technology Facilities Council, 2019. http://purl.org/net/epubs/work/43005596.
[182] R.C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Pearson, 2003.
[183] R.C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Pearson, 2009.
[184] R.C. Martin, The Boy Scout Rule, Chapter 8 in Ref. [127], 2010.
[185] R.C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Pearson, 2018.
[186] A. Martins, Git(Hub) Flow, Trunk Based Development, and Code reviews, 2021. https://bit.ly/3pnYE5m.
[187] S. McConnell, *Code Complete*, 2nd edn, Microsoft Press, 2004.
[188] McStas — A neutron ray-trace simulation package. http://www.mcstas.org.
[189] S. Mehta, Unit Testing Frameworks in Python, 2021. https://bit.ly/3n94eWw.
[190] B. Meyer, *Object-Oriented Software Construction*, 2nd edn, Prentice Hall, 1997.
[191] B. Meyer, *Agile! The Good, the Hype and the Ugly*, Springer, 2014.
[192] D. Michie, "Memo" functions and machine learning, *Nature* **218** (1968), 19–22. doi:10.1038/218019a0.
[193] Microsoft, C++ team blog, C11 and C17 Standard Support Arriving in MSVC, 2020. https://bit.ly/3vwM0C9.
[194] Microsoft, .NET Documentation, General Naming Conventions, 2021. https://bit.ly/3E7rLyh.
[195] Microsoft, Windows App Development, Windows Package Manager, 2021. https://bit.ly/3CqlfC2.
[196] C. Mimbs Nyce, The Winter Getaway That Turned the Software World Upside Down. How a group of programming rebels started a global movement, *The Atlantic* (2017). https://bit.ly/30uFFw5.
[197] R.M.L.M. Moreira, A.C. Paiva, M. Nabuco and A. Memon, Pattern-based GUI testing: Bridging the gap between design and quality assurance, *Softw Test Verif Reliab* **27** (2017), e1629. doi:10.1002/stvr.1629.
[198] A. Morin, J. Urban and P. Sliz, A quick guide to software licensing for the scientist-programmer, *PLoS Comput Biol* **8** (2012), e1002598. doi:10.1371/journal.pcbi.1002598.
[199] NumPy — The fundamental package for scientific computing with Python. https://numpy.org.
[200] NumPy User Guide. How to extend NumPy, 2021. https://numpy.org/doc/stable/user/c-info.how-to-extend.html.
[201] R. Nystrom, *Game Programming Patterns*, genever benning, 2014.
[202] Open Broadcaster Software (OBS) — Free and open source software for video recording and live streaming. https://obsproject.com/.
[203] Open Reflectometry Standards Organisation (ORSO), File Formats Working Group. https://www.reflectometry.org/working_groups/file_formats.
[204] Open Source Initiative, Licenses & Standards. https://opensource.org/licenses.
[205] Open Source Initative, The 3-Clause BSD License. https://opensource.org/licenses/BSD-3-Clause.
[206] T. O'Reilly, Lessons from open-source software development, *Commun ACM* **42** (1999), 33–37. doi:10.1145/299157.299164.
[207] J. Owens et al., Survey of general-purpose computation on graphics hardware, *Comput Graph Forum* **26** (2007), 80–113. doi:10.1111/j.1467-8659.2007.01012.x.
[208] PaN-learning — The Photon and Neutron E-learning platform. https://pan-learning.org.
[209] PaN-learning [208], Creating a Video Mini-Lecture. https://bit.ly/3HX2EzT.
[210] PaNOSC — The Photon and Neutron Open Science Cloud. https://www.panosc.eu.
[211] T. Pasquier et al., If these data could talk, *Sci Data* **4** (2017), 170114. doi:10.1038/sdata.2017.114.
[212] Paul Scherrer Institute (Villigen, Switzerland). https://www.psi.ch/.
[213] L. Pecora, SciPy Cookbook. C Extensions to NumPy and Python, 2006. https://bit.ly/2Z8hSku.
[214] R.D. Peng, Reproducible research in computational science, *Science* **334** (2011), 1226–1227. doi:10.1126/science.1213847.
[215] Y. Perez-Riverol et al., Ten simple rules for taking advantage of git and GitHub, *PLOS Comput Biol* **12** (2016), e1004947. doi:10.1371/journal.pcbi.1004947.
[216] J.M. Perkel, Programming: Pick up Python, *Nature* **518** (2015), 125–126. doi:10.1038/518125a.
[217] J.M. Perkel, Why Jupyter is data scientists' computational notebook of choice, *Nature* **563** (2018), 145–146. doi:10.1038/d41586-018-07196-1.
