Epistemic Issues in Computational Reproducibility: Software as the Elephant in the Room

Alexandre Hocquet & Frédéric Wieber
Archives Poincaré
UMR 7117 CNRS & Université de Lorraine
91 avenue de la Libération
54001 NANCY

corresponding author :
Alexandre Hocquet
Archives Poincaré
UMR 7117 CNRS & Université de Lorraine
91 avenue de la Libération
54001 NANCY
alexandre.hocquetATuniv-lorraine.fr

ETHICAL STATEMENT:

• Funding: (if any) : none

• Conflict of Interest: The authors declare that they have no conflict of interest.

• Ethical approval: For this type of study formal consent is not required.

• Informed consent: For this type of study formal consent is not required.

Abstract

Computational reproducibility (i.e. issues of reproducibility stemming from the computer as a scientific tool) possesses its own dynamics and narratives of crisis. Alongside the difficulties of computing as an ubiquitous yet complex scientific activity, computational reproducibility suffers from a naive expectancy of total reproducibility and a moral imperative to embrace the principles of free software as a non-negotiable epistemic virtue. We argue that the epistemic issues at stake in actual practices of computational reproducibility are best unveiled by focusing on software as a pivotal concept, one that is surprisingly often overlooked in accounts of reproducibility issues. Software is not only about designing and coding but also about maintaining, supporting, distributing, licensing, and governance; it is not only about developers but also about users. We focus on openness debates among computational chemists involved in molecular modeling software packages as empirical grounding for our argument. We then identify and analyse four epistemic characteristics (transparency, consistency, sustainability and inclusivity) as key to the role of software in computational reproducibility.

Keywords

computational reproducibility, software, computational chemistry, transparency, consistency, sustainability, inclusivity

Article

1. Computational reproducibility

Is there “a growing sense that science has reached a reproducibility crisis” (Getzelter, 2015)? The tension between reproducibility as a general moral imperative of “the core principles of science”, and the acknowledgment in almost every quarter of scientific domains that reproducibility indeed poses many problems, is key in the feeling that reproducibility in science might be in crisis (Baker, 2016). More often than not, this crisis is addressed in terms of epistemic transparency as a necessary and sufficient condition for science to be reproducible, in the (vague) name of open science.

Through the narratives of a reproducibility crisis in science throughout the last decade, there have been attempts to classify reproducibility into categories (for a review, see Atmanspacher and Maasen, 2016). The so-called “computational reproducibility” emerged as one of them. Computational reproducibility is naturally linked to the computer as a scientific tool: it “generally refers to the description and sharing of software tools and data in such a manner as to enable their use and evaluation by others” (AlNoamany & Borghi, 2018, p. 3). The notion has gained momentum because of the pervasiveness of the computer within scientific activity. In these narratives, computational reproducibility as a category is often contrasted with what is often referred to as experimental reproducibility. Computational reproducibility is, in this context, often naively perceived as the part of reproducibility that could (and should) be achieved exactly: “to re-run a computation however many times as wished for, in whichever context, without changes”[1].

Philosopher of science Sabina Leonelli, unsatisfied with an overarching definition of reproducibility across scientific disciplines, has convincingly defined six categories of reproducibility varying in scope and space (Leonelli, 2019). Her aim is to assert that scientific fields define reproducibility in diverse ways and that the very relevance of the concept of reproducibility is itself variable. The first of Leonelli's six categories is named “computational reproducibility” and is, according to her, the only one concerned with total reproducibility of datasets (as opposed to the plain reproduction of patterns or even inferences). Computational reproducibility is, in Leonelli’s words, the only one that is agnostic towards the circumstances of data production. In this regard, Leonelli, in line with her description of data-centric science, suggests a view of computation as an activity that is limited to data treatment. Many accounts of reproducibility conform to this view when it comes to the computational part of reproducibility (see for example Peng, 2011). Computational reproducibility is defined in those accounts as wearing the burden of a requirement of total reproducibility. We are sympathetic to Leonelli’s categorization because it offers a nuanced depiction of the reproducibility landscape in science, and we acknowledge the importance of computational reproducibility as a

[1] As per the wording of an anonymous reviewer.

category of its own. Nevertheless, we argue that the landscape of computational reproducibility is itself more complex.

