From Secret Sauce to Shared Recipes: Open Source Software in Mechanics

In addition to scientific articles, data, scripts, code, software, and their documentation are essential ingredients for scientific advancement, and in particular for computational mechanics. Relying on open-source tools and shared data can not only accelerate the dissemination of advances in methods and algorithms in science and engineering, but also catalyze collaborations and consolidate a broader community around computational tools and software. This article reviews current trends in open-source software in mechanics in today's rapidly changing landscape, where researchers, users, and developers are increasingly supported by powerful AI tools. The focus is on community-led scientific open-source software. Some concrete recipes and good practices are suggested for sharing software, data, and articles under the principles of Open Science.

Introduction: Types of Scientific Software

In science, and especially in computational mechanics, we constantly use various software tools and develop our own. We can broadly classify them as tools developed by commercial organizations (e.g., `Abaqus` by Dassault Systèmes SIMULIA or `Matlab` by The MathWorks), by a broad open-source community (e.g., `SciPy` or `LaTeX`), or by the scientific community itself (e.g., `PETSc` by Argonne National Laboratory). In each case, the software can be paid, conditionally paid, free, or even open-source, e.g., `AlphaFold` by Google DeepMind (open-source software developed by a commercial company) or `Z-set` (closed-source FEA software developed in academia at Mines Paris and at the French aerospace laboratory Onera, free for academia and paid for industry). The license types can also vary. The number of possible combinations is very high, and it would be too ambitious to try to cover all cases here. Therefore, let us focus on a simplified set of software classes:

- `$€Software` - closed and paid software like `ANSA` or `CFD Pointwise` for meshing, `Catia` or `SolidWorks` for CAD, and `Comsol` and `Ansys` for FEA;
- `LabClosedSoftware` - closed-source software developed, at least initially, by a scientific group and having commercial potential, like `Thermo-Calc`, thermodynamic software developed at KTH, `VASP` for DFT computations developed at the University of Vienna, and many other small, specific software tools often unknown to the general public.
- `OpenSoftware` - open-source software developed by the broad open-source community or by the scientific community itself, possibly in collaboration with industry, like `gmsh` for meshing, `FreeCAD` and `OpenCascade` for CAD, `LAMMPS` for MD, `MUMPS` and `PETSc` for linear solvers, `MFEM` and the `FEniCS Project` for FEA, and `ParaView` for visualization.

Powerful `$€Software` ensures fast access to advanced computational tools and, of course, enables science to move forward without spending precious time on learning comparable open-source tools or developing yet another in-house tool with limited functionality, performance, and testing. Many questions in mechanics and computational mechanics can be readily investigated with the aid of existing commercial tools, some of which provide users with relatively flexible environments enabling their

extension, such as UMAT user materials in Abaqus or user-programmable features, UPFs, in Ansys. Moreover, using such software, which are widely adopted in industry, by undergraduate and graduate students, as well as early-career scientists, helps prepare them for engineering roles in R&D departments if they choose this path. There are, however, several drawbacks to such software: the user is unaware of all the internal "cuisine" and has to rely on some features of the software as a black box; such software requires an additional budget, sometimes quite high; the obtained results are not reproducible by the broad community; and the flexibility of such tools is sometimes insufficient to carry out high-level research in computational mechanics.

There are several commercially successful LabClosedSoftware cases that transformed software developed in academic labs into business products. This can be small-scale engineering software constructed during a collaboration between academia and industry and sold to the latter, or it can be more general-purpose software like the above-mentioned VASP and Thermo-Calc, as well as many other commercial products that trace their origins to research labs or universities. Such a transfer from the lab to industry requires a high-quality, accurate, and robust product with a maintenance and support team. As an upside, developing such advanced closed-source software that solves industry pain points can give a competitive advantage to the developing team, enabling, e.g., privileged partnerships based on exclusive internal expertise and/or faster monopoly-like scientific publications in this subdomain before other groups can catch up. However, the downside is a shift of focus from science to business, which operates under strongly different incentives. Of course, not all closed initiatives transform into commercially successful products, and therefore the choice of the type of software into which the group will invest the precious time of researchers and public money is not trivial; weighing all risks is usually very tricky, especially today in such a rapidly changing software landscape. Having a secret sauce recipe and precious data can make you and your host institution face an uneasy choice: preserve this competitive advantage by keeping it closed, or share it with the community. The second choice often feels more natural to a science-oriented mindset and is probably better for scientific progress too.