[218] J.M. Perkel, Julia: Come for the syntax, stay for the speed, *Nature* **572** (2019), 141–142. doi:10.1038/d41586-019-02310-3.
[219] J.M. Perkel, Why scientists are turning to rust, *Nature* **588** (2020), 185–186. doi:10.1038/d41586-020-03382-2.
[220] J.M. Perkel, Challenge to scientists: Does your ten-year-old code still run?, *Nature* **584** (2020), 656–658. doi:10.1038/d41586-020-02462-7.

[221] PEX — A library for generating .pex (Python EXecutable) files which are executable Python environments. https://bit.ly/3Esw7jk.
[222] R. Pichler, The Scrum product owner role on one page, 2010. https://bit.ly/3wP15jl.
[223] R. Pike, Go at Google: Language Design in the Service of Software Engineering, 2012. https://bit.ly/3ftMZff.
[224] S. Pittet, Continuous integration vs. continuous delivery vs. continuous deployment. https://bit.ly/30GNKNT.
[225] Poetry — Python packaging and dependency management made easy. https://python-poetry.org.
[226] K. Poppe, R. Cools and B. Vandewoestyne, Error handling in Fortran 2003. https://bit.ly/3C5PJt8.
[227] M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*, Addison-Wesley, 2003.
[228] G. Pospelov et al., BornAgain: Software for simulating and fitting grazing-incidence small-angle scattering, *J Appl Cryst* **53** (2020), 262–272. doi:10.1107/S1600576719016789.
[229] E. Post, Real Programmers Don't Use PASCAL, 1982. https://www.ee.ryerson.ca/~elf/hack/realmen.html.
[230] M. Potel, MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java, 1996. https://bit.ly/3Etw3ja.
[231] T. Preston-Werner, Semantic Versioning 2.0.0. https://semver.org.
[232] A. Prlić and J.B. Procter, Ten simple rules for the open development of scientific software, *PLOS Comput Biol* **8** (2012), e1002802.
[233] Project Management Institute, PMI Lexicon of Project Management Terms [requires registration]. https://bit.ly/3pDSTjM.
[234] M. Pusz, A C++ Approach to Physical Units. Proposal P1935R2, ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++, 2020. https://bit.ly/30GOh2l.
[235] PyInstaller — Freezes Python applications into stand-alone executables. https://github.com/pyinstaller/pyinstaller.
[236] Pylint — A tool that checks for errors in Python code, tries to enforce a coding standard and looks for code smells. https://www.pylint.org.
[237] Pytest framework documentation. https://docs.pytest.org/.
[238] Python Developer's Guide, How to Become a Core Developer, Sign a Contributor Agreement, 2011–2021. https://bit.ly/3kX3G60.
[239] Python Package Index. https://pypi.org.
[240] Python Packaging User Guide, An Overview of Packaging for Python. https://packaging.python.org/overview.
[241] Python Packaging User Guide, Packaging and distributing projects. https://bit.ly/3poWdzt.
[242] Python Wiki, Advanced Books. https://wiki.python.org/moin/AdvancedBooks.
[243] Qt — The cross-platform software development framework. https://www.qt.io.
[244] Qt for Python. https://doc.qt.io/qtforpython.
[245] Qt Test Overview. https://doc.qt.io/qt-5/qtest-overview.html.
[246] E.S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, 1999.
[247] E.S. Raymond, Project structures and ownership, in: *Homesteading the Noosphere* [Revised part of the book [246]], 2000, https://bit.ly/30xHmIY.
[248] E.S. Raymond, *The Art of Unix Programming*, Addison-Wesley, 2003. Also published online at https://bit.ly/3CShqGQ.
[249] K. Reitz and T. Schlusser, *The Hitchhiker's Guide to Python. Best Practices for Development*, O'Reilly, 2016.
[250] D. Riehle, How Project vs. Product Confuses Open Source Terminology, 2018. https://bit.ly/3EE9I2f.
[251] S. Robinson, Python Circular Imports, 2017. https://bit.ly/3CsfRhP.
[252] Rust–Qt — Qt bindings for Rust language. https://github.com/rust-qt.
[253] C. Sadowski et al., Modern code review: A case study at Google, in: *ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018. doi:10.1145/3183519.3183525.
[254] SasView — Small angle scattering analysis. https://www.sasview.org.
[255] SasView, "about" https://www.sasview.org/about.
[256] M.C. Schabel and S. Watanabe, Boost.Units, 2003–2010. https://bit.ly/3jmKUUQ.
[257] H. Schreiner et al., An Introduction to Modern CMake. https://cliutils.gitlab.io/modern-cmake.
[258] K. Schwaber, *Agile Project Management with Scrum*, Microsoft Press, 2004.
