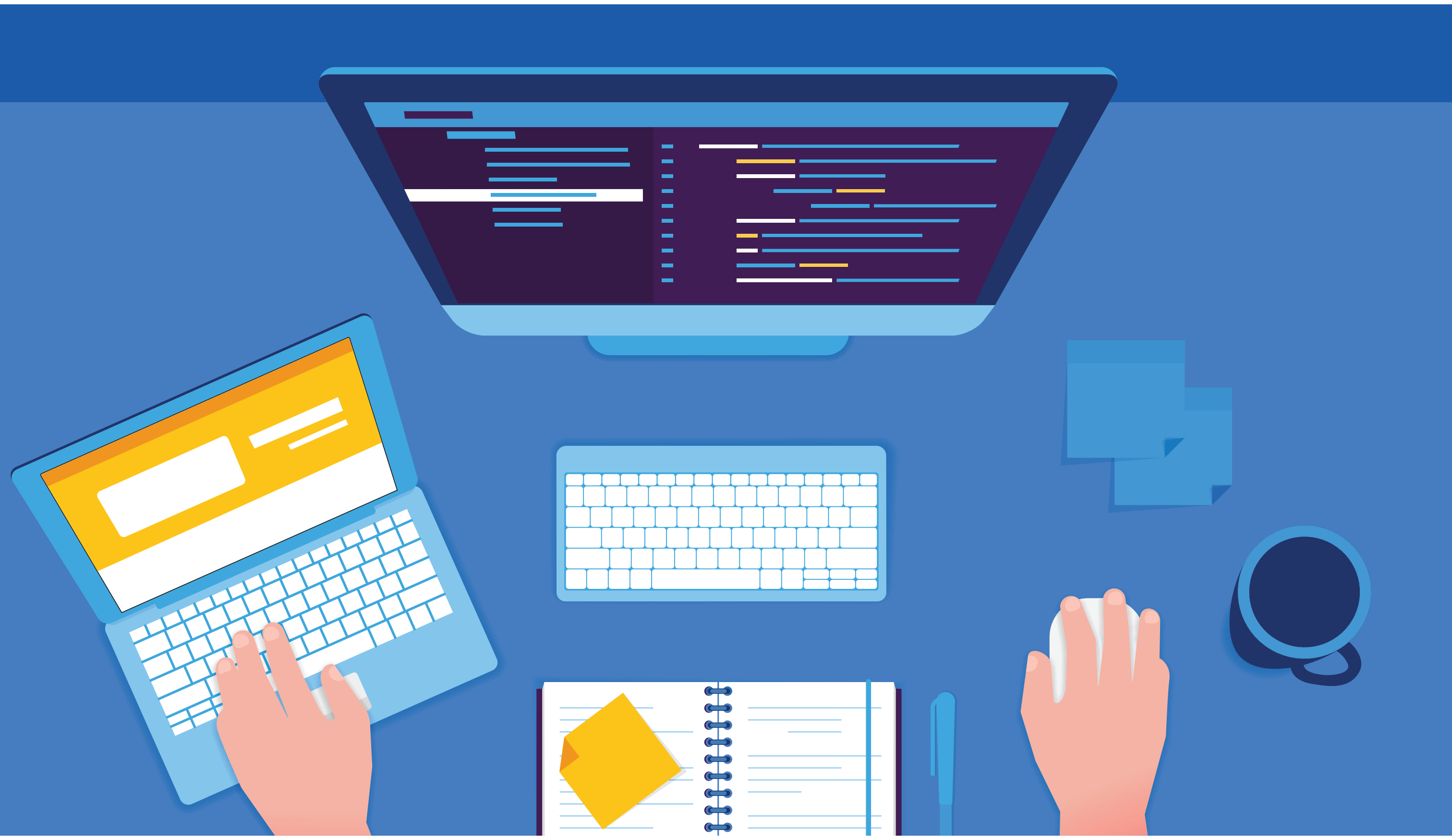
Editors: Konrad Hinsen, konrad.hinsen@cnrs.fr | Matthew Turk, matthewturk@gmail.com



The Roles of Code in Computational Science

Konrad Hinsen | Centre de Biophysique Moléculaire in Orléans

It's a safe bet that most readers of *CiSE* have written some code during their professional life. Many of us write code regularly as part of our scientific activity, perhaps even as a full-time job. But even though we write—and use—more and more code, we rarely think about the roles that this code will have in our research, in our publications, and ultimately in the scientific record.

In this article, I outline some frequent roles of code in computational science. These roles aren't exclusive; in fact, it's common for a piece of code to have several roles, at the same time or as an evolution over time. Thinking about these roles, ideally before starting to write the code, is a good habit to develop.

