Establishing trust in automated reasoning

Konrad Hinsen

Establishing trust in automated reasoning

Konrad Hinsen

konrad.hinsen@cnrs.fr

Centre de Biophysique Moléculaire (UPR4301 CNRS)
Rue Charles Sadron, 45071 Orléans Cédex 2, France

Synchrotron SOLEIL, Division Expériences
B.P. 48, 91192 Gif sur Yvette, France

Abstract

Since its beginnings in the 1940s, automated reasoning by computers has become a tool of ever growing importance in scientific research. So far, the rules underlying automated reasoning have mainly been formulated by humans, in the form of program source code. Rules derived from large amounts of data, via machine learning techniques, are a complementary approach currently under intense development. The question of why we should trust these systems, and the results obtained with their help, has been discussed by philosophers of science but has so far received little attention by practitioners. The present work focuses on independent reviewing, an important source of trust in science, and identifies the characteristics of automated reasoning systems that affect their reviewability. It also discusses possible steps towards increasing reviewability and trustworthiness via a combination of technical and social measures.

1 Introduction

Like all social processes, scientific research builds on trust. In order to increase humanity's knowledge and understanding, scientists need to trust their colleagues, their institutions, their tools, and the scientific record. Moreover, science plays an increasingly important role in industry and public policy. Decision makers in these spheres must therefore be able to judge

which of the scientific findings that matter for them are actually trustworthy.

In addition to the trust-forming mechanisms present in all social relationships, the scientific method is built in particular on *transparency* and *independent critical inspection*, which serve to remove the inevitable mistakes and biases in individual contributions as they enter the scientific record. Ever since the beginnings of organized science in the 17th century, researchers are expected to put all facts supporting their conclusions on the table, and allow their peers to inspect them for accuracy, pertinence, completeness, and bias. Since the 1950s, critical inspection has been implemented as a formal process called *peer review*, which is still widely regarded as a key criterion for trustworthy results.

Over the last two decades, an unexpectedly large number of peer-reviewed findings across many scientific disciplines have been found to be irreproducible upon closer inspection. This so-called "reproducibility crisis" has shown that our practices for performing, publishing, reviewing, and interpreting scientific studies are no longer adequate in today's scientific research landscape, whose social, technological, and economic contexts have changed dramatically. Updating these processes is a major aspect of the nascent Open Science movement.

The topic of this article is a particularly important recent change in research practices: the increasing use of automated reasoning. Computers and software have led to the development of completely new techniques for scientific investigation, and permitted existing ones to be applied at larger scales and by a much larger number of researchers. In the quantitative sciences, almost all of today's research critically relies on computational techniques, even when they are not the primary tool for investigation. Simulation, data analysis, and statistical inference have found their place in almost every researcher's toolbox. Machine learning techniques, currently under intense development, may well become equally ubiquitous in the near future.

From the point of view of transparency and critical inspection, these new tools are highly problematic. Ideally, each piece of software should perform a well-defined computation that is documented in sufficient detail for its users and verifiable by independent reviewers. Furthermore, users of software should receive adequate training to ensure that they understand the software's operation and in particular its limitations. Today's reality is very different. A large number of cases cited in discussions of the reproducibility crisis involves faulty software or inappropriate use of software. A particularly frequent issue is the inappropriate use of statistical inference techniques. They are available at the click of a button to a large number of researchers, many of which do not even know what they would need to learn in order to use these techniques correctly. Beyond reproducibility, the documented cases of faulty automated reasoning [e.g. Merali 2010] are probably just the tip of

the iceberg, and pessimistic but not unrealistic estimates suggest that most computational results in science are to some degree wrong [Soergel 2014].

The Open Science movement has made a first step towards dealing with automated reasoning in insisting on the necessity to publish scientific software, and ideally making the full development process transparent by the adoption of Open Source practices. While this level of transparency is a necessary condition for critical inspection, it is not sufficient. Almost no scientific software is subjected to in-depth independent review today. In fact, we do not even have established processes for performing such reviews. Moreover, as I will show, much of today's scientific software is written in a way that makes independent critical inspection particularly challenging if not impossible. If we want scientific software to become trustworthy, we therefore have to develop reviewing practices in parallel with software architectures that make reviewing actually feasible in practice. And where reviewing is not possible, we must acknowledge the experimental nature of automated reasoning processes and make sure that everyone looking at their results is aware of their uncertain reliability.

As for all research tools, it is not only the software itself that requires critical inspection, but also the way the software is used in a specific research project. Improper use of software, or inappropriateness of the methods implemented by the software, is as much a source of mistakes as defects in the software itself. However, the distinction between a defect and inappropriate use is not as obvious as it may seem. A clear distinction would require a well-defined interface between software and users, much like a written contract. If the software's behavior deviates from this contract, it's a defect. If the user's needs deviate from the contract, it's inappropriate use. But such detailed contracts, called *specifications* in the context of software, rarely exist. Even outside of science, the cost of writing, verifying, and maintaining specifications limits their use to particularly critical applications. This means that reviewing the use of scientific software requires particular attention to potential mismatches between the software's behavior and its users' expectations, in particular concerning edge cases tacit assumptions made by the software developers. They are necessarily expressed somewhere in the software's source code, but users are often not aware of them.

