On the role of computer languages in scientific computing

Dorian Leroy, June Sallou, Johann Bourcier, Benoit Combemale

On the role of computer languages in scientific computing

Dorian Leroy
CEA, DAM, DIF, F-91297 Arpajon, France

June Sallou
Univ Rennes, Géosciences Rennes, CNRS, Inria, IRISA - UMR 6074, F-35000 Rennes, France

Johann Bourcier
Université de Rennes 1, Inria, CNRS, IRISA, France

Benoit Combemale
Université de Rennes 1, Inria, CNRS, IRISA, France

***Abstract*—Scientific software are complex software systems. Their engineering involves various stakeholders using specific computer languages for defining artifacts at different abstraction levels and for different purposes. In this paper, we review the overall process leading to the development of scientific software, and discuss the role of computer languages in the definition of the different artifacts. We then provide guidelines to make informed decisions when the time comes to choose the computer languages to use when developing scientific software.**

On Scientific Computing

Scientific computing is a cross-cutting field, but its heart and soul lie in the development of mathematical models to understand physical systems through their simulations. Those models can be numerical (e.g., systems of differential equations), non-numerical (e.g., agent-based models) or based on analytics (e.g., machine learning models), and capture the behavior of the modeled system. Numerical models can be further refined as continuous or discrete. Simulations of mathematical models correspond to the execution of the computer programs containing these models, the so-called "simulation codes". In this paper, we refer to the subsuming concept of *scientific software*, which we define as software dedicated to scientific computing and simulation. The development of these scientific software includes both software engineering and scientific computing concerns. Mathematical models and scientific software are therefore tightly intertwined throughout their life cycles. The tools and methods used for their development (e.g., computer languages) have an impact on the definition of both, as well as on the engineering principles required to ensure the development of reliable scientific software.

When the time comes to implement a new model – and thus new simulation software – scientists and engineers are faced with decisive choices such as what computer language(s) to use (e.g., MATLAB, Mathematica, Fortran, Python, C++, or even an in-house domain-specific language). This choice has important consequences on the expressiveness available to implement the model and the corresponding simulation code, but also in terms of software engineering practices to develop reliable and efficient scientific software.

The more general-purpose the language is – with low-level, computing-related, system abstractions – the more flexibility and performance it may provide, but also the more rigorous engineering principles and Validation & Verification (V&V) activities it will require from the language user to develop a reliable piece of scientific software.

However, most scientists and engineers are not trained in software engineering and are therefore not aware of its best practices beyond programming (*e.g.,* version control management, component reuse, unit testing, continuous integration) [1], which has led to initiatives addressing this problem, such as the Research Software Engineering movement [2]. Since the final goal is to build and apply the model encoded in the simulation code, the code itself is merely a means to that end. Final stakeholders (*e.g.,* citizens, policy and decision makers, research institutions, system users) may even be ignorant of the importance of code for science and engineering.

In this paper, we explore the overall scientific software development process, we provide an integrated view of the scientific computing and software engineering activities, artifacts and roles, and we discuss the trade-offs on the computer languages at hand to help scientists and engineers make informed decisions.

Scientific Computing: Computer Languages to the Rescue

The implementation of scientific software is the result of the successive refinement of different artifacts, starting with *observations* to elaborate the *mathematical model* thanks to *theories*, then, applying *discretization methods* to obtain a *numerical scheme*, to finally end with the implementation of the *scientific software* (cf. Figure 1).

Thus, the design of scientific software based on mathematical models requires the involvement and cooperation of various stakeholders, ranging from scientists and engineers to experts in numerical analysis or software engineering. These stakeholders play one of three roles (according to the development context, one person might endorse more than one role): *scientists* as domain experts, *numerical analysts* as experts in the discretization of a continuous phenomenon, and *software engineers* as experts of software development to deliver the expected services. Each role is in charge of the elaboration of one of the artifacts: scientists define the mathematical model, numerical analysts refine it into a numerical scheme, and the software engineer implement the software.

Computer languages enable the different stakeholders to perform their activities at the corresponding level of abstraction. We can thus classify computer languages according to their level of abstraction and the support they provide to stakeholders.