The most obvious role of code is its use as a tool for doing computations. In fact, it's this role that sets code apart from other types of information. In its tool role, code is evaluated using criteria such as suitability for a task, robustness, flexibility, performance, ease of use, and so on. Because this role should be familiar to *CiSE* readers, I won't say much more about it and will concentrate on the other, less obvious roles.

Code as a Scientific Notation

One important family of roles can be summarized as executable expressions of scientific information. In these roles, code can be considered a form of scientific notation. Criteria such as clarity of expression and exposition are therefore important, just like for any other form of scientific communication. However, executability matters as well, otherwise we would stick to more traditional notations such as plain language or mathematical formulas. Being able to execute the code is a proof of completeness. In an article, you can forget to define a quantity without anybody noticing for a long time. In a computer program, such an omission yields an error message. Executability is also a proof of absence of certain types of mistakes: the mistakes that a compiler can identify, and the mistakes that cause a program to fail during its execution.

The very first step in a research project is exploration: doing experiments and computations to better understand some phenomenon. In terms of scientific writing, this means lab notebooks and back-of-the-envelope calculations, meant for personal use or for sharing with close collaborators.

I call this a family of roles because very different kinds of scientific information can be expressed by code, and each kind requires its specific style, both for the code itself and for the documentation that accompanies it. In this respect, code isn't all that different from traditional scientific writing: a journal article and a textbook for students have different goals and are therefore written in very different styles. This analogy between code and scientific writing turns out to be useful for identifying different types of executable scientific knowledge.

The very first step in a research project is exploration: doing experiments and computations to better understand some phenomenon. In terms of scientific writing, this means lab notebooks and back-of-the-envelope calculations, meant for personal use or for sharing with close collaborators. At this stage, computations are performed interactively or by writing short scripts. But even interactively entered commands are code. Just like experimentalists keep lab notebooks, computational scientists should keep a detailed log of everything they type into their computers. Like all notes taken during exploration, this code must be cleaned up before publication.

Next comes publication. A paper in a scientific journal reports on a scientific study, explaining the context and motivation, the exact system being studied, and the methods that were applied. Results are then presented and discussed. The computational equivalent is a detailed record of all computations that were performed, including the software, the input data, and the results, presented in an understandable way.

Note that this detailed record isn't just code but code in its scientific context. This is true in general for the role of code as scientific notation. In fact, it's true as well for other scientific notations. A mathematical formula—or even a sentence in plain English—doesn't convey much meaning on its own. It acquires meaning only in the context of other formulas and sentences. Similarly, the code that was run in a computational research project makes sense only in the context of an overall description of that project.

This insight has led to the development of computational notebooks, pioneered by Mathematica (www.wolfram.com/mathematica) and more recently implemented for several programming languages by the Jupyter project (http://jupyter.org). Notebooks have become popular because they make it possible to combine the computational protocol used in a study with a scientific narrative into a single document. The reader of such a notebook not only can read the narrative but also can run the code, inspect intermediate results, and change parameters to see their impact on results. This ability to work with the code makes it easier to understand the scientific narrative, and vice versa.

Unfortunately, the term "notebook" has created a lot of confusion with lab notebooks, whose role is quite different, as I explained above. To make it worse, computational notebooks are also convenient tools for interactive work during the exploration phase. Finally, a published notebook is often the starting point for exploration of somebody else's work. From a tool perspective, computational notebooks have eliminated the borderline between exploration and publication. But in terms of scientific communication, the distinction still matters. Uploading notebooks containing interactive explorations to GitHub is definitely not sufficient for publishing code.

If journal articles were the only type of scientific document, we would have drowned in the ocean of millions of individual scientific studies long ago. What makes it possible to keep an overview of progress in a specific field is condensed summaries, which take the form of review articles. They provide a snapshot of the state of the art in a field that still undergoes rapid development. In the world of code, the equivalent of a review article is a library or program package that implements state-of-the-art algorithms with the goal of making them easily accessible to other researchers. Like the readers of a review article, the users of such a library are primarily experienced practitioners who can be expected to know the ideas behind the algorithms. However, precise and up-to-date technical documentation is essential: users must be able to figure out easily what exactly the code does—in particular, what its limitations are.

Beyond review articles, there's a continuous consolidation process of scientific information whose final stage is textbook knowledge. Knowledge presented in textbooks is information that's well known and well understood by research practitioners from many specialty fields. The code equivalent of textbook knowledge is a library or program package that implements well-known and widely used algorithms in such a way that they can safely be used as black-box ingredients in research code. This requires particular attention to robustness. The code should behave reasonably even when used in an unforeseen way. Documentation must be written with nonexpert users in mind and should address both technical conditions and appropriate scientific contexts for the use of the code.