Finally, OpenSoftware allows the accumulated contributions of publicly- and sometimes industry-funded research and development to be shared with everybody for free [see License section], with all the associated upsides and downsides. The advantage of this type of software is its alignment with the principles of verifiability and reproducibility, as well as with the general principles of Open Science, where open-source software plays a central role. In addition, providing high-quality tools to the community enables other groups to focus directly on their niche contributions; ultimately, the community as a whole benefits. Relying on open-source tools makes research results more easily reproducible compared to commercial software, and can also gather a community to push the software forward through extensive testing, the addition of new advanced capabilities, etc. Among the downsides, there is a certain shift of focus from science to development, i.e., spending time on support and maintenance, as well as on community management. Nevertheless, the benefits for science can be considerable.

In this note, we will focus on OpenSoftware, i.e., open-source software developed by research teams or larger sub-communities. Nevertheless, this note also remains relevant to some extent even for small-scale scripts, for example, those used to post-process data and construct figures. At this scale, nowadays, it is good practice to share such data and scripts along with the scientific paper to make the research more open and reproducible, for the good of both the community and the authors themselves.

DIFFICULTIES AND BENEFITS OF OPEN-SOURCE SOFTWARE

The difficulties of developing an open-source project are listed below. (I) - if one wants to create completely new software, the time and resources needed to reach a *production-ready* state can be huge. Nowadays, there are plenty of ready-to-use open-source tools flexible enough to accommodate the boldest novelties and ideas*. Therefore, one has to propose something truly novel to justify the expense of resources: one's own time, that of early-career scientists, and thus public money. (II) - sustaining an ongoing open-source research software project

*It is probably unnecessary here to argue that there is not much value for the community in creating yet another finite-element software package with basic functionality. But even if the software is truly novel and valuable, it should solidly rely on existing infrastructure components and integrate smoothly with the existing landscape: for example, in terms of, for instance, linear solvers, visualization output, and data formats.

requires resources (engineers and developers, possibly a support team) and a lot of discipline and dedication. Quite often, successful projects rely on a single or a very few enthusiastic individuals investing a lot of time and energy. In the absence of such individuals, most projects cannot take off or die rapidly. This number of key developers is measured by a *"bus factor"* or a *"truck factor"* [1]: having $bf = n$ means that if the n key developers are "hit by a bus/ truck" (they leave the project), the whole project can collapse. Having $bf > 1$ is therefore a sign of greater resilience*. (III) - quite often, in computational mechanics, most developers are applied mathematicians or researchers in solid/fluid mechanics or material science, more rarely computer scientists, and even more rarely professional code developers or architects with professional software-development experience. Therefore, the quality of academic software is often lower than that of software constructed by commercial developers. Nevertheless, true double competence does exist in computational mechanics, and nowadays, with intelligent use of AI tools, which can be very helpful in code writing and infrastructure usage, many problems and gaps in experience can be smoothed out.

On the other hand, the upsides of open-source software are numerous. (1) - it is essential for research in computational mechanics to have full control over every element of the software, and this control is inherent in open-source software; (2) - contrary to small projects that are born and die during one undergraduate or graduate project, having a larger open-source software package makes it possible to integrate such contributions into a single framework with documentation and tests (of course, the quality of contributions depends on the group's culture and established practices); (3) - attracting a larger community helps enhance available features, improve examples and documentation, facilitate the exchange of experience and good practices, detect bugs, and strengthen testing; (4) - when input and output data are properly shared along with, e.g., scientific articles, open-source software enables the reproducibility of results, which is important for scientific progress; (5) - at the scale of individual developers, their contributions to open-source projects are praised by the community, encouraging passionate individuals to contribute to these *digital commons*†.

