https://doi.org/10.1038/s43588-024-00651-2

Software in science is ubiquitous yet overlooked

Alexandre Hocquet, Frédéric Wieber, Gabriele Gramelsberger, Konrad Hinsen, Markus Diesmann, Fernando Pasquini Santos, Catharina Landström, Benjamin Peters, Dawid Kasprowicz, Arianna Borrelli, Phillip Roth, Clarissa Ai Ling Lee, Alin Olteanu & Stefan Böschen

Check for updates

Software is much more than just code. It is time to confront the complexity of licenses, uses, governance, infrastructure and other facets of software in science. Their influence is ubiquitous yet overlooked.

In March 2020, Neil Ferguson, the scientist whose epidemiology model was used to justify COVID lockdown policies in the UK and around the world, was urged to make his model's source code public. The model received some criticism on scientific grounds, but the most vocal objections targeted its software engineering aspects, calling it poorly designed, written and documented[1]. Such a culture clash is not surprising to some computational scientists, whose daily routine consists of designing, writing, maintaining, supporting, testing, debugging, adapting to new hardware, documenting, sharing, licensing and packaging a piece of software. Both computational researchers and software engineers are used to interacting with different temporalities, constraints, norms and work cultures.

In June 2020, in the wake of Ferguson's controversies, colleagues across the sciences and humanities published a timely and relevant manifesto in *Nature* that proposes "five ways to ensure that models serve society"[2]. Yet the manifesto does not mention the concept of software in their consideration of models. We believe this is lacking because models and software are entangled in science, and software does critical work that models cannot perform on their own.

Software is indeed difficult to define, often being mistaken for code or algorithms. As historian of computing Thomas Haigh puts it: "Software always involves packaging disparate elements such as computer code, practices, algorithms, tacit knowledge, and intellectual property rights into an artifact suitable for dissemination". Scientific software involves a diversity of practices regarding programming, governance, licensing, distribution, maintenance and support. It is developed and used across a myriad of scientific disciplines and programming traditions. It ranges in size from personal 'scripts' to huge projects involving entire communities and global infrastructure. It encompasses freely shared code as well as commercial packages.

In this Comment, we emphasize the complexity of scientific software as a multifaceted socio-technical (and historically grown) system. We describe facets of software that we define as vantage points from which the different dimensions of software can be understood. The multifaceted nature of software implies that the work done by software has technical, legal, sociological and epistemic consequences. Models and software are entangled in computational science, and much remains to be done to comprehend these consequences. We also point out the diversity of situations involving software in computational science, which further complicates how to approach software facets. We highlight a few case studies, with the hope that this starting conversation about software will be enriched by further input.

Engineering

Ferguson's story reveals something important and widespread[1] about a culture clash between science and software engineering. Some software professionals may regard scientists as end-user programmers, yet scientists do not necessarily share the same norms, aims and practices as software engineers.

For example, given that validation and verification are often intertwined, test suites designed by scientists may focus more on the stability and reproducibility of simulation results than on the efficiency of the code or the structure of the program. In some scientific projects, risk-averse approaches oppose agile methods[3]. Software longevity may not be understood in the same way; whereas software engineers consider adaptation to new hardware, operating systems or platforms to be essential, backward compatibility, stability and replicability are more important for scientists.

Also, software engineers may tend to account for diverse potential users, whereas scientists sometimes devise software for the exclusive use of their close collaborators. Therefore, standards pertaining to portability or the user-friendliness of interface design may differ substantially. Tasks such as software maintenance or bug fixes might be idiosyncratic and have temporalities of their own.

To manage the unmanageable in increasing software complexity and in the labyrinth of available libraries, communities develop guidelines on how to use and improve scientific software in alignment with scientific research norms. In doing so, the field of research software engineering has emerged, aiming to bridge both cultures. The growing importance of research software engineering underlines the need to study the diversity of working cultures in scientific software. To this end, the FAIR (findable, accessible, interoperable and reusable) principles for research software[4] and similar initiatives should be assessed and compared to established practices in open source communities. Moreover, the issue of recognition or credit for engineering work in science is also pressing.