Practitioners such as computer scientists, computational scientists or software curators express concerns about computational reproducibility. For example, Benureau & Rougier (2018) consider that problems of replicability in computational software exist precisely because "it is easy to believe that if a program runs once and gives the expected results it will do so forever" (Benureau & Rougier, 2018, p. 1). AlNoamany & Borghi (2018) observe that in discussions about reproducibility, emphasis is most of the time put on data-related scientific practices but not sufficiently on software practices. Yet, computational reproducibility poses specific problems, such as "the lifetime reproducibility of a particular code" (Hinsen & Rougier, 2019, p. 634), or "the lack of transparency in disclosure of computational methods" (Stodden et al., 2016, p. 1240). What is at stake is thus not only to be able to reproduce data, but also to focus on the disclosure and sustainability of computational methods used to produce data in the first place.

To a certain extent, advocating that computational reproducibility issues are important is a way, for some practitioners, to stress that software has to be taken into account within the Open Science movement, along with publications (as in Open Access) and data (as in Open Data). As Hey & Payne (2015) write: "[The] global momentum towards 'open science' […] necessarily requires not only open access to research publications, but also to the metadata and data required to validate and make sense of the results of research. And improving the comprehensibility and reproducibility of computational science is an important step in this endeavour [by offering] access to the *software*, data and computer environment used to produce the results." (Hey & Payne, 2015, p. 367, our emphasis).

Thus, computational reproducibility cannot be reduced to reproducibility concerns associated with open or closed literature, even if they are linked to matters of publication's practices (like for example the availability of the code used within a scientific paper). Nor can it be reduced to reproducibility concerns associated with closed or open data, even if they are linked to issues of data disclosure (like for example the minutiae of model parameters used in a calculation). A common statement made by most practitioners discussing computational reproducibility is that computational methods have led to significant changes in actual scientific practices. Yet, they complain that the disclosure of models, computational steps and computing environment used to produce data is insufficient in the published scientific literature to achieve reproducibility. "A detailed understanding of what a given piece of software does is often limited to the software's authors", as Hinsen (2014, p. 2) summarizes. He even goes further as, for him, "there is no clear separation between the tool (software) and the model it operates on". Hinsen considers that computational scientists should strive to explain to their peers the computational models and methods they use, because "scientific software is much too complicated to be an efficient way to communicate these models and methods" (Hinsen, 2014, p. 2). Computational transparency is not that easy to achieve, indeed.

Whether they are writing, using or reviewing scientific software, actors express their dismay about how to achieve reproducibility in practice. We reckon that the issue of computational reproducibility linked to software is complex and we aim to address this complexity, beyond the mere requirement for code transparency. To this end, we argue in this paper that epistemic issues

beyond transparency are key to the role of software. In addition to transparency, we offer a categorization of three more epistemic characteristics (namely consistency, sustainability and inclusivity) to depict a more complex epistemic landscape. We also argue that this complexity is to be understood within the entanglements of the different layers of what we call a software millefeuille, beyond its reduction to "code". To put it bluntly, software, even though ubiquitous in the practices, is surprisingly missing in the analyses, like the elephant in the room (Hatton and van Genuchten, 2019).

2. Computational Science and Software

Our way to deal with computational reproducibility is to focus on an important subpart of computational science, one that is concerned with calculations based on scientific models translated into computer programs. This activity of modeling as well as simulations has received considerable interest from philosophers of science. It even possesses its own "epistemology of computer simulations" (Winsberg, 2019, 2010).