The lists of upsides and downsides are not exhaustive and can vary strongly depending on the project type, research group culture, administration of the host institution, particular domain, and so on.

Sharing Open-Source Software

In most situations, if you are contributing to an open-source project, everything is already set up either on a locally deployed GitLab instance (thanks to the developers of GitLab!) or directly on GitHub. Other possibilities exist, for example, the non-profit community -led platform for free open-source projects Codeberg and its deployable version Forgejo‡. There is also the paid hosted service SourceHut and the commercial platform Bitbucket. Having at least two hosting repositories, mirrored automatically makes a project more robust and independent: a hosting provider can restrict or suspend account access, sometimes for reasons unrelated to the project itself, such as an automated security flag, and a project with no alternative remote becomes temporarily or permanently inaccessible until the issue is resolved. There are, however, several exceptions in sharing practices: for example, the powerful open-source linear solver MUMPS is distributed under the CeCILL-C license, but to get the source code, one needs to fill in an online form]. Apart from having a living repository, it is helpful to have permanent release versions or snapshots stored on dedicated platforms, for example, Software Heritage [5]. A Zenodo repository would also do the job, but it is not designed for this particular purpose and therefore, contrary to Software Heritage, does not rely on version control and thus requires storing the full raw code repository every time. But of course, to ensure true reproducibility, one needs to think about dependencies,

**To give several examples with rough estimations: $bf = 1$ for Mfront, Tamaas; $bf = 2$ for LibMesh; $bf = 4$ for 4C-multiphysics; $bf > 5$ for MFEM, Moose, and Trilinos.*

†Of course, not every academic organization values this hidden but crucial work of development and maintenance, but the trend is positive, and we believe that such contributions should be valued at the same level as scientific papers.

‡The Dutch administration, for instance, has deployed Forgejo at code.overheid.nl to migrate all administration-related open-source projects from Github and Gitlab.

compilers, operating systems, and so on. For Python projects, having an appropriate description of the `conda` environment or detailed requirements for `pip` may be enough. But for multi-language projects with various dependencies, container-based solutions are required, such as `Docker` or `Singularity/Apptainer`; an alternative HPC-oriented solution is a `Spack` environment.

An example of bad practice when sharing open-source software would be to share only a snapshot of the code on `Zenodo`: it does not allow users to see the history of development, nobody can contribute or open an issue, and every new version requires sharing the entire repository again.

PROMOTING OPEN-SOURCE SOFTWARE

Sharing your open-source software is not a sufficient condition for its usefulness to the community. Of course, the starting point is to use the software by research group itself, including undergraduate students and early-career scientists. Good communication can be very helpful in demonstrating the utility of your open-source software to a broader community: showing that it solves a problem better than existing software can help attract new users and, eventually, contributors, which is of course very beneficial for the project. Beyond the added value of the software itself, there are several essential ingredients, such as (i) efficient sharing of information at conferences* and in private, (ii) continuous use of the software by scientific collaborators and early-career scientists passing through the group, and (iii) ease of use and availability of documentation, tutorials, and examples†.

A living open-source software project is expected to solve a problem or a set of problems for a scientific community. Therefore, many developers pay particular attention to the use of their software in scientific papers; see, for example, software for microstructure generation `Neper`'s application page or the molecular dynamics software `LAMMPS` publications page with an impressive collection of journal covers based on MD simulation results.

Quite often, one of the first publications featuring the software serves as a reference software paper, as for example for `libMesh` [2]. An alternative option is to write a software-dedicated article in `The Journal of Open Source Software` (JOSS), hosted on a modern platform and benefiting from an open review process aimed at checking the quality of the code, documentation, examples, contribution friendliness, and the position of the software in the existing software landscape, as for example for `MoFEM` [3].

To consolidate and promote the software, organizing user meetings and conferences is very helpful and can catalyze software growth, as well as the exchange of ideas and good practices between users and contributors. This practice is adopted by many software projects, such as `gmsh`, `FreeFEM++`, the `FEniCS Project`, `MFEM`, `Moose`, `Code_Aster`, `MFront`, `LAMMPS`, and many others.

