Explaining software and computational methods

Published June 25, 2025

Citation

Hinsen, K. (2025, June 25). Explaining software and computational methods. *Konrad Hinsen's Blog*. https://doi.org/10.59350/g0hyh-qzx40

How can we document software and computational analyses in such a way that others can convince themselves of their validity, and build on them for their own work? The question has been around for many years, and a number of attempts have been made to provide partial answers. This post provides a brief review and describes my own tentative answer, inviting you to play with it.

Explainable AI is a hot topic today. Many people, in particular in research, are worried about the rapid adoption of AI-based techniques that provide answers that cannot be questioned, because nobody is able to figure out how the AI's output is derived from its huge inputs. However, most researchers seem to overlook that we have exactly the same problem with traditional software. Even if your full software stack is Open Source, its size and complexity make it unrealistic for anyone to compile a list of all the models, hypotheses, and assumptions that its various authors baked into it. In thirty years of practicing computational science, I have seen quite a few unreasonable assumptions in source code, sometimes even pointed out in comments, but out of sight of the vast majority of users who run the code but never look at it. This is why I advocate that we should peer-review research software.

However, reviewing software source code is envisageable only of it is actually written to be understandable by fellow researchers, who are mostly not software professionals. We won't get reviewable code by simply adopting software engineering methods from the software industry, as we have mostly done over the last decades. Fortunately, people have started thinking about explainable and understandable software more than 40 years ago.

Literate programming

The well-known computer science textbook Structure and Interpretation of Computer Programs (see here for a free online version) states one of its goals in the preface to the first edition, which was published in 1984:

> *First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute.*

One can argue that the authors achieved that goal, because the code they present is indeed very readable. However, it consists of textbook examples, and its target audience is computer science students. Writing a large software system in such a way that its *users* could understand it is a much more challenging task, which the book does not address.

That same year, Donald Knuth introduced the idea of literate programming. He summed up the motivation in one sentence:

> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*

Knuth developed the WEB programming system to support this new style of writing software, and applied it himself, most notably to his well-known and widely used programs TeX and Metafont, but also many smaller and more specialized ones. TeX and Metafont are not particularly large software systems by today's standards, but they are not textbook examples either. One feature of literate programming as advocated by Knuth is writing code and narrative together, in a single file, and use tools to extract these two aspects for further processing. Writing literate code imposes a new writing style for code, and that is an important part of the exercise.

Many literate programming tools have been created after WEB, but few of them were ever adopted by anyone else than their authors. Perhaps the most widely used one today is the Babel package that is part of Org Mode for Emacs. It offers many features that go beyond Knuth's original concept, so it is hard to say if its users really apply literate programming as intended by Knuth.

A major limitation of literate programming is the restriction to static artifacts, i.e. program source code and documentation. The code is never executed and need not even be executable. This limits the scope of exploration. For any but the most trivial software, developing a good understanding of its operation requires running examples on various inputs and inspecting the outputs, in addition to reading the code. Traditional literate programming tools do not prevent this in any way, but they do not support it either.

Computational notebooks

Just a few years after literate programming, and possibly inspired by it, the first notebook interfaces appeared in two commercial software packages: MathCAD and Mathematica. Many similar interfaces followed, out of which the most popular one today is Jupyter, which goes back to 2011 (then under the name "IPython notebook"). While many people consider literate programming and notebook interfaces roughly the same, there are few important differences. Most of all, literate programming is technology for *publishing* documented software, whereas notebooks are primarily interactive interfaces for *performing* explorative computations. They are in fact an extension of the much older read-eval-print loop, to which they add documentation, preservation of outputs, and usually data visualization. Another important difference is that literate programming can document arbitrary units of software but no input or result data, whereas notebooks document a specific computation, i.e. a linear chain of expressions or commands, including their inputs, outputs, and intermediate results.

The restriction to a linear narrative embedding a linear sequence of code snippets is the main limitation of notebooks. They cannot explain complex software assemblies that would require multiple indepenendent narratives. And they don't given access to the library code that is called from the notebook's code snippets. If you see `print(median(data))` in a notebook written by your former student who you remember for always confusuing mean and median in your statistics class, you would probably like to click on `median` and see the code behind it, but that's not part of a typical notebook interface.