[259] K. Schwaber and J. Sutherland, The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game, 2020. https://scrumguides.org.
[260] S. Schwartz, Style Guides and Rules, Chapter 8 in Ref. [323], 2020.
[261] scipp — Multi-dimensional data arrays with labeled dimensions. https://scipp.github.io.
[262] SciPy — A Python-based ecosystem of open-source software for mathematics, science, and engineering. https://www.scipy.org.
[263] Scoop — A command-line installer for Windows. https://scoop.sh.
[264] C. Scott, Professional CMake: A Practical Guide. https://crascit.com/professional-cmake.
[265] M. Sebor, Understanding GCC warnings, 2019. https://red.ht/3neuG15.
[266] M. Seemann, Put cyclomatic complexity to good use, 2019. https://bit.ly/3jpTORA.
[267] SINE2020 — Science & Innovation with Neutrons in Europe in 2020, Workpackage 10, Data Treatment Software. https://bit.ly/3aYxaLi.
[268] A.B. Singer, *Practical C++ Design: From Programming to Architecture*, Apress, 2017.
[269] Slant, What are the best Mac package managers? https://www.slant.co/topics/511.
[270] Slant, What are the best Windows package managers? https://www.slant.co/topics/1843.
[271] A. Sinhal, MVC, MVP and MVVM Design Pattern, 2017. https://bit.ly/33wtTD6.
[272] Snap — The app store for Linux. https://snapcraft.io.

[273] Software Sustainability Institute. https://www.software.ac.uk.
[274] P. Sommerlad, Only the Code Tells the Truth, Chapter 62 in Ref. [127], 2010.
[275] D. Spinellis, Put Everything Under Version Control, Chapter 68 in Ref. [127], 2010.
[276] StackExchange Software Engineering, What to call tests that check that output has not changed? https://softwareengineering.stackexchange.com/questions/432723.
[277] StackExchange Super User, Difference between a stand-alone executable file, and an installed executable? https://superuser.com/a/685047/269343.
[278] Stackoverflow. https://stackoverflow.com.
[279] Stackoverflow, How deep are your unit tests?, answer by K. Beck, 2008. https://stackoverflow.com/a/153565/1017348.
[280] I. Stančin and A. Jović, An overview and comparison of free Python libraries for data mining and big data analysis, in: *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, K. Skala, ed., 2019.
[281] A. Stellman and J. Greene, *Applied Software Project Management*, O'Reilly, 2005.
[282] A.G. Stephenson et al., Mars Climate Orbiter Mishap Investigation Board. Phase I Report, 1999. https://go.nasa.gov/3m0jshc.
[283] V. Stodden, The legal framework for reproducible scientific research. Licensing and copyright, *Comput Sci Eng* **11** (2009), 35–40. doi:10.1109/MCSE.2009.19.
[284] T. Storer, Bridging the chasm: A survey of software engineering practices in scientific programming, *ACM Comput Surv* **50**(47) (2017), 1–32. doi:10.1145/3084225.
[285] B. Stroustrup, Bjarne Stroustrup's FAQ, 2021. https://www.stroustrup.com/bs_faq.html.
[286] B. Stroustrup and H. Sutter, C++ Core Guidelines, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.
[287] S. Sturluson, Catch2 vs Google Test, 2018. https://snorristurluson.github.io/Catch2.
[288] G. Suryanrayana, G. Samarthyam and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*, Elsevier, 2015.
[289] R. Swidzinski, *Modern CMake for C++*, Packt, 2022.
[290] A. Tarlinder, *Developer Testing: Building Quality into Software*, Addison-Wesley, 2016.
[291] therecipe/qt — Allows you to write Qt applications entirely in Go, JavaScript/TypeScript, Dart/Flutter, Haxe and Swift. https://github.com/therecipe/qt.
[292] G.K. Thiruvathukal, K. Läufer and B. Gonzalez, Unit testing considered useful, *Comp Sci Eng* **8** (2006), 76–87. doi:10.1109/MCSE.2006.124.
[293] D. Thomas and A. Hunt, *The Pragmatic Programmer: Journey to Mastery*, 20th anniversary edn [= 2nd edn], Addison-Wesley, 2019.
[294] P. Tonella, Concept analysis for module restructuring, *IEEE T Software Eng* **27** (2001), 351–363. doi:10.1109/32.917524.
[295] A. Tornhill, The Signs of Trouble: On Patterns, Humbleness and Lisp, 2012. https://bit.ly/3CwGdzp.
[296] A. Tornhill, *Patterns in C. Patterns, Idioms and Design Principles*, Leanpub, 2015.