Languages to define the mathematical model

Defining a *mathematical model* and deriving the corresponding *scientific software* can be done by scientists using languages such as Mathematica (https://www.wolfram.com/mathematica or more specifically the Wolfram language https://www.wolfram.com/language), or MATLAB (https://matlab.mathworks.com). Such languages provide continuous mathematical constructs (*e.g.,* algebraic computation and differential blocks in MATLAB's block diagrams) allowing scientists to directly define their mathematical models with the language. The language infrastructure is then able to automatically discretize the mathematical models defined with the language, possibly in a configurable way, and to derive the corresponding scientific software.

Languages to specify the numerical scheme

Alternatively, some languages allow deriving scientific software directly from a numerical scheme. Languages dedicated to the definition of *numerical schemes* (or with the right abstractions to do so), such as Julia (https://julialang.org/), R (https://www.r-project.org/), or NabLab (https://cea-hpc.github.io/NabLab/), allow to automatically derive the corresponding piece of scientific software without having to handle software engineering concerns. Thus, once numerical analysts obtain a numerical scheme as a result of the application of their chosen discretization method to the mathematical model, they can directly implement it using the discrete mathematics constructs offered by the language. From this encoded numerical scheme, the infrastructure of the language (*e.g.,* model transformations, interpreters, compilers, code generators) derives the corresponding piece of scientific software.

Languages to implement the scientific software

Finally, when software engineers deal with the execution related concerns (*e.g.,* architecture, hardware, optimization, storage, etc.), system-level languages such as C (https://www.iso.org/standard/74528.html), C++ (https://isocpp.org/), and Fortran (https://fortran-lang.org/) can be used, conjointly with frameworks like OpenMP (https://www.openmp.org/), and standards such as MPI [3]. Language users express the particularities of their simulator with regard to all the concerns involved in the development of scientific software, ranging from the mathematical model, to the encoded numerical scheme, to system-level concerns such as concurrency, memory, and data handling.

The artifacts at each level of abstraction can capture different concerns (e.g., data curation, mesh definition and numerical analysis all relate to a numerical scheme, while concurrency and memory management are both related to the scientific software). Capturing these different concerns in a given artifact can be achieved with a single *general-purpose* language, or with separate, though coordinated, *dedicated* languages, leading to a polyglot development of this artifact.

Moreover, the successive refinement of the different artifacts can be done automatically by the language infrastructure provided by interpreters or compilers. However, this refinement can also be done (at least partially) manually by the different stakeholders, by specifying the behavior of certain concerns according to their specific expertise. While automatic refinement through the language infrastructure provides a predefined way of refining a given artifact, manual refinement lets different roles handle concerns on their own and optimize their implementation for a given context [4]. For instance, a numerical scheme specified with NabLab is usually compiled using one of the compilation chain, thereby automatically taking into account the execution flow, parallelism, and memory model. Yet, one may want to handcraft the C++ code generated from a NabLab specification to customize how the related concerns are handled in a particular application.

Computer languages: V&V techniques to the rescue

Language choice allows to select the level of abstraction at which one wants to work. This determines which artifacts must be defined as part of the development process, and which artifacts are automatically derived through the language infrastructure. While this language infrastructure guarantees the correctness of the derived artifacts with regard to user-defined ones, the V&V concerns corresponding to those user-defined artifacts still need to be addressed.

For example, using a language at the discrete mathematics abstraction level allows numerical analysts to derive the scientific software from their numerical scheme. This derived software is guaranteed to be correct with regard to the provided numerical scheme, but the correctness of both the numerical scheme and the governing equations constituting the mathematical model still remains to be assessed.

We illustrate this on Figure 1, a V-Model for scientific computing, *aka.* scientific V-Model, where the different artifacts involved in scientific software development are represented on the left, from observations, to mathematical model, to numerical scheme, to actual scientific software. Facing each of these artifacts are the corresponding V&V concerns to be addressed. In addition, for each artifact and V&V concern, the figure indicates the associated roles, *i.e.,* the skills necessary to develop the artifacts, and address their corresponding V&V concerns.

The model contains a nested V-model (SE V-model on the figure) representing the artifacts specific to software engineering (SE) that are defined over the course of the development of the actual scientific software, from stakeholders requirements to the implementation. This nested SE V-model also contains the SE-specific V&V activities required to address the V&V concerns corresponding to each of these SE-specific artifacts.