As the name suggests, notebooks were inspired by the lab notebooks that experimentalists use to keep track of what they did and what they observed, and they fulfill a similar role for doing computer-aided research. Neither kind of notebook is intended for publication in the traditional academic sense, although both can be shared publicly in the context of open-notebook science. Computational notebooks are sometimes advertised as publishing technology, and even as the successor of the traditional scientific article (see here for example). And there are publishing systems that take notebooks as inputs, e.g. Quarto. The term "notebook" is probably overused today, meaning different things to different people. What matters for my purposes is the distinction between a log of a research activity and a publication written for explaining research to someone else. I am interested in the latter, which is not well supported by most of today's notebook variants.

Explorable explanations

A different take on the question of how to make computational techniques understandable is Bret Victor's essay "Explorable Explanations". It focuses on interactive exploration of a computational model by the reader of a pedagogical narrative. Technically, it's a narrative with embedded interactive programs. The code of the latter is usually hidden, it's not the topic of the explanation. Many notebook interfaces nowadays also provide interactive elements for readers, e.g. Jupyter's widgets, which work the same way: they provide functionality, but their implementation is opaque to the user, unlike the code in the notebook itself.

There are in general three categories of code in computational explanations:

1. Code that you want the reader to look at, and maybe run.
2. Code that you want the reader to run, and maybe look at.
3. Lower-level support code.

Notebooks focus on the first kind, explorable explanations focus on the second kind. Neither one makes an effort to let you access the third kind.

Explainable software systems

Finally, there has been some work in software engineering on making software systems as a whole explainable to non-developers (see here for an example). This work is about complex software systems, which cannot be described by any single narrative. An explainable software system combines multiple techniques and technologies to achieve its goal: narratives with embedded computations, like a notebook, but also interactive elements, like an explorable explanation, plus views on data or on processes that are automatically generated from a working system, rather than hand-crafted separately as part of documentation. This is the most comprehensive approach. It also removes the distinction between author and reader. Everyone experiences the system through the same interface, though different people typically focus on different parts of it.

After a few years of experience with Glamorous Toolkit, which is the software platform underlying the work cited above, I am convinced that this is a good approach for making computational work explainable (for the author) and explorable (for everyone else). It removes the main limitation of notebooks, where explanation is restricted to a single top-level narrative. It also removes the opacity of the code from explorable explanations, because the implementation of interactive elements is easily accessible. From a bird's eyes view, you have a large codebase and any number of narratives that can refer to code, transclude code, and embed the results of running code, in addition to embedding other media, such as video clips. That's a very expressive medium for communcating computational ideas. It can be seen as a variant of hypermedia that focuses on code. Crafting high-quality explanations is still a challenging task, but then, that is true for writing high-quality pedagogical material in traditional media (such as books) as well.

However, I also came to the conclusion that Glamorous Toolkit is not a good medium for communicating computational science. It is great for communication within an organization, among a relative small number of people who know each other and work on a common project at the same time. We have this kind of collaboration in research, of course, but we also have the long-term loose-knit open and anonymous collaboration that happens in a scientific community, involving hundreds of people, spread all over the globe, working on a challenging topic for several decades. An important communication concept in this mode of collaboration is the *publication*, which traditionally is a narrative published in a journal and then available for everyone to read and build on. In the digital era, this requires an update, which so far has not really happened. Publications moved from printed pages to PDF files, but we still struggle to integrate digital datasets, software, and interactive exploration tools.

Glamorous Toolkit is a comprehensive solution for all that's missing from the traditional article, but it lacks the concept of a publication, i.e. a version of record that everyone can consult, and that everyone can refer to, knowing that whoever follows the reference will access exactly the same information. Publication adds two requirements to an explainable software system. One is durability: a publication must remain usable and present itself identically over a few decades, which is the time scale of evolution of most scientific disciplines. The second one is an asymmetry between authors and explorers (formerly called readers). Researchers consult at least 100 times more publications than they co-author. Exploring a publication must be easy, with no technical obstacles such as software installation. You cannot expect the same level of engagement from an explorer of a publication as from a collaborator in a team. You can, on the other hand, expect authors to invest more effort into polishing their work for the benefit of many anonymous explorers.

HyperDoc

In today's computing environments, the only technical choice for computational publications is making them available on a Web server, for access with a standard Web browser, because a Web browser is the only piece of software that you can reasonably expect every explorer to have available. That leaves a lot of options for designing and implementing such a publication

system. Exploring them, and developing experience with authoring computational publications in such a setting, is the goal of my latest project: HyperDoc. The link points to a server that runs a demo with a few small publications: two software packages whose documentation is based on HyperDoc, and two computational publications, one of which (a tiny data paper) being cited and reused by the other one (a simple data analysis).