Unfortunately, we haven't yet found a good way to present libraries and program packages to human readers in their scientific context. Ideally, readers should see the code embedded in a discussion of the methods implemented. This discussion should be illustrated by example applications, and test cases should point out assumptions about the inputs, the handling of edge cases, and other subtleties. Literate programming[1] goes a long way toward this goal. Its basic premise is that software should be written as an essay for human readers, with markup that permits extracting pure code files for execution.

Note that literate programming is more complex than the computational notebooks I mentioned above. A notebook describes a linear sequence of computational steps with inputs and outputs alongside a scientific narrative. In contrast, algorithms and computational methods aren't linear. They don't have fixed inputs either—rather, they're designed to work with a wide range of inputs. This is perhaps one reason why literate programming hasn't been adopted yet by computational scientists. Another reason is the unsatisfactory state of support tools, which are incomplete and incompatible with each other. But the main reason is that the scientific community has only recently acknowledged the importance of the scientific notation role played by code.

The notation roles that I've discussed up to here deal with scientific models and methods, which have occupied the central stage of research for a few centuries. Today, data plays an ever more important role, and datasets are increasingly recognized as scientific publications in their own right. Most published datasets are raw or processed observational data, but it can also be of interest to publish simulation output if regenerating it is particularly costly or difficult. Whatever the nature of the data, every electronic dataset has been created with the help of software, and that software's source code contains valuable information for interpreting the data. Ideally, the syntax and semantics of datasets should be well defined and documented (a topic I've written about before[2]), making it less important to have access to the code that generated the published files. However, just as descriptions of algorithms in plain language tend to be incomplete, descriptions of data models aren't always sufficient in practice. In the case of processed data, a detailed record of the processing steps in the form of the code that was actually run is also very helpful for interpretation. This role of code overlaps with the role of documenting a scientific study. The main difference is a focus on the process in one case compared to a focus on the results in the other. Another difference is that code that documents datasets should ideally be attached to the datasets as metadata and not require fetching another document from a different place.

Code as the Shared Asset of a Scientific Community

A final role of code that deserves some discussion is its role in the structuring of scientific communities. Major program packages or libraries, but also more widely used infrastructure tools such as programming languages, have often been the nucleus for the formation of sizable user and developer communities, which, in the case of programming languages and infrastructure libraries, transcend the traditional scientific disciplines. The social impact of code has been much larger than for any other technical aspect of doing research.

I see the main reason for this in the enormous complexity of software. Keep in mind that when you run a 10-line Python script on your computer, you're really deploying a stack of software items that includes the Python interpreter, the libraries installed along with it, and an operating system such as Linux. Moreover, getting this software stack working on your computer has required other software that you may no longer see, such as a C compiler. The development, maintenance, and deployment of this enormous amount of code can't be handled by any individual or even a team of people working in close contact.

Consider a computational scientist working on the development of domain-specific code. This task requires a good understanding of the level below in the stack—a programming language, various libraries, and various tools such as compilers. Acquiring a useful level of competence with all of this is an important effort, so few people can

In practice, this means that the choice of a program package or a programming language implies the choice of a community one will interact with and maybe even become a part of.

become proficient in more than one or two such environments. The programming environment thus becomes an important aspect of one's work, and discussing this work is much easier with people who share the same background. Collaborating on code development imposes even stricter compatibility: all developers must be proficient using almost identical environments. In comparison, traditional scientific instruments are simple. A scientist familiar with the principles of, for example, electron microscopy can rapidly learn to use any particular electron microscope and has no difficulty in talking to or collaborating with colleagues who use different models. Likewise, a theoretician can easily read articles written by scientists from a different school that favors different mathematical notations or uses different definitions.

In practice, this means that the choice of a program package or a programming language implies the choice of a community one will interact with and maybe even become a part of. But every community also has shared values and attitudes that aren't directly related to its shared technologies. One community might be conservative, valuing long-term compatibility over progress, whereas another might like its code on the bleeding edge. Moving even further away from technical aspects, communities can be more or less welcoming to newcomers and more or less open to new ideas. At the extreme, communities can have sexist, racist, or elitist attitudes that can be serious obstacles to productive research work.

The social roles of code are likely to gain in importance as code itself is more and more recognized as important in research. At this time, publishing code has become common though not yet universal. But unlike journal articles, code isn't reviewed in the publication process. One obstacle to independent code reviewing is that a potential reviewer must have both the right scientific domain knowledge and solid technical competence in the development technologies employed by the code authors. If reviewing code becomes common, this will likely create social pressure toward standardization of such technologies inside each scientific domain.

Tensions between the Roles

The different roles that a piece of code has aren't independent. On the contrary, they can easily be in conflict, requiring scientific software developers to find a workable compromise. This is why I recommend considering the roles of your code before starting to write it. Compromises tend to be better when they're the result of conscious decisions rather than accidents of development.