One of its features, according to Winsberg, is that computer simulations are "motley" in the sense of that they may depend not "just on theory but on many other model ingredients and resources as well, including parameterisations, numerical solution methods, mathematical tricks, approximations and idealizations, outright fictions, ad hoc assumptions, function libraries, compilers and computer hardware, and perhaps most importantly, the blood, sweat, and tears of much trial and error" (Winsberg, 2019, part 4.1). Procedures of validation (between theory and model) and verification (between model and program) are not that easily discernable, according to Winsberg. For instance, discretisation techniques, as a classical example of epistemic trade-offs, rely more on engineering and programming practices than on pure formalization of theories. In the same vein, Lenhardt and Küster (2019) explicitly express computational reproducibility concerns in terms of this kind of imperfect "numerical solutions". Building on this acknowledgment of the entanglement of engineering practices and formal theories, Symons and Alvarado (2019) discuss the concern for trust in computer simulations. They argue that "computer simulations and computational methods in general—as instruments deployed in scientific inquiry—are neither reliably transparent conveyors in all contexts, nor can they be regarded as equivalent to expert sources of testimony" (p. 47). "Thus, when computer simulations can be trusted it is because of their adherence to theoretical principles, empirical evidence, or engineering best practices and not because of their output alone" (p. 52). Winsberg (2019) adds that "the credentials [of computer simulations] develop over an extended period of time and become deeply tradition-bound. In Hacking's language, the techniques and sets of assumptions that simulationists use become 'self-vindicating'. […] the locus of our trust in simulations [resides] in practical aspects of the craft of modeling and simulation, rather than in any features of the models themselves" (part 4.2). These descriptions and analysis of computer simulations clearly emphasise the complexity of computational methods. They show that to trust a computational simulation, one needs more than the validation of a simulated model.

Some authors express this idea in terms of software. Gelfert (2011) distinguishes two aspects concerned with computational reproducibility. First, so-called "software" aspects involve the

implementation of a model into a computational template[2] that makes it computationnally tractable. Gelfert considers that "this stage of setting up simulations often involves tacit know-how on the part of the investigator and is rarely fully documented" (p. 154). He then adds that "replicability in the case of computer simulation, however, also has a 'hardware' component, […] [that] concerns the further problem of the reliability of the concrete device, on which a simulation is being run" (p. 154). Focusing on the influence of software on these reproducibility matters, Horner and Symons (2014) posit that "the high conditionality of typical [scientific] software […] implies that [...] there is no known effective method for characterizing […] the error distribution in software" (p. 491), thereby asserting the difficulty and specificity of the task.

Thus, engineering practices of programming, tacit know-how and entrenched craft, software error management, implementation on concrete devices highlight issues that belong to the realm of software. The requirement for transparency as a simple way to achieve total reproducibility of computational methods, either at the level of computational templates or within code, appears then naive. Yet, we consider that software, in those accounts (as algorithms in the case of Gelfert, or as programming in the case of Symons and Horner), is viewed as too narrow and not sufficiently historicized. Building on these works, but also on the field of the history of software, we thus aim to expand on the multifaceted nature of software to better apprehend the complexity of computational reproducibility issues, as a simplistic moral imperative for more transparency may be not enough.

As the entanglement between all these aspects is of tremendous importance for the issue of computational reproducibility, we are choosing, as a case study, the field of computational chemistry to shed light on this entanglement. Computational chemistry is an interesting computational science, with peculiarities concerning software. Because of its particular historical context of development, because of its proximity to the pharmaceutical industry, computational chemistry has had to deal with academic norms as well as business norms (Hocquet & Wieber, 2017). Software packages, in this context, are devised to produce novel scientific results. It is also important to acknowledge that they are distributed, maintained and licensed under these sometimes conflicting norms. We thus argue that computational chemistry is an interesting field to engage in, so as to articulate multiple dimensions of software within the problem of computational reproducibility.