The scientific requirement of *independent* reviewing is related to another aspect of automated reasoning that I will address, in particular in my proposals for improving our current practices: the preservation of epistemic diversity. As Leonelli has pointed out [Leonelli 2022], the Open Science movement has so far largely neglected this point. Epistemic diversity is about different perspectives and research methodologies coexisting, enriching and critiquing each other. Automation, be it in industry or in research, tends to reduce diversity by encouraging standardization that enables economies

of scale. In the Open Science movement, this tendency is implicit in the quest for reusability, one of the four FAIR principles [Wilkinson et al. 2016; Barker et al. 2022]. Reusing someone else's code or data requires adopting the authors' methodologies, and to some degree their general perspective on the phenomenon under study. In the extreme case of a single software package being used by everyone in a research community, there is nobody left who could provide critical feedback at all.

This article has two main parts. In the first part (section 2), I look at the factors that make automated reasoning more or less reviewable. It is a critical examination of the state of the art in scientific software and its application, which should help scientists to get a better grasp of how reliable automated reasoning can be expected to be. In the second part (section 3), I consider how the reviewability of automated reasoning can be improved, both through better reviewing processes and by restructuring software for better reviewability.

2 Reviewability of automated reasoning systems

Automated reasoning can play different roles in scientific research, with different reliability requirements.[1] The numerical preprocessing of observational data before scientific analysis, nowadays often integrated into scientific instruments, is an example where high reliability is required, because its outputs are used without any further verification. On the other hand, protein structure prediction by AlphaFold [Jumper et al. 2021] is known to be unreliable, but it is nevertheless very useful if coupled with experimental validation of its predictions [Nielsen 2023]. Traditional computer simulation is often used similarly in biology as a hypothesis generator whose outputs require subsequent validation, whereas in engineering, simulations of mechanical systems are routinely performed to support critical decisions, thus requiring high reliability.

What these examples illustrate is that tools, processes, and results in science do not necessarily have to be perfectly reliable. Higher-level validation processes act much like error-correction protocols in engineering. The coherence of multiple approaches to a question, coming from different perspectives, is another higher-level source of reliability, indicating robustness. This again illustrates the importance of epistemic diversity that I have mentioned in the introduction. What matters, however, is a clear understanding of the reliability of individual scientific contributions, which in turn requires a clear understanding of the reliability of the tools and processes on which those contributions are based.

[1] This is of course true for software in general, see e.g. the discussion in [Shaw 2022:22].

In this section, I discuss five characteristics (summarized in Fig. 1) of automated reasoning systems that influence how their reliability can be assessed by independent critical inspection, which in the following I will call *review* for brevity. This use of *review*, inspired by the tradition of scientific peer review, should not be confused with the software engineering technique of *code review*, which is a quality control step performed *internally* by a development team. Also for brevity, I will use the term *software* instead of "automated reasoning system", extending its usual meaning to include trained neural networks and other models obtained via machine learning techniques.

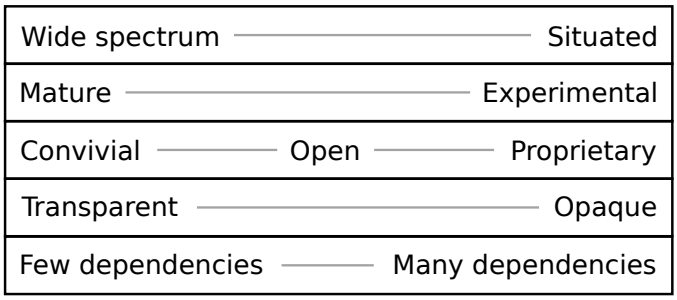

Figure 1: The five dimensions of scientific software that influence its reviewability.

2.1 Wide-spectrum vs. situated software

Wide-spectrum software provides fundamental computing functionality to a large number of users. In order to serve a large user base, it addresses a wide range of application scenarios, each of which requiring only a part of the software's functionality. Word processors are a well-known example: a package like LibreOffice can be used to write a simple letter, but also a complex book. LibreOffice has huge menus filled with various functions, of which most users only know the handful that matters to them. General-purpose large language models are another example of wide-spectrum software.

Situated software (a term introduced by Shirky [Shirky 2004]) is software written for a specific use case or a specific user group. It addresses a specific need very well, but is not transferable to other application scenarios. Spreadsheets are usually situated, as are games, and many shell scripts.

A useful numerical proxy for estimating a software package's location on this scale is the ratio of the number of users to the number of developers, although there are exceptions. Games, for example, are situated software with few developers but many users.

In scientific computing, the wide-spectrum end of the scale is well illustrated by mathematical libraries such as BLAS or visualization libraries such as

matplotlib, which provide a large collection of functions from which application developers pick what they need. At the situated end, we have the code snippets and scripts that generate the plots shown in a paper, as well as computational notebooks and computational workflows. In between these extremes, we have in particular domain libraries and tools, which play a very important role in computational science, i.e. in studies where computational techniques are the principal means of investigation. Many ongoing discussions of scientific software, in particular concerning its sustainability [Hettrick 2016], concentrate on these domain libraries and tools, but are not always explicit about this focus.

Reviewing wide-spectrum software represents a major effort, because of its size and functional diversity. Moreover, since wide-spectrum software projects tend to be long-lived, with the software evolving to adapt to new use cases and new computing platforms, its critical examination must be an ongoing process as well. On the other hand, this effort can be considered a good investment, because of the large user base such software has.

Situated software is smaller and simpler, which makes it easier to understand and thus to review. However, its evaluation can only be done in the specific context for which the software was written. This suggests integrating it into the existing scientific peer reviewing process, along with papers and other artifacts that result from a research project.

It is the intermediate forms of software that are most difficult to review. Domain tools and libraries are too large and complex to be evaluated in a single session by a single person, as is expected in peer review as it is practiced today by journals. However, they don't have a large enough user base to justify costly external audits, except in contexts such as high-performance computing where the importance of the application and the high cost of the invested resources also justify more careful verification processes.