The user interface is heavily inspired by Glamorous Toolkit. The browser shows a lineup of panes, each of which represents an object in memory, for which there can be any number of views that are accessible via the tabs. Hypertext pages and code pages are just particular types of objects, with primary views that render them for reading. Visualizations and interactive elements are implemented as views on classes of data objects. By holding the alt key when clicking on the tab for a view, you get to see the source code rather than the output of the view. That's how all code is instantly inspectable without cluttering the interface.

The basic publishing unit is called a HyperDoc. It has *text pages* (written in HTML or Markdown) and *code pages*, which are plain source code files. Code pages are rendered with some useful decoration, most notably links from function and class names to the source code that defines these items, but also links to text pages. Text pages can contain links to other pages, but also links to the value of a computed expression, which is how you link to e.g. visualizations. Instead of linking, you can also embed (or, if you prefer fancier jargon, transclude) other views.

The source code for a HyperDoc typically resides in a version-controlled repository. See here for an example. The code is written in Common Lisp (more on that later), and the whole repository is a standard Common Lisp code repository that should look familiar to any Common Lisp programmer. The only particularity is that it also contains hypertext pages, and needs to conform to a few conventions. Having a HyperDoc represented as a source code repository means that installation, distribution, and archiving are taken care of by existing infrastructure for software. Software citation mechanisms such as CodeMeta are applicable as well: the "Authors" view on a HyperDoc shows author information from a CodeMeta file.

One important element for exploration is unfortunately missing from the online demo: the "Playground" view that permits explorers to run arbitrary code in the context of a data object, e.g. to inspect data by running code snippets. Allowing the execution of arbitrary code on a server is a major security risk, which is why the playground is disabled. You can, however, install everything on your own computer (e.g. via Guix, or Quicklisp plus Ultralisp) and then use the playground. A safe public playground requires sandboxing, which isn't implemented yet.

Common Lisp

As I already mentioned, HyperDoc as well as all the demo code is written in Common Lisp. It is one of the very few programming languages that fulfills my two primary criteria:

1. Strong support for code introspection, making it possible e.g. to access the source code of a function or a class.
2. A stable language definition and library ecosystem, for durability.

Criterion 1 is about simplicity of implementation for HyperDoc. It is possible to implement HyperDoc in or for static languages such as Fortran or C++, but the effort would be much higher. For a single-person exploratory project, it looks prohibitive. If HyperDoc turns out to be attractive to enough people, more elaborate implementations become doable.

Criterion 2 is about suitability for scientific publication. Technically, HyperDoc would be almost as easy to implement in Python as in Lisp. But Python and its scientific library ecosystem have undergone too many breaking changes in recent years to be able to support a long-term publication infrastructure. You can always *run* old Python code in a snapshotted environment. That's how reproducibility works. But being a good citizen of scientific publishing also requires that people can *build on* code and data from a ten-year-old publication. That requires both old and new HyperDocs to run on top of a shared collection of libraries. I gave up on the Python edition of ActivePapers for a reason. Otherwise, HyperDoc would have become a user interface layer for ActivePapers.

Next steps

The HyperDoc demo contains four very small and simple HyperDocs. The challenge I have set myself for the next phase of the project is to implement examples that people would actually want to explore, i.e. the HyperDoc equivalents of journal articles, monographs, or textbooks. This might include integrating Leibniz, my equally experimental Digital Scientific Notation.

If you want to help with this project, the most useful contribution right now would be feedback on the four HyperDocs in the online demo. I am well aware of one difficulty: few people reading this post are likely to be familiar with the Common Lisp language. I made the choice *not* to explain Common Lisp in these HyperDocs. Just like a journal article tacitly supposes prior knowledge of English and math notation, my HyperDocs tacitly assume some knowledge of its computational notation. Otherwise, they would be more of a Common Lisp tutorial than explanations of computational data analyses. My hope is that the careful choice of names makes the code at least superficially comprehensible. Again, that's something I'd appreciate feedback on.

The best place for feedback are the issue trackers of the code repositories: HyperDoc itself and the three other demo HyperDocs:

- the Tabular data library
- the data paper "Acute respiratory infections in France"
- the analysis paper "Influenza epidemics in France"