3. Computational chemistry as a case study

Computational chemistry is a scientific community involved in designing models to explore the physico-chemical properties of molecules (and materials). It is a field where reproducibility concerns are expressed in numerous "benchmarks" to assess the validity of computer simulations. These "benchmark" publications (some of them contain the word "reproducibility" in their very title) strive to compare scientific models such as density functionals (Lejaeghere et al. 2016) or molecular dynamics force fields (Schappals et al., 2017). These benchmarks include their mathematical expressions, the parameterization of these expressions, their translation into code, and their embedding into software packages. As an extreme example, a recent publication has even

[2] On computational templates and computational tractability, see Humphreys (2004).

highlighted a Nuclear Magnetic Resonance (NMR) chemical shift calculation that depends on the operating system the software is installed on! (Bhandari Neupane et al., 2019)

Our way to study computational chemistry is to focus on application software designed by computational chemists to model physico-chemical properties, a genre of software designed within the community and for the community. While some of the members of this community do program software as developers, others are only users of these application software packages, either in Academia or in the industry. Some even sell or market software packages. The profiles in this community are thus extremely diverse when it comes to using or developing software. It is also notable that computational chemistry has emerged in times of mobilization of American universities to produce innovation and was involved with two major industries: the computer manufacturers and the pharmaceutical industry, the latter becoming a potential market for the former through molecular modelling software packages. In this specific context, tensions between academic norms and business norms arise (Hocquet & Wieber, 2017).

In another publication (Wieber & Hocquet, 2020), we have argued that opacity in computational chemistry models lies in different layers: in the parameters of the mathematical formalization of the model itself, in the code into which these models are translated, in the packages where these implementations are embedded, in the licensing policies that define how software is shared and used, and in the communities of developers and users. We have also argued that these layers are entangled and not easily separated. Our point is that models and software have to be addressed together: beyond the entanglement of theories and motley models, and models and code, we argue that other dimensions of software are to be studied.

Hence, our aim in the present paper is to move a step further by shedding light on the incompleteness of expressing reproducibility in terms of mere transparency of code. To better comprehend the complexity of computational reproducibility, we describe different layers of what we call a software millefeuille. We build on this description to depict three more epistemic characteristics (namely consistency, sustainability and inclusivity) beyond transparency.
As empirical grounding for our argument, we are focusing in this paper on a timely series of five opinion pieces evolving into an asynchronous debate within the computational chemistry community. They have been written and published in 2015 and 2016, and they explicitly refer to a "reproducibility crisis". The articles have been published in "Journal of Physical Chemistry Letters" for three of them and in blog posts for three of them (one was a blog post eventually turned into a paper). We make use of these five papers and we also at times refer to debates within the Computational Chemistry List along the years, especially the so-called 1993, 2001 and 2011 flame wars we have analyzed before (Hocquet & Wieber, 2017).

The first paper (Gezelter, 2015) has a self-explaining title: "Open Source and Open Data Should Be Standard Practices". Gezelter, as a computational chemist and an Open Science activist, argues for code sharing as desirable transparency, especially for peer reviewing, encourages versioning to improve code sustainability, and laments the lack of academic recognition for the activity of developing software. It is followed by a paper by ten co-authors (all involved in the development of commercial software packages among the most renowned in their field) entitled "What is the price of open source software?" that counters Getzelter's arguments (Krylov et al., 2015). The aim of this reply is to stand up for their business and epistemic model, and to question the feasibility and

sustainability of open source software in computational chemistry. This second paper is in turn replied to by a blog post co-authored by four ("The Price of Open-Source Software – A Joint Response") (Chue Hong et al., 2015) and another paper by Jacob ("How Open Is Commercial Scientific Software?") (Jacob, 2016) that mitigate both views. A fifth blog post by Miletic, "What is the price of open-source fear, uncertainty, and doubt?", criticizes Krylov et al. from the viewpoint of libre activism (Miletic, 2015). Chue Hong and his co-authors are software engineers and work in the field of scientific software sustainability. Jacob is a computational chemist and part of the development team of yet another commercial package. Miletic is a contributor to a decentralized open source package project. Throughout these five pieces, typical software and epistemic issues related to computational reproducibility are discussed, a diversity of viewpoints are opposed, arguments are put forward, countered and debated.