2.2 Mature vs. experimental software

Mature software is developed and maintained with the goal of providing a reliable tool. Signs of maturity in software are its age, a clear definition of its purpose, respect of standards, respect of software engineering practices, detailed documentation, and a low frequency of compatibility-breaking changes. The Linux kernel and the text editor Emacs are examples of very mature software.

Experimental software is developed and maintained to test new ideas, be they technical (software architecture etc.) or related to the application domain. Experimental software evolves at a much faster pace than mature software, and documentation is rarely up to date or complete. Users therefore have to stay in touch with the developer community, both to be informed about

changes and to have an interlocutor in case of unexpected behavior.

Infrastructure software, i.e. packages that much other software depends on, should by definition be mature, and much of it is. This applies both to general-purpose infrastructure, such as the Linux kernel or the GNU Compiler Collection, and to scientific infrastructure, such as BLAS or HDF5. A grey zone is occupied by prototypes for future infrastructure software, such as the Julia programming language, which typically doesn't advertise its experimental nature and is easily taken to be mature by inexperienced users. There is also software that clearly positions itself as infrastructure but lacks the required maturity. Such software is often a cause of computational irreproducibility. The libraries of the scientific Python ecosystem are an example, suffering from frequent changes that break backward compatibility. With most users of these tools being unaware of these issues, they often find out too late that some of their critical dependencies are not as mature as they seemed to be.

Scientific domain libraries and tools tend to be in the middle of the spectrum, or try to cover a large part of the spectrum. There is an inevitable tension between providing reliable code for others to build on and implementing state-of-the-art computational techniques. Often the targeted user community for these two goals is the same. Pursuing both goals in the same software project can then be the least-effort approach, but it also makes the reliability of the software difficult to asses both by users and by outside reviewers.

Experimental software is, by its very nature, very difficult to review independently, unless it is small. This is not very different in principle from evaluating experiments that use prototypes for scientific instrumentation. The main difference in practice is the widespread use of experimental software by unsuspecting scientists who believe it to be mature, whereas users of instrument prototypes are usually well aware of the experimental status of their equipment.

2.3 Convivial vs. proprietary software

Convivial software [Kell 2020], named in reference to Ivan Illich's book "Tools for conviviality" [Illich 1973], is software that aims at augmenting its users' agency over their computation. Malleable software is a very similar concept, as is re-editable software, a term introduced by Donald Knuth in an interview in opposition to reusable, i.e. off-the-shelf, software [Hinsen 2018a]. In contrast, proprietary software offers users fixed functionality and therefore limited agency. At first sight this looks like users should always prefer convivial software, but agency comes at a price: users have to invest more learning effort and assume responsibility for their modifications. Just like most people prefer to choose from a limited range of industrially-made

refrigerators, rather than build their own precisely to their needs, most computer users are happy to use ready-made e-mail software rather then writing their own.

In the academic literature on software enginering, convivial software is discussed with the focus on its developers, most commonly referred to as *end user programmers* [Nardi 1993; Ko et al. 2011]. Shaw recently proposed the less pejorative term *vernacular developers* [Shaw 2022]. The subfield of end user software engineering aims at providing vernacular developers with methods and tools to improve the quality of their software, recognizing that the methods and tools designed for software professionals are usually not adapted to their needs.

The risk of proprietary technology, which Illich has described in detail, is that widespread adoption makes society as a whole dependent on a small number of people and organizations who control the technology. This is exactly what has happened with computing technology for the general public. You may not want to let tech corporations spy on you via your smartphone, but the wide adoption of these devices means that you are excluded from more and more areas of life if you decide not to use one. Some research communities have fallen into this trap as well, by adopting proprietary tools such as MATLAB as a foundation for their computational tools and models.

In between convivial and proprietary software, we have Free, Libre, and Open Source software (FLOSS). Historically, the Free Software movement was born in a universe of convivial technology. The few computer users in academia in the 1980s typically also had programming skills, and most of the software they produced and used was placed in the public domain. The arrival of proprietary software in their lives, exemplified by the frequently cited proprietary printer driver at MIT [2002], pushed them towards formalizing the concept of Free Software in terms of copyright and licensing, as they saw legal constraints as the main obstacle to preserving conviviality.

With the enormous complexification of software over the following decades, a license was no longer sufficient to keep software convivial in practice. The right to adapt software to your needs is of limited value if the effort to do so is prohibitive. Software complexity has led to a creeping loss of user agency, to the point that even building and installing Open Source software from its source code is often no longer accessible to non-experts, making them dependent not only on the development communities, but also on packaging experts. An experience report on building the popular machine learning library PyTorch from source code nicely illustrates this point [Courtès 2021]. Conviviality has become a marginal subject in the FLOSS movement, with the Free Software subcommunity pretending that it remains ensured by copyleft licenses and much of the Open Source subcommunity not considering it important. It survives mainly in communities that have their

roots in technology of the 1980s, such as programming systems inheriting from Smalltalk (e.g. Squeak and Pharo), or the programmable text editor GNU Emacs.

In scientific computing, there is a lot of diversity on this scale. Fully proprietary software is common, but also variants that do allow users to look at the source code, but don't allow them to compile it, or don't allow the publication of reviews. In computational chemistry, the widely used Gaussian software is an example for such legal constraints [Hocquet and Wieber 2017]. FLOSS has been rapidly gaining in popularity, and receives strong support from the Open Science movement. Somewhat surprisingly, the move beyond FLOSS to convivial software is hardly ever envisaged, in spite of it being aligned with the traditional values of scientific research: before the arrival of computers, theories and models have always been convivial technologies.