Governance

Governance — that is, the social structure of a software project — is an important facet of software that has more than one author. The way software development and maintenance are collectively organized affects the science that relies on it.

In the computational chemistry project Q-Chem, a professional workforce dedicated to development and maintenance is financed by software package sales. The project is thus commercial, centralized

and proprietary, which is supposed to ensure its stability[5]. Open source would arguably be a better way forward for transparency, but it does not solve the problem of who is able to commit what (and themselves) to a project. The SciPy community consists of scientist-developers with diverse interests, both in terms of numerical techniques and scientific disciplines, and different computational needs. Even though the SciPy libraries are open source, development choices tied to hierarchies in governance or the representation of scientific disciplines in the community influence how practical their use can be in different communities. Forking can mitigate the diversity issue but is not always an effective solution, because it tends to fragment or even divide open source communities.

Users and funders are sometimes not aware of governance issues. To understand a software project, one should situate it within diverse types of social structure[6]. Moreover, governance should include both developer and user communities, because their perspectives and priorities often differ considerably.

Licensing

Beyond governance, software is also concerned with the administration of its uses. Licenses are the contracts that software authors and users must abide by. Although definitely entangled with governance, licensing takes a legal rather than a social perspective, translating intellectual property rights into the world of software.

For example, licensing may differ for academic and industrial users. The MacroModel licensing policy distinguishes between discounted academic licenses that forbid tinkering with certain model parameters, and industrial licenses that do allow such tinkering. Some scientists have argued that academic licenses restrain scientific potentialities while the industrial ones raise reproducibility issues linked to uncertain versioning[7].

Given that the license defines what the user is entitled to do, the actionability of a model embedded in a piece of software follows directly from licensing policies. Yet such end-user license agreements are notoriously seldom taken into account by users. Indeed, much scientific software lacks any licensing policy at all. Even within open source projects, license differences affect the possibilities for the reuse and combination of software[8]. For example, 'Permissive' licenses such as MIT, Apache or BSD differ from 'copyleft' licenses such as GPL or LGPL. Better literacy regarding licensing issues is desirable, as these issues illustrate a tension: scientific software is at once a valuable technical artifact subject to intellectual property, and an expression of models and methods whose scientific value comes from disclosure and sharing.

Circulation

According to Haigh[9], software is only as useful as it is "suitable for dissemination", but what this means depends on the context. As soon as exchange is envisioned for a computational project, software is what enables code to be packaged for traveling through space (that is, across different communities or userbases), time (because of maintenance and support), pieces of hardware (for instance, for portability), and software environments (for backwards compatibility).

For example, the history of the Gaussian computational chemistry package is a decades-long story of strategic changes. Gaussian began as a freely available source code, and eventually a company was founded to distribute and sell Gaussian as a software suite. The Gaussian story, however, is not merely one of software commodification. For Gaussian, maintaining control over official versions is key for the accountability and durability of the software project in the context of diverse hardware and portability initiatives. That is why the source code of Gaussian is provided, as a warrant for transparency, but many corporate actions forbid users to modify it, to avoid proliferation of uncontrolled and inconsistent versions of the program[7]. The history of distribution strategies of the Gaussian package over decades sheds light on different strategic choices regarding reproducibility. Nowadays, it is the evolution of software rather than hardware environments that needs to be taken into account.

Software environments can be stored and transferred, which is the role of container technologies such as Docker. These have become popular in scientific computing, alongside version control systems that permit source code changes to be tracked. The missing link between version control and executable containers is a record of the transformation process from source code to executable. This task is performed by compilers and related tools and orchestrated by package managers. However, some package managers do not keep track of the versions of compilation tools, which are subject to change as well. A different compiler can cause unpredictable changes in the results of calculations. Software management tools such as Nix or Guix ensure full provenance tracking, but their use is still far from widespread.

Infrastructure

Infrastructure studies have revealed issues of long-term development, scale and the interplay of technical and organizational structures, as well as tensions between what is planned and what emerges. Infrastructure constitutes a software facet of its own, especially when software projects involve or support entire communities[10].