4. Software and epistemic issues of reproducibility

Software is a difficult topic to grasp, first of all because its very definition is elusive. As historian of software Nathan Ensmenger puts: "Although the idea of software is central to our modern conception of the computer as a universal machine, defining exactly what software is can be surprisingly difficult" (Ensmenger, 2010, p. 7).

In many accounts, software and code are interchangeably used, yet software encompasses many dimensions that go beyond what is understood as code. Ensmenger summons the notion of socio-technical object to circumscribe these many dimensions. "Unlike hardware, which is almost by definition a tangible thing that can readily be isolated, identified, and evaluated, software is inextricably intertwined with the larger sociotechnical system of computing that includes machines [...], people (users, designers, and developers), and processes [...]" (Ensmenger, 2010, pp. 7-8).

To address computational reproducibility, we argue that a focus not only on the design of programs but also on the implementation, distribution, maintenance, support and uses of software is necessary. For an operational assessment of the different yet entangled layers of the software millefeuille, we will in this paper try to stick to a categorization that distinguishes between the model, the code, the computing environment, the package, the licence and the community, even though each of these categories are not impermeable, and even though other phrasings exist to designate layers within or in between those categories, such as methods, algorithms, or programs. We are trying here to design an operational categorization to highlight the diversity of layers at stake.

We thus are choosing the term *"model"* as encompassing the methods for which scientists envision the possibility to compute "numerical solutions" out of them. By *"code"*, we mean the translation of those models into a language that a computer comprehends. Both layers are entangled. *"Computing environment"* refers to the interactions of code with other computer layers such as operating systems, or libraries dependencies, or the compiling into an executable. It is also entangled with the previous layers because some hardware evolution like parallelization or the use of graphical processing units need reassessment of programming and calculation techniques. *"Package*" illustrates that part of software that is designed as something to be distributed. It might embed several different scientific models into an assemblage of code sub-units, possibly tied together within a user interface, and possibly evolving into a full software suite. Support and

maintenance, when they exist, apply mainly to this package layer[3]. The *"licence"* is the contract that links the package and its users. It defines under which conditions the package is distributed and used. Finally, the *"community"* embraces scientists, developers, programmers, maintainers, professionals, possible marketers and sellers, but also users. Users are in turn potentially further divided into categories such as lay users or lead users.

We argue that not only transparency is difficult to achieve indeed throughout this software millefeuille, it is also not enough to account for the reproducibility problems at stake. We thus introduce three other epistemic characteristics, namely consistency, sustainability and inclusivity that are distinct from transparency and yet play a significant role in computational reproducibility issues.

We define *"transparency"* as the ability to check how a tool functions and what it does. As far as software is concerned, transparency is obviously linked to open-source code as a commonly-viewed necessary prerequisite to any reproducibility attempt. Yet, transparency extends well beyond the scope of open-source code as we clarify further down in this paper. We define *"consistency"* as the ability to be certain about which tool is actually used in which computing environment, transparent or not. Transparency may not be enough for replication, if one cannot ascertain which transparent tool is used in which computing environment, and therefore ensure that said tool does what it claims it does. Consistency is linked for example to the practices of versioning, or to software dependency problems. We define *"sustainability"* as the guarantee that the tool works the same way over time, or with different hardware configurations, or with different compilers. Sustainability is linked to support, maintenance but also portability. Sustainability is an essential matter for results to be replicated over time or different computing environments, something that transparency and consistency alone cannot offer. We define *"inclusivity"* as the ability for anyone to use, benchmark or develop the tool. Transparency, consistency and sustainability may be not enough for replication if some people, but not all, are actually allowed, or empowered to test or try to replicate. Evidently, these characteristics are interrelated. Yet, they are also at times conflicting, and we argue that this categorization is useful to illustrate the tensions within computational reproducibility. Consistency, sustainability and inclusivity are in a way the hidden part of the iceberg of computational reproducibility. Beyond the sole transparency, taking those four characteristics into account altogether allow for a deeper understanding of the epistemic complexity of computational reproducibility.