Concerning reviewing, the convivial-to-open part of the scale is similar to the situated-to-wide-spectrum scale: convivial software is easier to understand and therefore easier to review, but each specific adaptation of convivial software requires its own review, whereas open but not convivial software makes reviewing a better investment of effort. Fully proprietary software is very hard to review, because only its observed behavior and its documentation are available for critical inspection.

2.4 Transparent vs. opaque software

Transparent software is software whose behavior is readily observable. In a word processor, or a graphics editor, every user action produces an immediately visible result. In contrast, opaque software operates behind the scenes and produces output whose interpretation and correctness are not obvious, nor easily related to the inputs. Large language models are an extreme example.

Strictly speaking, transparency is not a characteristic of a piece of software, but of a computational task. A single piece of software may contain both transparent and opaque functionality. Taking the word processor as an example, inserting a character is highly transparent, whereas changing the page layout is more opaque, creating the possibility of subtle bugs whose impact is not readily observable. I use the term "opaque software" as a shorthand for "software implementing opaque operations".

Most scientific software is closer to the opaque end of the spectrum. Even highly interactive software, for example in data analysis, performs non-obvious computations, yielding output that an experienced user can perhaps judge for plausibility, but not for correctness. As a rough guideline, the more scientific models or observational data have been integrated into a piece of software, the more opaque the behavior of the software is to its users. Since

these are also the ingredients that make a piece of software scientific, it is not surprising that opacity is the norm rather than the exception.

It is much easier to develop trust in transparent than in opaque software. Reviewing transparent software is therefore easier, but also less important. When most users can understand and judge the results produced by a piece of software, even a very weak trustworthiness indicator such as popularity becomes sufficient.[2]

The more opaque a computation is, the more important its documentation becomes. Inadequately documented opaque software is inherently not trustworthy, because users don't know what the software does exactly, nor what its limitations are. This is currently a much discussed issue with machine learning models, but it is not sufficiently recognized that traditional computer software can be just as opaque from a user's point of view, if source code is the only available documentation of its behavior.

Opacity is an aspect of automated reasoning that has been treated extensively in the philosophy of science. Durán and Formanek [Durán and Formanek 2018] discuss epistemic opacity (which is not exactly the same as my pragmatic definition of opacity in this section) in the context of trust in the results of computer simulations, but much of their discussion equally applies to other uses of scientific software. They focus in particular on *essential* epistemic opacity, which is the degree of ignorance about an automated reasoning process that is due to the gap between the complexity of computer hardware and software and the limited cognitive capacities of a scientist. As an alternative source of trust, they propose *computational reliabilism*, which is trust derived from the experience that a computational procedure has produced mostly good results in a large number of applications. However, the accumulation of a sufficiently large body of validated applications is possible in practice only for mature wide-spectrum software.

2.5 Size of the minimal execution environment

Each piece of software requires an execution environment, consisting of a computer and other pieces of software. The importance of this execution environment is not sufficiently appreciated by most researchers today, who tend to consider it a technical detail. However, it is the execution environment that defines what a piece of software actually does. The meaning of a Python script is defined by the Python interpreter. The Python interpreter is itself a piece of software written in the C language, and therefore the meaning of its source code is defined by the C compiler and by the processor which

[2] A famous quote in software engineering, often referred to as "Linus' law", states that "given enough eyeballs, all bugs are shallow". However, this can only work if the many eyeballs are sufficiently trained to spot problems, meaning that "mere" users of opaque software don't qualify.

ultimately executes the binary code produced by the C compiler. As an illustration for the importance of the execution environment, it is an easy exercise to write a Python script that produces different results when run with version 2 or version 3 of the Python interpreter, exploiting the different semantics of integer division between the two versions.

In addition to this semantic importance of execution environments, reviewability implies a pragmatic one: reviewers of software or its results need access to an adequate hardware and software environment in order to perform their review. Scientific computing mostly relies on commodity hardware today, with two important exceptions: supercomputers and Graphical Processing Units (GPUs). Supercomputers are rare and expensive, and thus not easily accessible to a reviewer. GPUs are evolving rapidly, making it challenging to get access to an identical configuration for reviewing. Supercomputers often include GPUs, combining both problems. Resource access issues are manageable for wide-spectrum software if they are deemed sufficiently important to warrant the cost of performing audits on non-standard hardware.

Software environments have only recently been recognized as highly relevant for automated reasoning in science and beyond. They play a key role in computational reproducibility, but also for privacy and security, which are the prime motivations for the Reproducible Builds movement. The issues of managing software environments are now well understood, and two software management systems (Nix and Guix) implement a comprehensive solution. However, they have not yet found their way into mainstream computational science. In addition to ease of use issues that can be overcome with time, a major obstacle is that such management systems must control the complete software stack, which excludes the use of popular proprietary platforms such as Windows or macOS.

Assuming that the proper management of scientific software envronments will be achieved not only in theory, but also in practice, it is the size of this environment that remains a major characteristic for reviewability. The components of the execution environment required by a piece of software are called its *dependencies* in software engineering. This term expresses their importance very well: every single quality expected from a software system is limited by the quality of the components that enter in its construction. For example, no software can be more mature than its dependencies, because of the risk of software collapse [Hinsen 2019]. Reviewing software therefore requires a review of its dependencies as well. This can become an obstacle for software that has hundreds or even thousands of dependencies.

2.6 Analogies in experimental and theoretical science

For developing a better understanding of the reviewability characteristics described above, it is helpful to consider analogies from the better understood experimental and theoretical techniques in scientific research. In particular, it is helpful to examine where such analogies fail due to the particularities of software.