Nowadays, platforms as infrastructure are becoming increasingly detached from their hardware support. In science, this means that the portability of models to a variety of competing hardware is less of an issue than it was a few decades ago, whereas software infrastructure is nowadays more fragile, described by historian Paul Edwards as "flammable"[10]. Large scientific instruments such as telescopes are now well established elements of scientific infrastructure, and have corresponding funding models, but the same cannot yet be said for software, which has a similarly fundamental role.

For example, the field of computational neuroscience is striving to separate the formal specification of concrete neural network models from generic simulation engines, which can run a variety of models from different research groups. This kind of generic engine rests on software infrastructure suffering from distinctive long-term development and maintenance issues[11]. With many research groups depending on the continued usability of the shared engine, its maintenance must be governed and funded collegially and on a timescale extending far beyond that of a typical research grant[12].

Budgets for software maintenance must be planned and approved as long-term investments, just like the budgets for traditional scientific infrastructure such as particle accelerators. For this to happen, science funding and policy actors need a better understanding of how software is made usable and for whom.

Embedded theory

In scientific models, software embeds theory, and different versions of a piece of software entail different versions of a model or its parameters, or even different underlying theoretical principles. In the context of in silico experiments in climate modeling[13], changes in the software might imply changes in the models and theories they are based upon and, thus, correspond to different settings for such experiments.

To ensure consistency, some climate researchers have adopted methods for comparative assessment of models and parameters that also include evaluation of the software.

Another example is the effort to standardize mathematical concepts in computational neuroscience. An analysis of connectivity patterns in neural network models implemented either in terms of predefined routines of a generic simulator or as custom code in a general-purpose programming language has unveiled a diversity of interpretations of its core connectivity concept that challenges reproducibility[14].

The problem is not only one of theoretically diverse conceptions of connectivity, but also one of the implementation of any of these conceptions across different software frameworks such as MATLAB, NEURON or NEST. Using different pieces of software thus means using different connectivity theories. The way forward lies in developing standardized ontologies of the terms the community is using, backed up not only by mathematical definitions but also by reference software implementations.

Users

Because users rarely form a homogeneous group, the potential diversity of uses accentuates the underlying complexity and diversity of software. As a medium, software constitutes an interface within and through which users operate. As such, software sets operational affordances that organize users' interactions with models. For example, a command-line interface and the use of scripts may enhance reproducibility because invocations can be recorded[15], whereas a graphical user-friendly interface might enhance usability. Beyond the command-line interface versus graphical user-friendly interface debate, users' interactions with software must be understood as being bound to research cultures. For example, in protein crystallography, user interfaces shape the handling of models on the screen, but the interface design itself is influenced by a common understanding of molecules through physical ball-and-stick models[16].

The diversity of application scenarios often transcends the scientific context itself. For instance, in water management, computer models are supposed to be used by water management professionals. Although such programs are nowadays published as open source code, they are less frequently used by professionals other than the scientists involved in their creation, as their design may be somewhat opaque to non-scientists. For a scientific computer model to become usable in water management, extensive development effort is required to transform it into a software package suited to a wider audience. This translation process of turning models into usable software is pivotal[17].

Even within scientific communities, such as that of functional magnetic resonance imaging, the engagement and retention of users is challenged by competing software packages. Usability assessment is crucial because user experience choices presumably affect the scientific analysis itself[18]. Beyond code, reflexive studies about scientific software need a broader perspective to encompass the entire trajectory from the context of development to the context of application.

Conclusion

Our argument is that software influences models and their outputs, just as it shapes (and is shaped by) scientific practices. That software is multifaceted implies that the work software performs has not only technical or sociological but also epistemic consequences. Concerns about software robustness, maintenance and durability, reproducibility and actionability, dissemination and consistency, all have epistemic dimensions.

Some of the issues are currently being addressed. To name some initiatives, Software Heritage endeavors to preserve all available versions of scientific code; Software Carpentry promotes computational literacy; the Software Sustainability Institute and the Research Software Alliance work towards better recognition; the ReScience C journal aims at replicating results.

Nevertheless, more is needed. Coming back to the abovementioned manifesto about models and society[2], it should now be clear that the entangled epistemic, social and technical dimensions of software give substance to the issues raised in said manifesto[2].