LICENSING

When sharing your open-source software, an appropriate license should be selected [6,7]. Essentially, the choice is made between a *copyleft* license and a *permissive* one. By a copyleft license, sometimes also referred to as a *contaminating* license, we mean a license that requires derivative code, or software that uses it under certain conditions, to be distributed under the same license. The most well-known examples are: **GPL** (GNU General Public License, v2 and v3) - the strongest copyleft license, widely used for standalone software (`Linux kernel`, `GCC`, `Code_Aster` the French EDF finite element software); **LGPL** (GNU Lesser General Public License) - a weaker copyleft license, which allows linking by proprietary software without imposing the same license on the proprietary code, and is commonly used for libraries (for example, `FreeFEM++`, `4C`).

By a *permissive license*, we mean a license that allows derivative works to be redistributed under different terms, including proprietary ones. The most common examples are: **MIT** - minimal conditions, very widely used in the scientific computing community (for example, `MoFEM`); **BSD** (2-clause and 3-clause) - very similar to MIT, used for example by `NumPy`, `SciPy`, `ParaView`, `Trilinos`, and `MFEM`; **Apache 2.0** - permissive, but with an explicit patent protection clause, and is preferred by some larger projects (for example, `Kokkos` - C++ Performance Portability Programming Ecosystem). Sometimes dual licensing is used, for example by `deal.II`, which is distributed under

**Including mini-symposia dedicated to Open Source Software at ECCOMAS and WCCM conferences, enabling participants to present their software without being counted under the "one presentation per participant" rule.*

†Take a look at the wiki page of the `Basilisk` CFD software, the `FreeFEM++` forum, or the GitHub issues page of `MFEM`.

the LGPL and Apache 2.0 licenses.

The choice between copyleft and permissive licenses depends on the goals of the project and the target community of users. However, to maximize adoption, including by industry and commercial software developers, a permissive license is generally preferred. Therefore, in the scientific computing community, **permissive licenses** (MIT, BSD) tend to dominate, as they facilitate integration into larger codebases and commercial simulation tools. They also allow developers to create startups and spin-offs more easily.

Open Data

Software produces data and can rely on data too. For data that can be easily reproduced with open-source software, sharing only the input files may be enough. However, some data might require extensive HPC resources and long computing hours; in this case, sharing such data could be beneficial for the community. Today, there are plenty of solutions for the long-term storage and sharing of datasets. A prominent platform is Zenodo, developed under the European OpenAIRE program and operated by CERN, with a limit of 50 GB per record. Other non-profit initiatives with long-term certified storage include the European EUDAT/B2SHARE, the Dutch 4TU.ResearchData, the French Recherche Data Gouv, and institutional-scale platforms like Harvard Dataverse or DaRUS of the University of Stuttgart; an alternative platform from a commercial vendor is figshare. These platforms provide DOIs for datasets.

The current standard for data sharing is based on *FAIR* principles [6]: *F* - Findable, *A* - Accessible, *I* - Interoperable, and *R* - Reusable. The ease of finding the data strongly depends on careful and rich metadata and on cross-referencing. For example, a dataset should be cited in the scientific paper, and vice versa, the paper should be cited by the dataset*. Accessibility of the data depends on the chosen platform, and Zenodo is a safe choice. Again, to make the data interoperable and reusable, they should preferably rely on open-source software because proprietary binary formats may reduce their utility for the community.

Thus, the key to efficient data sharing is careful and detailed metadata, together with a clear organization of raw datasets, scripts for their analysis, and plotting scripts. A good practice is to provide as much information as possible to make the dataset valuable and usable by others. To help with data curation, one can use Solidipes and the Dissemination of Computational Solid Mechanics DCSM platform. Some examples of well-documented shared data can be found in [8].