Experimental setups are situated. They are designed and constructed for a specific experiment, described in a paper's methods section, and reviewed as part of the paper review. Most of the components used in an experimental setup are mature industrial products, ranging from commodities (cables, test tubes, etc.) to complex and specialized instruments, such as microscopes and NMR spectrometers. Non-industrial components are occasionally made for special needs, but are discouraged by their high manufacturing cost. The use of prototype components is exceptional, and usually has the explicit purpose of testing the prototype. Some components are very transparent (e.g. cables), others are very opaque (e.g. NMR spectrometers). The equivalent of the execution environment is the physical environment of the experimental setup. Its impact on the observations tends to be well understood in the physical sciences, but less so on the life sciences, where it is a common source of reproducibility issues (e.g. [Kortzfleisch et al. 2022] or [Georgiou et al. 2022]).

The main difference to software is thus the much lower prevalence of experimental components. A more subtle difference between instruments and software is that the former are carefully designed to be robust under perturbations, whereas computation is chaotic [Hinsen 2016]. A microscope with a small defect, or used somewhat outside of its recommended operating conditions, may show a distorted image, which an experienced microscopist will recognize. Software with a small defect, on the other hand, can introduce unpredictable errors in both kind and magnitude. The increasing integration of computers and software into scientific instruments may lead to experimental setups becoming less robust as well over time.

Analogies with traditional scientific models and theories are instructive as well, where "traditional" means not relying on any form of automated reasoning. Wide-spectrum theories exist in the form of abstract reasoning frameworks, in particular mathematics. The analogue of situated software are concrete models for specific observational contexts. In between, we have general theoretical frameworks, such as evolutionary theory or quantum mechanics, and models that intentionally capture only the salient features of a system under study, pursuing understanding rather than precise prediction. Examples for the latter are the Ising model in physics or the Lotka-Volterra equations in ecology.

Abstract frameworks and general theories are the product of a long knowledge consolidation process, in which individual contributions have been reviewed, verified on countless applications, reformulated from several perspectives, and integrated into a coherent whole. This process ensures both reviewability and maturity in a way that has so far no equivalent in software development.

Opacity is an issue for theories and models as well: they can be so complex and specialized that only a handful of experts understand them. It also happens that people apply such theories and models inappropriately, for lack of sufficient understanding. However, automation via computers has amplified the possibility to deploy opaque sets of rules so much that it makes a qualitative difference: scientists can nowadays use software whose precise function they could not understand even if they dedicated the rest of their career to it.

The execution environment for theories and models is the people who work with them. Their habits, tacit assumptions, and metaphysical beliefs play a similar role to hardware and software dependencies in computation, and they are indeed also a common cause of mistakes and misunderstandings.

3 Improving the reviewability of automated reasoning systems

The analysis presented in the previous section can by itself improve the basis for trust in automated reasoning, by providing a vocabulary for discussing reviewability issues. Ensuring that both developers and users of scientific software are aware of where the software is located on the different scales I have described makes much of today's tacit knowledge about scientific software explicit, avoiding misplaced expectations.

However, scientists can also work towards improving their computational practices in view of more reviewable results. These improvements include both new reviewing processes, supported by institutions that remain to be created, and new software engineering practices that take into account the specific roles of software in science, which differ in some important respects from the needs of the software industry. The four measures I will explain in the following are summarized in Fig. 2.

3.1 Review the reviewable

As my analysis has shown, some types of scientific software are reviewable, but not reviewed today. Several scientific journals encourage authors to submit code along with their articles, but only a small number of very specialized journals (e.g., Computo, the Journal of Digital History, ReScience C) actually review the submitted code, which tends to be highly situated.

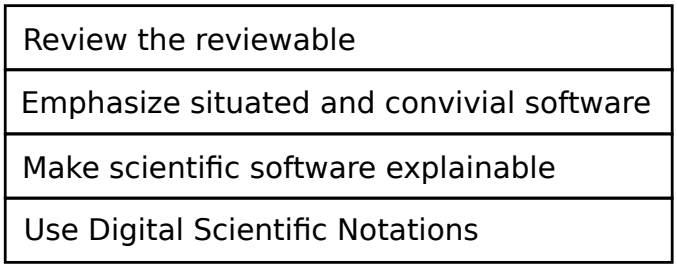

Figure 2: Four measures that can be taken to make scientific software more trustworthy.

Other journals, first and foremost the Journal of Open Source Software, review software according to generally applicable criteria of usability and software engineerging practices, but do not expect reviewers to judge the correctness of the software nor the accuracy or completeness of its documentation. This would indeed be unrealistic in the standard journal reviewing process that asks a small number of individual researchers to evaluate, as volunteers and within short delays, submissions that are often only roughly in their field of expertise.

The first category of software that is reviewable but not yet reviewed is mature wide-spectrum software. Reviewing could take the form of regular audits, performed by experts working for an institution dedicated to this task. In view of the wide use of the software by non-experts in its domain, the audit should also inspect the software's documentation, which needs to be up to date and explain the software's functionality with all the detail that a user must understand. Specifications would be particularly valuable in this scenario, as the main interface between developers, users, and auditing experts. For opaque software, formal specifications could even be made a requirement, in the interest of an efficient audit. The main difficulty in achieving such audits is that none of today's scientific institutions consider them part of their mission.

The second category of reviewable software contains situated software, which can and should be reviewed together with the other outputs of a research project. For small projects, in terms of the number of co-authors and the disciplinary spread, situated software could be reviewed as part of today's peer review process, managed by scientific journals. The experience of pioneering journals in this activity could be the basis for elaborating more widely applied reviewing guidelines. For larger or multidisciplinary projects, the main issue is that today's peer review process is not adequate at all, even in the (hypothetical) complete absence of software. Reviewing research performed by a multidisciplinary team requires another multidisciplinary team, rather than a few individuals reviewing independently. The integration

of situated software into the process could provide the occasion for a more general revision of the peer review process.