In the following sections, we describe three different perspectives about how the concept of software can be grasped in the scientific milieu. First of all, software is akin to a scientific instrument, a tool that is used by scientists to perform scientific calculations. Secondly, as scientific software development norms are at times at odds with academic norms, software is also a business model, being actually either pursued or criticized as such. Finally, software introduces a social dimension. First, development involves many interacting humans. Software also defines a relationship between users on the one hand and developers on the other. This social dimension raises the bureaucratic question of governance as a reproducibility matter.

[3]We thank a anonymous reviewer for suggesting "documentation" as yet another layer. While we agree that a variety of epistemic concerns may arise from such a layer, we have chosen, based on our case study corpus, to account for the documentation layer within what we have named the "package" layer.

In these three different perspectives, the way the four previously defined epistemic characteristics are interrelated is diverse, bringing a kaleidoscopic view of computational reproducibility epistemic issues. We will now assess software as instrument, software as distribution and software as governance in terms of these four epistemic characteristics linked to the six layers of software millefeuille (model, code, computing environment, package, licence and community), in order to stress how computational reproducibility is impacted.

5. Software as Instrument

Scientists construct models in ways that are linked to the phenomena their models try to represent, to the theories they can use, but also to the technological, professional, economic and political context in which they work. These epistemic roots are a pivotal step to understand how the models they construct are then translated into software, and how these models influence the actors' discourses and the tensions at stake among them. As the historian of software Michael Mahoney writes, computational scientists "have put their portion of the world into the computer" (Mahoney, 2008, p. 8). These models, and their translations into software, are "operative" representations. "Through the computer, we can manipulate them, and they in turn can trigger actions in the world" (Mahoney, 2008, p. 13).

In a publication describing the designing of a piece of software within a computational fluid dynamics academic laboratory, Spencer (2015) points out this performativity: "The focus on the framework, on keeping it workable, portable, durable, arises from the intersection of the properties of expanding software with its conditions of application as a particular kind of medium for research". Scientific software thus shares many aspects with scientific instrumentation. It is a tool, designed by scientists for scientists, part and parcel of the scientific lab.

Typically, the actors of our corpus view software as a complex scientific instrument. The Nuclear Magnetic Resonance (NMR) apparatus is a frequent metaphor they employ. Not only NMR and molecular modelling are the bread and butter of many chemistry labs instrumentation, the metaphor is also used because NMR is exemplar of modernity and complexity in the chemistry lab, at the crossroads between Big Science and Entrepreneurship Science, as narrated by Reinhardt (2006).

Two different, almost opposed, Weltanschauungen regarding scientific instrumentation are referred to in the papers of the corpus we explore. In the first one, the instrument is home-made. The practitioner knows the instrument well. The trust of the community is based on the *transparency* of the instrument: it is open for all to see. The claim here is that transparency is the epistemic virtue that entails reproducibility. Failing to achieve transparency leads to the instrument being accused of being a so called "black box", i.e. an instrument where it is not known what is going on "under the hood".

In the second one, the instrument is manufactured by a corporation. The trust of the practitioner and the community is based on an industrial commodity, that is supposed to be calibrated, standardised and even liable. The corporation is in charge of providing a reliable instrument. It is not challenged to open the black box. Reproducibility is here based on the stability and dissemination of a (industrial) standard. Failure to warrant it leads to issues of *consistency*, as in the proliferation of

non-certified versions of the instrument, and *sustainability*, as for example in the lack of its maintenance.