The scientific V-model reads as follows. First, the left descending branch of the V-Model indicates which artifacts must be defined, based on the level of abstraction at which one works: artifacts above the chosen level of abstraction have to be defined as well, as each acts as specification for the artifact directly below. Second, the right

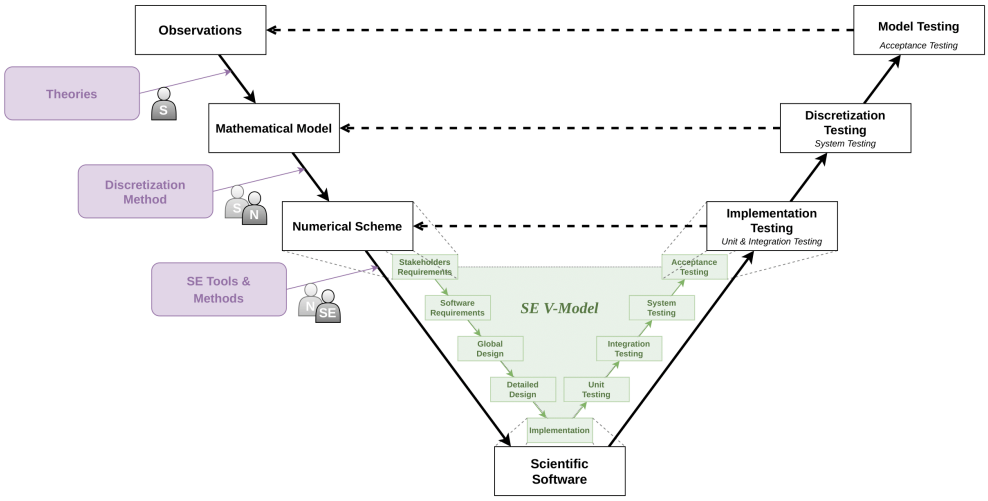

Figure 1: Overall scientific software development process across the scientific V-model [4].

ascending branch of the V-model indicates the V&V activities to be undertaken for each artifact defined by the stakeholders. Thus, this includes the V&V activities corresponding to the artifacts defined with the chosen language, and every V&V activity situated above. In addition, while languages provides guarantees over the software they allow to derive, any V&V activity not handled by a language is left to the developers.

For a more detailed look at the scientific V-model, we direct the reader to our previous work [4].

Classifying Computer Languages for Scientific Computing

In Table 1, we propose a guide supporting decision-making with regard to the computer language(s) to use for scientific computing developments. We evaluate a range of computer languages commonly used in scientific computing [5], [6], aiming to highlight how well each language supports the definition of the three categories of artifacts (mathematical model, numerical scheme, and scientific software), and how they facilitate the required V&V activities for these artifacts.

We propose a scale assigning a score to each language for the development of the three identified artifacts based on their ability to accurately describe these artifacts and the level of expertise required for their use. The more '+' symbols, the more detailed the language can describe the corresponding artifact, and the more expertise it requires from the designer. For example, languages providing fine control over concurrency and memory are better suited if one needs to directly work at the system level to define the scientific software (*e.g.,* for performance or architecture reasons). In the remainder of this section, we give a brief overview of these languages.

The Wolfram Language, provided as part of Mathematica, offers continuous mathematical constructs, while also providing some expressivity with regard to discrete mathematics (https://www.wolfram.com/language/index.php.en). MATLAB works similarly, but also provides discrete numerical constructs (https://www.mathworks.com/products/matlab.html). R is a language more geared toward statistics, but can also be used for matrix computations, and provides continuous mathematical abstractions as well (https://www.r-project.org/). For each of these languages, software engineering abstractions are mostly kept out of the hands of the language user, and the associated concerns are addressed as part of their supporting infrastructure.

NabLab is a language dedicated to numerical analysis, which provides code generators targeting an array of C++ backends [7]. The language exclusively exposes numerical abstractions, and software engineering concerns are addressed as part of the provided generators and compilation

Language	Mathematical Model	Numerical Scheme	Scientific Software
Mathematica (Wolfram Language)	+++	++	
MATLAB	++	++	
R	+	+	
NabLab		+++	
Julia		++	+
SciPy		++	+
Python			+
Java			++
C/C++			+++
Fortran		++	+++

Table 1: Overview of languages commonly used in Scientific Computing according to the associated levels of abstraction

chains.

Julia is a language gaining traction in scientific computing. It offers numerical abstractions, while giving finer control as well over some system concerns such as multi-threading and networking, enabling its use in the field of HPC (https://julialang.org/). However, when such system-level abstractions are used, the corresponding V&V concerns need to be addressed as usual, requiring software engineering skills.