3.2 Science vs. the software industry

In the first decades of computing technology, scientific computing was one of its main application domains, alongside elaborate bookkeeping tasks in commerce, finance, and government. Many computers, operating systems, and compilers were designed specifically for the needs of scientists. Today, scientists use mostly commodity hardware. Even supercomputers are constructed to a large degree from high-grade commodity components. Much infrastructure software, such as operating systems or compilers, are also commodity products developed primarily for other application domains.

From the perspective of development costs, this evolution makes economic sense. However, as with any shift towards fewer but more general products serving a wider client base, the needs of the larger client groups take priority over those of the smaller ones. Unfortunately for science, it is today a relative small application domain for software technology.

In terms of my analysis of reviewability in section 2, the software industry has a strong focus on proprietary wide-spectrum software, with a clear distinction between developers and users. Opacity for users is not seen as a problem, and sometimes even considered advantageous if it also creates a barrier to reverse-engineering of the software by competitors. Maturity is an expensive characteristic that only few customers (e.g. banks, or medical equipment manufacturers) are willing to pay for. In contrast, novelty is an important selling argument in many profitable application domains, leading to attitudes such as "move fast and break things" (the long-time motto of Facebook founder Mark Zuckerberg), and thus favoring experimental software.

As a consequence of the enormous growth of non-scientific compared to scientific software, today's dominant software development tools and software engineering practices largely ignore situated and convivial software, the impact of dependencies, and the scientific method's requirement for transparency. However, it can be expected that the ongoing establishment of Research Software Engineers as a specialization at the interface between scientific research and software engineering will lead to development practices that are better aligned with the specific needs of science. It is such practices that I will propose in the following sections.

3.3 Emphasize situated and convivial software

As I have explained in section 2.1, many important scientific software packages are domain-specific tools and libraries, which have neither the large user base of wide-spectrum software that justifies external audits, nor the narrow

focus of situated software that allows for a low-effort one-time review by domain experts. Developing suitable intermediate processes and institutions for reviewing such software is perhaps possible, but I consider it scientifically more appropriate to restructure such software into a convivial collection of more situated modules, possibly supported by a shared wide-spectrum layer. However, this implies assigning a lower priority to reusability, in conflict with both software engineering traditions and more recent initiatives to apply the FAIR principles to software [Barker et al. 2022].

In such a scenario, a domain library becomes a collection of source code files that implement core models and methods, plus ample documentation of both the methods and implementation techniques. The well-known book "Numerical Recipes" [Press et al. 2007] is a good example for this approach. Users make a copy of the source code files relevant for their work, adapt them to the particularities of their applications, and make them an integral part of their own project. In the jargon of FLOSS, users make a partial fork of the project. Version control systems ensure provenance tracking and support the discovery of other forks. Keeping up to date with relevant forks of one's software, and with the motivations for them, is part of everyday research work at the same level as keeping up to date with publications in one's wider community. In fact, another way to describe this approach is full integration of scientific software development into established research practices, rather than keeping it a distinct activity governed by different rules. Yet another perspective is giving priority to the software's role as an expression of scientific knowledge over its role as a tool.

The evolution of software in such a universe is very different from what we see today. There is no official repository, no development timeline, no releases. There is only a network of many variants of some code, connected by forking relations. Centralized maintenance as we know it today does not exist. Instead, the community of scientists using the code improves it in small steps, with each team taking over improvements from other forks if they consider them advantageous. Improvement thus happens by small-step evolution rather than by large-scale design. While this may look strange to anyone used to today's software development practices, it is very similar to how scientific models and theories have evolved in the pre-digital era.

Since this approach differs radically from anything that has been tried in practice so far, it is premature to discuss its advantages and downsides. Only practical experience can show to what extent pre-digital and pre-industrial forms of collaborative knowledge work can be adapted to automated reasoning. Nevertheless, I will indulge in some speculation on this topic, to give an idea of what we can fear or hope for.

On the benefit side, the code supporting a specific research project becomes much smaller and more understandable, mitigating opacity. Its execution en-

vironment is smaller as well, and entirely composed of mature wide-spectrum software. Reviewability is therefore much improved. Moreover, users are encouraged to engage more intensely with the software, ensuring a better understanding of what it actually does. The lower entry barrier to appropriating the code makes inspection and modification of the code accessible to a wider range of researchers, increasing inclusiveness and epistemic diversity.

The main loss I expect is in the efficiency of implementing and deploying new ideas. A strongly coordinated development team whose members specialize on specific tasks is likely to advance more quickly in a well-defined direction. This can be an obstacle in particular for software whose complexity is dominated by technical rather than scientific aspects, e.g. in high-performance computing or large-scale machine learning applications.

The main obstacle to trying out this approach in practice is the lack of tooling support. Existing code refactoring tools can probably be adapted to support application-specific forks, for example via code specialization. But tools for working with the forks, i.e. discovering, exploring, and comparing code from multiple forks, are so far lacking. The ideal toolbox should support both forking and merging, where merging refers to creating consensual code versions from multiple forks. Such maintenance by consensus would probably be much slower than maintenance performed by a coordinated team. This makes it even more important to base such convivial software ecosystems on a foundation of mature software components, in order to avoid maintenance work necessitated by software collapse [Hinsen 2019].