An often-discussed tension is the one between performance in the tool role and clarity in the notation role. Performance optimization tends to make code less understandable. In the extreme, for example, when parallelizing code at a low level using MPI, it can become very hard to figure out what the optimized code does exactly. It's possible that better notations—that is, new programming languages designed from the start to act like scientific notations—can alleviate this tension, but for the moment, this is just wishful thinking.

Another important conflict exists between the generality that's desirable for the tool and social roles on one hand and the requirement for simplicity in the notation role on the other. Tools applicable to a wide range of problems are advantageous for their users, who can get away without learning too many tools. They're also advantageous for their developer communities because these communities can recruit members from a wider base and thus share the development work among more people. However, general tools lead to larger and more complex code bases that are usually more difficult to understand than simple code written specifically to solve one problem.

A related tension stems from the reliance on other code, in particular, libraries, something that we've discussed in this department before as well.[3] From the viewpoint of the tool role, using existing library code, assuming it's of good quality, is usually an advantage. In the social role, code with library dependencies creates asymmetric links between two communities, which has good and bad sides. From the notation viewpoint, dependencies on libraries implementing well-known techniques is an advantage, as readers are likely to be already familiar with them. But dependencies on new or little-known libraries make understanding the code more difficult for the average reader because there's more code to read—in fact, this is a special case of the generality dilemma.

Interactions between Different Categories of Code

Another aspect to consider in writing code that fulfills its intended roles well is that no piece of scientific code exists in isolation. The computational protocol of a specific study relies on program packages and libraries of varying degrees of maturity, and each program package and library adds dependencies of its own.

From the tool perspective, the computational protocol has almost no value because its work is done. It might serve as an inspiration for the development of other tools, but it's unlikely to be reused in its entirety. Consequently, there's no point in doing any maintenance. In contrast, the domain-specific or general libraries it depends on will be used by larger communities for a longer time. They'll be maintained and extended, but that also means that they undergo permanent change.

This poses a problem from the notation perspective, which requires that a published computational protocol should remain reproducible for as long as the study itself remains of scientific interest, which can easily be several decades. This is the core issue of the reproducibility problem, which has come up repeatedly in *CiSE*, in particular in two theme issues (published in January/February 2009 and July/August 2012). In the past, the tool perspective has dominated scientific software, with the result that individual computational protocols are effectively built on quicksand foundations consisting of tools that change through continuous improvement. The increasing emphasis on reproducibility—and the ongoing tendency to publish scientific software—is thus likely to create pressure on stability at least in widely used libraries and program packages.

A final aspect to consider is the evolution of roles in time. From the notation perspective, it's desirable to write new code as the status of a computational method advances from innovative through established to well known. The requirements for these respective roles are too different to be assumed by a single code base. The analogy with scientific writing is useful again: it would be ludicrous to compose a review article from pasted-together sections of original research articles. But scientific code is still rarely written with its notation role in mind. From the tool viewpoint, reusing working code may well seem preferable, and that's also the way of least immediate effort.

Readers waiting for a final coherent message from this article will be disappointed: I can't offer any simple conclusion. The only advice I can offer is to take all the issues I've outlined into account when starting a coding project. Consider why you're writing the code right now and for how long you expect to use it. Consider who else might use the code in the foreseeable future and who might collaborate with you on its future development. Do you plan to publish it? This might actually not be a choice for much longer, so better prepare to answer "yes."

For any code meant to be published, the notation roles should be explored carefully. As with any form of scientific communication, you have to write for your audience. For a code base, that audience consists of the users of the code but also consists of the audience that its users are addressing with their own work. If you want me to use your code, you have to make it easy for me to explain to my readers what your code does.

The social role of code is particularly important to consider when building a community around a software package is your explicit goal. But even if you write small and specialized software that's unlikely to ever become widely used, it's worth thinking about how your code will be judged by the user community of your development and infrastructure tools, if only because in the not-too-remote future this judgment might become part of the scientific reviewing process. Adopting the best practices of this community is the equivalent of making an effort to write your papers in good English—something that I haven't heard anyone argue against yet. ■

References

1. D.E. Knuth, "Literate Programming," *Computer J.*, vol. 27, no. 2, 1984, pp. 97–111.
2. K. Hinsen, "Caring for Your Data," *Computing in Science & Eng.*, vol. 14, no. 6, 2012, pp. 70–74.
3. M. Turk, "Vertical Integration," *Computing in Science & Eng.*, vol. 17, no. 1, 2015, pp. 64–66.

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans and at the Synchrotron Soleil in Saint Aubin. His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University. Contact him at konrad.hinsen@cnrs.fr.