Python is a popular language in scientific computing, despite not providing any native abstractions suited to continuous or discrete mathematics. This popularity stems from its low entry level in terms of software engineering skills (*e.g.,* dynamic typing, managed memory), its extensive library support, such as SciPy (https://scipy.org/) or NumPy (https://numpy.org/) providing the missing abstractions for scientific computing, and a mature support for the definition of wrappers for C/C++ applications.

Java does not natively provide mathematical abstractions, but abstracts some system-level concerns, such as memory management. It is also cited in the literature as one of the frequently used languages by the scientific community [6].

C and C++ are extensively used in the scientific computing community, despite missing numerical and mathematical abstractions, and working at a very low level of abstraction [8]. This is due to its good performance and the significant number of libraries available for scientific computing. However, developing scientific software with C or C++ demands to address numerous software engineering V&V concerns, which come in addition to the usual numerical and mathematical V&V concerns. Fortran is a similar case to C and C++, except that it does provide numerical abstraction, as it was designed first hand to write scientific software (https://fortran-lang.org/).

Conclusion

In this paper, we presented a scientific software development process that integrates both scientific computing and software engineering activities: the scientific V-model. This model describes how the different artifacts and stakeholders involved in this process are related to each other. We also provide a categorization of computer languages according to their ability to develop specific artifacts and discuss the impact on verification and validation activities.

We argue that, when choosing the computer language to implement a scientific software, the modeler must consider the level of abstraction at which they are working and keep in mind that this choice has an impact on the V&V activities they must manage. To facilitate an informed decision on the choice of computer languages, we finally provide a guide gathering the most commonly used languages in scientific computing with respect to the skills required to take full advantage of them in the definition of artifacts and associated V&V activities.

Finally, with this article, we wish to make scientific computing practitioners aware of the role of computer languages and to initiate the discussion on the information needed when faced with the choice of computer language(s) to use in scientific computing.

ACKNOWLEDGMENT

We would like to thank our colleagues at the Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), at the Geoscience department of the Observatoire des Sciences de

l'Univers de Rennes (OSUR), as well as at the Commissariat à l'énergie atomique et aux énergies alternatives (CEA), for their input and providing such interesting conversations that contributed to the writing and improvement of this paper.

REFERENCES

1. Wilson, G., Aruliah, D.A., Brown, C.T., Hong, N.P.C., Davis, M., Guy, R.T., Haddock, S.H., Huff, K.D., Mitchell, I.M., Plumbley, M.D. and Waugh, B., 2014. Best practices for scientific computing. PLoS biology, 12(1).
2. Cohen, J., Katz, D.S., Barker, M., Hong, N.C., Haines, R. and Jay, C., 2020. The four pillars of research software engineering. IEEE Software, 38(1), pp.97-105.
3. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
4. Dorian Leroy, June Sallou, Johann Bourcier, and Benoit Combemale. When scientific software meets software engineering. *Computer*, 54(12):60–71, 2021.
5. Prakash Prabhu, Hanjun Kim, Taewook Oh, Thomas B. Jablin, Nick P. Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, David Walker, Yun Zhang, Soumyadeep Ghosh, David I. August, Jialu Huang, and Stephen Beard. A survey of the practice of computational science. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.
6. Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Jeffrey C. Carver. Software engineering practices for scientific software development: A systematic mapping study. *Journal of Systems and Software*, 172:110848, 2021.
7. Lelandais, B., Oudot, M.P. and Combemale, B., 2018, October. Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo). In Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (pp. 200-204).
8. Joe Pitt-Francis and Jonathan Whiteley. *Guide to scientific computing in C++*. Springer, 2017.

Dorian Leroy is a Research Engineer at CEA. His research interests lie in the field of Software Language Engineering and include metaprogramming approaches and V&V facilities. Contact him at dorian.leroy@cea.fr.

June Sallou is a Postdoctoral Researcher in Software Engineering at University of Rennes 1. Her research interests include Scientific Modelling, Approximate Computing and Environmental Science. Contact her at june.benvegnu-sallou@univ-rennes1.fr.

Johann Bourcier is an Associate Professor of Software Engineering at University of Rennes 1. His research interests in Software Engineering include Self-Adaptive Systems, Model-Driven Engineering, and Distributed and Heterogeneous Software Environments. Contact him at johann.bourcier@irisa.fr.

Benoit Combemale is a Full Professor of Software Engineering at University of Rennes 1. His research interests in Software Engineering include Software Language Engineering, Model-Driven Engineering, and Software Validation & Verification. Contact him at benoit.combemale@inria.fr.