3.4 Make scientific software explainable

Opacity is a major obstacle to the reviewability of software and results obtained with the help of software, as I have explained in section 2.4. Depending on one's precise definition of opacity, it may be impossible to reduce it. Pragmatically, however, opacity can be mitigated by *explaining* what the software does, and providing tools that allow a scientist to *inspect* intermediate or final results of a computation.

The popularity of computational notebooks, which can be seen as scripts with attached explanations and results, shows that scientists are indeed keen on making their work less opaque. But notebooks are limited to the most situated top layer of a scientific software stack. Code cells in notebooks refer to library code that can be arbitrarily opaque, difficult to access, and to which no explanations can be attached.

An interesting line of research in software engineering is exploring possibilities to make complete software systems explainable [Nierstrasz and Girba 2022]. Although motivated by situated business applications, the basic ideas should be transferable to scientific computing. The approach is based on three

principles. The first one is the same as for computational notebooks: the integration of code with explanatory narratives that also contain example code and computed results. Unlike traditional notebooks, Glamorous Toolkit [feenk.com 2023], the development environment built to explore these ideas, allows multiple narratives to reference a shared codebase of arbitrary structure and complexity. The second principle is the generous use of examples, which serve both as an illustration for the correct use of the code and as test cases. In Glamorous Toolkit, whenever you look at some code, you can access corresponding examples (and also other references to the code) with a few mouse clicks. The third principle is what the authors call *moldable inspectors*: situated views on data that present the data from a domain perspective rather than in terms of its implementation. These three techniques can be used by software developers to facilitate the exploration of their systems by others, but they also support the development process itself by creating new feedback loops.

3.5 Use Digital Scientific Notations

As I have briefly mentioned in the introduction, specifications are contracts between software developers and software users that describe the expected behaviour of the software. Formal specifications are specifications written in a formal language, i.e. a language amenable to automated processing. There are various techniques for ensuring or verifying that a piece of software conforms to a formal specification. The use of these tools is, for now, reserved to software that is critical for safety or security, because of the high cost of developing specifications and using them to verify implementations.

Technically, formal specifications are *constraints* on algorithms and programs, in much the same way as mathematical equations are constraints on mathematical functions [Hinsen 2023]. Such constraints are often much simpler than the algorithms they define. As an example, consider the task of sorting a list. The (informal) specification of this task is: produce a new list whose elements are (1) the same as those of the input list and (2) sorted. A formal version requires some additional details, in particular a definition of what it means for two lists to have "the same" elements, given that elements can appear more than once in a list. There are many possible algorithms conforming to this specification, including well-known sorting algorithms such as quicksort or bubble sort. All of them are much more elaborate than the specification of the result they produce. They are also rather opaque. The specification, on the other hand, is immediately understandable. Moreover, specifications are usually more modular than algorithms, which also helps human readers to better understand what the software does [Hinsen 2023].

The software engineering contexts in which formal specifications are used today are very different from the potential applications in scientific computing

that I outline here. In software engineering, specifications are written to formalize the expected behavior of the software *before* it is written. The software is considered correct if it conforms to the specification. In scientific research, software evolves in parallel with the scientific knowledge that it encodes or helps to produce. A formal specification has to evolve in the same way, and is best seen as the formalization of the scientific knowledge. Change can flow from specification to software, but also in the opposite direction. Moreoever, most specifications are likely to be incomplete, leaving out aspects of software behavior that are irrelevant from the point of view of science (e.g. resource management or technical interfaces such as Web APIs) but also aspects that are still under exploration and thus not yet formalized. For these reasons, I prefer the term *Digital Scientific Notation* [Hinsen 2018b], which better expresses the role of formal specifications in this context.

4 Conclusion

My principal goal with this work is to encourage scientists and research software engineers to reflect about their computational practices. Why, and to what degree, do you trust your own computations? How reliable do they have to be to support the conclusions you draw from their results? Why, and to what degree, do you trust the computations in the papers you read and cite? Do you consider their reliability sufficient to support the conclusions made?

These questions are abstract. Answering them requires considering the concrete level of the specific software used in a computation. The five categories I have discussed in section 2 should help with this step, even though it may be difficult at first to evaluate the software you use on some of the scales. Situated software is easy to recognize. The size of a software environment is not difficult to measure, but it requires appropriate tools and training in their use. Likewise, the evaluation of maturity is not difficult, but requires some effort, in particular an examination of a software project's history. Conviviality is hard to diagnose, but rare anyway. This reduces the examination to Open Source vs. proprietary, which is straightforward.

The transparency vs. opacity scale deserves a more detailed discussion. Most experienced computational scientists make sure to examine both intermediate and results for plausibility, making use of known properties such as positivity or order of magnitude. But plausibility is a fuzzy concept. Software is transparent only if users can check results for correctness, not mere plausibility. The strategies I proposed (sections 3.3, 3.4 and 3.5) have the goal of making such correctness checks easier. If plausibility is all we can check for, then the software is opaque, and its users are faced with a dilemma: if their results are

neither obviously correct nor obviously wrong, are they entitled to consider them good enough? In practice they do, because the only realistic alternative would be to stop using computers. We even tend to consider popularity, which roughly means "this software is used by many people who didn't find anything obviously wrong with it", as an indicator for trustworthiness. Soergel [Soergel 2014] considers this "trust by default" misplaced, given what software engineering research tells us about the frequency of mistakes. Examples from the reproducibility crisis support this view that scientists tend to overestimate the reliability of their work in the absence of clear signs of problems.

Computational reliabilism, proposed by Durán and Formanek [Durán and Formanek 2018], offers a way out of this dilemma: it says that we can justify trust by default if we have a large body of experience reports about our software, of which a majority is favorable. Independent reviews would be particularly valuable experience reports, since their authors specifically look for potential problems. However, the large body of experience required for the reliabilism argument can be gathered only for mature software.