Open Scientific Articles

In addition to data, scientific software heavily relies on scientific articles, and vice versa. The core principles of Open Science can guide us in the choice of an appropriate publication platform. In addition to traditional publication venues in our discipline, such as CMAME (Elsevier), Comp Mech (Springer), IJNME (Wiley), and others, there exist community-led Diamond Open Access† journals such as The Journal of Open Source Software, CR Mécanique of the French Academy of Sciences, the Polish Archives of Mechanics operating since 1949, Technische Mechanik, Journal of Theoretical, Computational and Applied Mechanics (JTCAM), and others.

As an example of a community-led journal, consider JTCAM: in addition to ensuring classical peer review with either open or single-blind review, at the reviewer's discretion, the journal publishes Open Reviews along with the paper. These reviews are signed by authors, the editor and by reviewers willing to reveal their identity, while the authors benefit from the Data Editor's help in the *FAIR sharing* of their data [9].

AI in the Loop

When software, articles, and data are truly open‡, they are open not only to humans but also to bots. The corpus of available code is huge: for example, GitHub hosts 630M repositories written mainly in

**This is possible because Zenodo allows users to reserve the DOI before publication of the dataset, so this reference can be inserted in the paper. Conversely, as soon as the paper DOI or another permalink is available, it can be inserted in the dataset metadata before its publication.*

†Diamond OA journals are scholarly journals without commercial incentives, free for authors and readers, and led by the scientific community. The Diamond OA model should not be confused with the Gold OA model, which requires authors to pay an Article Processing Charge (APC), putting the paywall in front of the authors rather than in front of the reader, as in the classical subscription-based model.

‡Commercial publishers, however, aggressively block text and data mining on their platforms even for open-access content, which is a controversial practice.

Python - 17%, Java - 12%, Go - 10%, JavaScript - 10%, C++ - 9%, TypeScript - 7%, and others*. This available corpus of code has contributed to the training of the coding capabilities of modern LLMs†, which have impressively progressed in programming over the last few years. Apart from classical chatbots, programming-oriented tools and platforms are becoming increasingly popular, such as GitHub's Copilot, OpenAI's Codex, Anthropic's Claude Code, Google's Antigravity and Cursor. These tools allow seamless integration of LLM-based development into IDEs, thus raising the pace of development to "supersonic" speed by deploying one or several relatively autonomous agents. Such AI agents can read, search, execute programs, test, and organize the whole repository, work efficiently with version control, review pull requests, document, and deploy. Recently, some companies have claimed that their engineers no longer write a single line of code manually.

No doubt, open-source software has largely contributed to the success of modern AI technologies in programming. AI companies have extracted a lot of value from the open-source movement. But is this rise of LLM-generated code mutually beneficial for the open-source community? First, AI technology has practically removed the entry barrier for coding and contributing: anyone able to formulate a reasonable request in plain language and write a minimal technical task, can ask an LLM to construct an efficient program in almost any programming language and guide them through its deployment. The downside of this barrier removal is that open-source repositories are now flooded by the so-called "AI slop" - poor-quality AI-generated contributions that make it much harder for volunteer maintainers to deal with them. Nevertheless, modern AI tools, when used smartly (with a good understanding of the code, its architecture, and the particularities of its development), can produce valuable high-quality code and automate time-consuming routine procedures, thus potentially contributing positively to the growth and prosperity of open-source software. This is an actively debated topic, and there are as yet no simple solutions or well-established practices. What can be said for sure is that AI technology strongly impacts the open-source software community.

Another upside or downside of capable AI code-generation tools is their powerful "clean room"‡ capacity - the full rewriting of existing code or software, including closed-source software. A recent story concerns the LGPL open-source chardet package, a character-encoding detection tool, which was "clean-roomed" to make it more accurate and faster; its new version was released under a more permissive MIT license. This situation raised a rich technical and ethical discussion [10]. An even more recent "clean room" precedent concerns about 500K lines across ≈ 2K accidentally leaked Claude Code orchestration files. Sigrid Jin managed AI agents that could rewrite it all within 2(!) hours