The diversity of software practices implies that a form of interdisciplinarity is key to understanding software facets. We should gather perspectives from different academic (such as computational scientists as well as humanists and social scientists) and professional backgrounds (such as developers, users, maintainers, and so on) to reveal the tensions between different meanings of software.

In this spirit, more case studies in various scientific fields and epochs should help us to understand the entanglement of software and models within their diversity and different temporalities. We hope this will improve our comprehension of the situatedness of software and enrich the conversation we are calling for.

Alexandre Hocquet [1,2] ✉, Frédéric Wieber [1], Gabriele Gramelsberger [2], Konrad Hinsen [3,4], Markus Diesmann [5], Fernando Pasquini Santos [2,6], Catharina Landström [2,7], Benjamin Peters [2,8], Dawid Kasprowicz [2], Arianna Borrelli [2,9], Phillip Roth [2], Clarissa Ai Ling Lee [2,10], Alin Olteanu [2] & Stefan Böschen [2]

[1]Archives Poincaré, Université de Lorraine, Nancy, France. [2]Käte Hamburger Kolleg, Cultures of Research, RWTH, Aachen, Germany. [3]Centre de Biophysique Moléculaire, CNRS, Orléans, France. [4]Synchrotron SOLEIL, Saint Aubin, France. [5]Institute for Advanced Simulation (IAS-6), Forschungszentrum Jülich, Jülich, Germany. [6]Department of Computer Science, Calvin University, Grand Rapids, MI, USA. [7]Science, Technology and Society Division, Chalmers University of Technology, Gothenburg, Sweden. [8]Department of Media Studies, The University of Tulsa, Tulsa, OK, USA. [9]History of Science Institute, TU Berlin, Berlin, Germany. [10]Center for Interactive Media, Multimedia University, Cyberjaya, Selangor, Malaysia.
✉e-mail: alexandre.hocquet@univ-lorraine.fr

Published online: 01 July 2024

References

1. Thimbleby, H. *Computer J.* **67**, 1381–1404 (2024).
2. Saltelli, A. et al. *Nature* **582**, 482–484 (2020).
3. Kelly, D. *J. Syst. Softw.* **109**, 50–61 (2015).
4. Barker, M. et al. *Sci. Data* **9**, 622 (2022).
5. Hocquet, A. & Wieber, F. *Eur. J. Phil. Sci.* **11**, 38 (2021).
6. Schrape, J.-F. *Convergence* **25**, 409–427 (2017).
7. Hocquet, A. & Wieber, F. *IEEE Ann. Hist. Comput.* **39**, 40–58 (2017).
8. Morin, A. et al. *PLOS Computat. Biol.* **8**, e1002598 (2012).
9. Haigh, T. *Commun. ACM* **56**, 31–34 (2013).
10. Edwards, P. N. Platforms are infrastructures on fire. In *Your Computer is on Fire* (eds Mullaney, T. S. et al.) 313–336 (MIT Press, 2021).
11. Einevoll, G. et al. *Neuron* **102**, 735–744 (2019).
12. Knowles, R. et al. *Nat. Computat. Sci.* **1**, 169–171 (2021).
13. Gramelsberger, G. et al. *J. Adv. Model. Earth Syst.* **12**, e2019MS001720 (2019).
14. Senk, J. et al. *PLOS Computat. Biol.* **18**, e1010086 (2022).
15. Baker, M. *Nature* **541**, 563–565 (2017).

16. Myers, N. *Rendering Life Molecular: Models, Modelers, and Excitable Matter* (Duke Univ. Press, 2015).
17. Landström, C. *TATuP J. Technol. Assess. Theory Practice* **32**, 36–42 (2023).
18. Pasquini, F. et al. in *Proc. 18th Int. Joint Conf. Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)* Vol. 2, 63–72 (SCITEPRESS, 2023).

Acknowledgements
The joint research was funded by the Käte Hamburger Kolleg Cultures of Research for Advanced Study in the Humanities with funds from the German Federal Ministry of Education and Research.

Author contributions
All authors contributed equally.

Competing interests
The authors declare no competing interests.

Additional information
Peer review information *Nature Computational Science* thanks Stefanie Betz and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.