The ideal structure for a reliable scientific software stack would thus consist of a foundation of mature software, on top of which a transparent layer of situated software, such as a script, a notebook, or a workflow, orchestrates the computations that together answer a specific scientific question. Both layers of such a stack are reviewable, as I have explained in section 3.1, but adequate reviewing processes remain to be enacted.

The remaining issue is experimental opaque software packages which, as I explained in section 2.2, are numerous in science. Evolving them towards maturity requires time, a large user base, and high software engineering standards. Mitigating opacity, e.g. by adopting the strategies I have proposed, requires a significant effort. Reliability comes at a cost. Making good choices requires a cost-benefit analysis in the context of a specific research project. The arguments for the choice should be published as part of any research report, to permit readers an assessment of the reliability of the reported findings.

The difficulty of reviewing scientific software also illustrates the deficiencies of the current digital infrastructure for science.[3] The design, implementation, and maintenance of such an infrastructure, encompassing hardware, software, and best practices, has been neglected by research institutions all around the world, in spite of an overtly expressed enthusiasm about the scientific progress made possible by digital technology. The situation is improving for research data, for which appropriate repositories and archives are becoming available. For software, the task is more complex, and hindered by the

[3]For more examples, see [Saunders 2022].

contagious neophilia of the software industry. Scientists, research software engineers, research institutions, and funding agencies must recognize the importance of mature and reliable infrastructure software, which requires long-term funding and inclusive governance.

References

Barker, M., Chue Hong, N.P., Katz, D.S., et al. 2022. Introducing the FAIR Principles for research software. *Scientific Data 9*, 1, 622.

Chapter 1: For Want of a Printer. 2002. In: *Free as in freedom: Richard Stallman's crusade for free software*. O'Reilly, Sebastopol, Calif. : Farnham.

Courtès, L. 2021. What's in a package. *GuixHPC blog*. https://hpc.guix.info/blog/2021/09/whats-in-a-package/.

Durán, J.M. and Formanek, N. 2018. Grounds for Trust: Essential Epistemic Opacity and Computational Reliabilism. *Minds and Machines 28*, 4, 645–666.

feenk.com. 2023. Glamorous Toolkit. https://gtoolkit.com.

Georgiou, P., Zanos, P., Mou, T.-C.M., et al. 2022. Experimenters' sex modulates mouse behaviors and neural responses to ketamine via corticotropin releasing factor. *Nature Neuroscience 25*, 9, 1191–1200.

Hettrick, S. 2016. *Research Software Sustainability*. Knowledge Exchange.

Hinsen, K. 2016. The Power to Create Chaos. *Computing in Science & Engineering 18*, 4, 75–79.

Hinsen, K. 2018a. Reusable Versus Re-editable Code. *Computing in Science & Engineering 20*, 3, 78–83.

Hinsen, K. 2018b. Verifiability in computer-aided research: The role of digital scientific notations at the human-computer interface. *PeerJ Computer Science 4*, e158.

Hinsen, K. 2019. Dealing With Software Collapse. *Computing in Science & Engineering 21*, 3, 104–108.

Hinsen, K. 2023. The nature of computational models. *Computing In Science & Engineering 25*, 1, 61–66.

Hocquet, A. and Wieber, F. 2017. "Only the Initiates Will Have the Secrets Revealed": Computational Chemists and the Openness of Scientific Software. *IEEE Annals of the History of Computing 39*, 4, 40–58.

Illich, I. 1973. *Tools for conviviality*. Calders and Boyars, London.

Jumper, J., Evans, R., Pritzel, A., et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature 596*, 7873, 583–589.

Kell, S. 2020. Convivial design heuristics for software systems. *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, ACM, 144–148.

Ko, A.J., Abraham, R., Beckwith, L., et al. 2011. The state of the

art in end-user software engineering. *ACM Computing Surveys 43*, 3, 1–44.

Kortzfleisch, V.T. von, Ambrée, O., Karp, N.A., et al. 2022. Do multiple experimenters improve the reproducibility of animal studies? *PLOS Biology 20*, 5, e3001564.

Leonelli, S. 2022. Open Science and Epistemic Diversity: Friends or Foes? *Philosophy of Science 89*, 5, 991–1001.

Merali, Z. 2010. Computational science: ...Error. *Nature 467*, 7317, 775–777.

Nardi, B.A. 1993. *A small matter of programming: Perspectives on end user computing*. MIT Press, Cambridge, MA.

Nielsen, M. 2023. How is AI impacting science? https://michaelnotebook.com/mc2023/.

Nierstrasz, O. and Girba, T. 2022. Making Systems Explainable. *2022 Working Conference on Software Visualization (VISSOFT)*, IEEE, 1–4.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. 2007. *Numerical recipes: The art of scientific computing*. Cambridge University Press, Cambridge, UK ; New York.

Saunders, J.L. 2022. Decentralized Infrastructure for (Neuro)science. http://arxiv.org/abs/2209.07493.

Shaw, M. 2022. Myths and mythconceptions: What does it mean to be a programming language, anyhow? *Proceedings of the ACM on Programming Languages 4*, HOPL, 1–44.

Shirky, C. 2004. Situated Software. https://web.archive.org/web/20040411202042/http://www.shirky.com/writings/situated_software.html.

Soergel, D.A.W. 2014. Rampant software errors undermine scientific results. *F1000Research 3*, 303.

Wilkinson, M.D., Dumontier, M., Aalbersberg, Ij.J., et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data 3*, 1, 160018.