[297] L. Torvalds, Linux kernel coding style. https://bit.ly/3lYu7cs.
[298] L. Torvalds, Git. Source code control the way it was meant to be! https://www.youtube.com/watch?v=4XpnKHJAok8.
[299] A. Trenk and B. Dillon, Test Doubles, Chapter 13 in Ref. [323], 2020.
[300] E.R. Tufte, *The Visual Display of Quantitative Information*, 2nd edn, Graphics Press, 2001.
[301] J. Turner and C. Best, Practices: A Forkable Coding Standards Document. https://bit.ly/3nisfLb.
[302] UK Research and Innovation. https://www.ukri.org.
[303] Units — A run-time C++ library for working with units of measurement and conversions between them and with string representations of units and measurements. https://github.com/LLNL/units.
[304] NeXus, User Manual and Reference Documentation. https://bit.ly/3GDR8JJ.
[305] F. van Laenen, Automate Your Coding Standard, Chapter 4 in Ref. [127], 2010.
[306] G. van Rossum, B. Warsaw and N. Coghlan, PEP 8 — Style Guide for Python Code, 2001. https://www.python.org/dev/peps/pep-0008.
[307] S. Velikan, Open Source Candies. The Free-for-Open-Source Services List. https://github.com/velikanov/opensource-candies.
[308] Virtanen et al., SciPy 1.0: Fundamental algorithms for scientific computing in Python, *Nat Methods* **17** (2020), 261–272. doi:10.1038/s41592-019-0686-2.
[309] P. Vuollet, Software Architecture Document? Do You Need One? https://bit.ly/3rcNkIL.
[310] N. Wadeson and M. Basham, Savu: A Python-based, MPI Framework for Simultaneous Processing of Multiple, N-dimensional, Large Tomography Datasets, arXiv preprint, 2016. arXiv:1610.08015.
[311] A.H. Watson and T.J. McCabe, Structured testing: A testing methodology using the cyclomatic complexity metric, in: *NIST Special Publication*, D.R. Wallace, ed., National Institute of Standards and Technology, 1996.
[312] P. Wayner, CI/CD as a service: 10 tools for continuous integration and delivery in the cloud, 2019. https://www.infoworld.com/article/3341320.
[313] Wikipedia, Benevolent dictator for life. https://en.wikipedia.org/wiki/Benevolent_dictator_for_life.
[314] Wikipedia, American and British English spelling differences. https://bit.ly/3nmtMkd.
[315] Wikipedia, Basic Linear Algebra Subprograms. https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms.
[316] M. Wilkinson et al., The FAIR guiding principles for scientific data management and stewardship, *Sci Data* **3** (2016), 160018. doi:10.1038/sdata.2016.18.

[317] P.K. Willendrup and K. Lefmann, McStas (i): Introduction, use, and basic principles for ray-tracing simulations, *J Neutron Res* **22** (2020), 1–16. doi:10.3233/JNR-190108.
[318] P.K. Willendrup and K. Lefmann, McStas (ii): An overview of components, their use, and advice for user contributions, *J Neutron Res* **23** (2021), 7–27. doi:10.3233/JNR-200186.
[319] G. Wilson, Why We Don't Teach Testing (Even Though We'd Like To), 2014. https://bit.ly/3B4e4Ox.
[320] G. Wilson et al., Best practices for scientific computing, *PLOS Biol* **12** (2014), e1001745. doi:10.1371/journal.pbio.1001745.
[321] T. Winters, What Is Software Engineering? Chapter 1 in Ref. [323], 2020.
[322] T. Winters, Dependency Management, Chapter 21 in Ref. [323], 2020.
[323] T. Winters, T. Manshreck and H. Wright, eds., *Software Engineering at Google. Lessons Learned from Programming over Time*, O'Reilly, 2020.
[324] J. Wuttke, Laplace–Fourier transform of the stretched exponential function: Analytic error bounds, double exponential transform, and open-source implementation libkww, *Algorithms* **5** (2012), 604–628. doi:10.3390/a5040604.
[325] J. Wuttke, libQCR — Qt Capture & Replay. https://jugit.fz-juelich.de/mlz/libqcr.
[326] YAML Ain't Markup Language. https://yaml.org.
[327] YAPF — Yet another Python formatter. https://github.com/google/yapf.
[328] S. Yegulalp, 6 stellar libraries for profiling Python code, 2021. https://www.infoworld.com/article/3600993.
[329] S. Yegulalp, Julia vs. Python: Which is best for data science? 2020. https://bit.ly/2Zjrlps.
[330] Zenodo. https://zenodo.org.
[331] Zero Install — A decentralised cross-platform software installation system. https://0install.net.