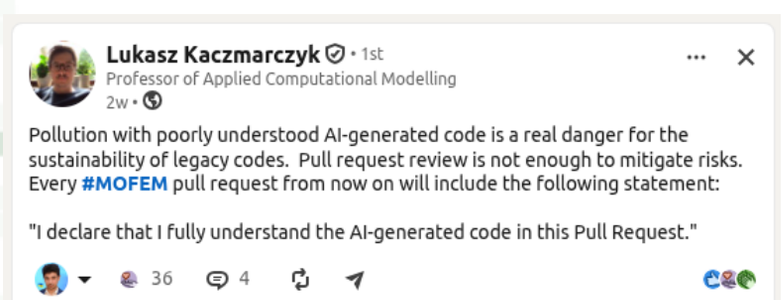

Figure 1. Lead developer of the MoFEM open-source software on the risks of "poorly understood AI-generated code".

**Data from madnight.github.io/githut 2024.*

†LLM - Large Language Model, such as OpenAI's ChatGPT, Anthropic's Claude, Mistral's chat, Meta's Llama, Google's Gemini, xAI's Grok, Alibaba's Qwen, DeepSeek, and many others.

and released it as open-source software, although apparently without a clear license - Claw Code [11].

Here I quote [12]: *"When the cost of generating code goes down that much, and we can re-implement it from test suites alone, what does that mean for the future of software? Will we see a lot of software re-emerging under more permissive licenses? Will we see a lot of proprietary software re-emerging as open source? Will we see a lot of software re-emerging as proprietary?"*

This new AI era might force developers to adjust their roles, switching from true builders to architects and managers. The new value then shifts to knowing what to build and why, and also to knowing how to efficiently orchestrate agentic AI to build what is needed; this higher-level orchestration requires a clear vision and mastery of new tool sets. Boris Cherny - Claude Code's lead developer - compares this shift for programmers to the one experienced by monks copying manuscripts by hand when the printing press was invented.

Another facet of this complex AI impact on software is that some people are starting to speak about the end of the one-size-fits-all software era, with a shift toward much narrower, application-specific tools, potentially relying on reliable foundational software components such as solvers, meshers, and post-processing tools.

Concluding Remarks

In conclusion, we can only express our gratitude to all contributors to open-source software, who make our lives so much easier by lowering entry barriers, providing efficient tools, and enabling us to expand the frontiers of research and applications.

References

[1] Avelino, G., Valente, M.T. and Hora, A. (2017). What is the Truck Factor of popular GitHub applications? A first assessment (No. e1233v3). PeerJ Preprints. 10.7287/peerj.preprints.1233v3

[2] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey (2006). libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. Engineering with Computers, 22(3—4):237—254. 10.1007/s00366-006-0049-3

[3] Kaczmarczyk et al., (2020). MoFEM: An open source, parallel finite element library. Journal of Open Source Software, 5(45), 1441, 10.21105/joss.01441

[4] Software Heritage is a non-profit organization which provides a service for archiving and referencing historical and contemporary software with a focus on human readable source code. Founders Roberto Di Cosmo, Stefano Zacchiroli

[5] Mark Wilkinson et al (2016). The FAIR Guiding Principles for scientific data management and stewardship. Sci Data 3, 160018. 10.1038/sdata.2016.18

[6] GitHub resource Choose an open source license choosealicense.com

[7] SPDX License List spdx.org/licenses

[8] Zenodo JTCAM community zenodo.org/communities/jtcam.

[9] Simon Willison's blog "Can coding agents relicense open source through a “clean room” implementation of code?" (March 5, 2026) simonwillison.net/2026/Mar/5/chardet/

[10] Mehul Gupta (April 6, 2026). *Claw Code Killed Claude Code?* Article on Medium.com

[11] Armin Ronacher (March 5, 2026) *AI And The Ship of Theseus https://lucumr.pocoo.org/2026/3/5/theseus/*

VLADISLAV A. YASTREBOV
RESEARCH SCIENTIST AT
CNRS, MINES PARIS - PSL, FRANCE
VLADISLAV.YASTREBOV@MINESPARIS.PSL.EU