Of course, these are ideal types: software, as instruments in general, may belong to an intermediate category, or be as complex as to mix parts belonging to each category. Instruments may even share characteristics of one category and the other at the same time (for a review of epistemic instrumentation issues in chemistry, see Reinhardt, 2001). Yet, these two ideal types function in the narratives of the actors to oppose two Weltanschauungen of how reproducibility is built.

In our case study, the two types are exemplified in advocacy for open source software vs proprietary software. The proponents of open source software as an imperative typically argue that transparency should be a must for "taxpayers funded" science. Science as a public good should not be enclosed into proprietary software. Transparency must therefore be achieved by publication of the details of the *model*, the source *code* being open (and intelligible), accessible to reviewers, to users, or to any "taxpayer". According to whom it is actually transparent, *inclusivity* concerns are here at stake.

These arguments are countered by Krylov et al. (2015) by referring to the complexity of a quantum chemistry program. The metaphor of the "phone app" is used to describe what in contrast "complex quantum chemsitry code" is: it is slow to write, develop, maintain and support and the community of contributors is small, akin to an NMR apparatus, whereas the "phone app" can afford to be "free" (the narrative is replete with confusion between free beer and free speech) because the phone app, in their words, is easy to write, is disposable, and the community of contributors can be large. Therefore, proprietary licensing is used by "complex quantum chemistry code" developers to generate revenue that allows to finance the required workforce to develop, maintain and support the software *package* in order to achieve *sustainability* and *consistency*. Reproducibility is to be attained thanks to this proprietary financing.

Moving beyond the open/proprietary dichotomy, the well known (in the field) Gaussian *package* is subject to a similar tension. The company is priding itself on providing a readable source *code* (to allow *transparency*), yet it forbids its compiling with unsupported compilers. It does so to avoid the proliferation of "suboptimal" versions of the official *package*, expressing concerns of *consistency*. It also does so to avoid the portability to unsupported hardware, thus expressing concerns of *sustainability* regarding the *computing environment*. The debates around this very tension have been analysed in Hocquet & Wieber (2017). As Chue Hong et al. (2015) point out, the mere publication of equations, parameters or algorithms or even source code is not enough to warrant reproducibility. The mere compiling of a piece of software or the dependence on evolving libraries may jeopardize the reproducibility of a calculation. Hence, epistemic characteristics are clashing here: *transparency* on the one hand, as expressed by the fact that a readable source code may not be transparent enough for reproducibility of calculations, and *consistency* on the other, as expressed by the fact that a modifiable source code may lead to a proliferation of versions that hamper the consistency of results.

6. Software as Distribution

Software is also what turns code into an activity linked to the rest of the world, sometimes an industrial one. Nathan Ensmenger argues that software, as the interface between computer and

society, defines our relationship to the computer. “Software sits uncomfortably at the intersection of science, engineering and business, a heterogeneous technology that blurs the boundaries between the technological and organizational, fraught with disputes over conflated social, political and technological agendas” (Ensmenger, 2010, p. 8).

In the professional history of software, the phrasing "software crisis" describes a major disenchantment that occurred soon after software was envisioned as a business opportunity: in a nutshell, from the late nineteen sixties onwards, the computing industry realised that while the performance of hardware went up and its cost went down, the software followed the exact opposite trend, precisely because of the increased difficulty to keep up with the hardware development pace, with increasing demands for workforce, for investment and for technology, especially in the domains of maintenance and support more than the actual design or programming. More than a turning point, the "software crisis" has been known more as an art-de-vivre ever since (Ensmenger, 2010). In his description of the academic developing of a piece of software, Spencer (2015) similarly emphasises on the tensions introduced by time: “Both brittleness and the accumulation of supporting processes make demands on time, whether it is time spent figuring out how to make changes or fix bugs, or time spent adhering to new administrative routines for tracking changes and logging bugs”.

Software has also to be understood as something to share or distribute. More than fifty years ago, scientists began to write programs to be used by others. “Market-oriented” scientific software thus shares concerns that are similar to the general software industry. Even if scientific software is not necessarily a commercial product (and it sometimes is), it is prone to commodification. Miletic (2015) argues that the limit of the NMR instrument metaphor shows when distribution issues arise: from a certain open point of view, the sharing of a piece of software is unlimited. Yet, issues of (lack of) academic recognition regarding the labour of programming can be found in scientific fields where software is a tool and not the object of research itself. The notion of a (lack of) "business model" is ubiquitous in actor phrasings, exemplifying the tensions between academic norms and business norms.

Business models

Software developing oftentimes is an unrewarded scientific activity within Academia. The traditional publication reward of scientific activity fails to account for the amount of work, energy, time and money invested in developing a scientific method turned into a piece of software. It is an acute problem in computational science because developing, and even more so maintaining, is barely the core activity of computational scientists, in contrast to computing scientists. In this regard, computational scientists are often portrayed as “bad programmers”. Many computational reproducibility issues are summed up into a lack of sound programming practices (AlNoamany & Borghi, 2018). In the same vein, the lack of incentives and rewards, or mechanisms of recognition for software developing (quite similarly to data curating) is a well known problem mentioned in computational reproducibility pamphlets (Stodden et al., 2016). What is much less mentioned however is the fact that a "sustainable business model" is one of the way computational scientists try to address the issue.

Not unexpectedly, proprietary software is criticized by open source promoters. Not only do proprietary *packages* turn software into epistemically opaque scientific instruments that lack *transparency*, Miletic also points out that they trap their users-customers in a "vendor lock-in" situation: price is not negligible, users cannot benchmark, *portability* is discouraged, and above all, users are restricted to only make use of the scientific methods that are validated by the software corporation. The market is such that distributors (industrial or academic turned into companies) embed several (but not all) molecular modeling methods into competing software *suites* that are mutually exclusive in terms of the choice of *models*. Access to the tools is thus restricted, expressing an issue of *inclusivity*.

Even though it is common to find arguments in favour of alternative business models in the realm of general open source software (like paid services, assistance, support, maintenance...), it is much more difficult for a “complex quantum chemistry code” with a limited community of developers and users, in contrast to the (in)famous phone app. Actually, even the most hardcore promoters of open source in our case study acknowledge the need for a sustainable business model. Still, their suggested business models are vague or surprising (like a proprietary GUI around an open source engine, for example). The “cost of open software” in general is actually an issue that is gaining attention (Geiger et al., 2019).

Proprietary packages rely on a professional workforce to ensure "strict coding guidelines" (in Krylov et al. words), support and maintenance (at least in their rhetorics). Linked to the issue of industry, recognition, market and workforce, they express a concern about the reliance on interns, graduates or post-docs to code (an activity that is detrimental to their career), de facto transforming junior researchers into the lumpenproletariat of scientific software in "free software" development. The portability of the *package* into diverse pieces of hardware, the rewriting of the *code* with regard to vectorisation, parallelisation or GPU processors, are issues of *computing environment sustainability* that, they argue, can only be addressed by a (well paid) professional team of programmers. Yet, it is argued on the opposing side, that this labour is also part and parcel of the computational scientist toolkit. Such understanding, leanrt by experience, of the entanglements of model and code are empowering the coding scientist. The division of labour is thus criticized as an issue of *inclusivity*, since it lowers the number of people able to delve into these entanglements.

It is also argued that equating *transparency* of software as reproducible good practice with free software principles may lead to conceptual dead ends. As scientists turned into software developers commonly fear for the stealing of their code (precisely because this activity requires a lot of investment and gives only minuscule rewards in return), some are wary to give away their code to the reviewer, even though the code is supposed to be licensed under an open licence. Delaying for as long as possible the sharing of ones code is a technique used to minimize the dangers of being stolen of a cutting edge scientific tool in a competitive scientific world. The actual practices of publishing and reviewing in competitive research are surely key here, and this participates in a wider wariness of scientists towards the publishing process. In contrast to code commenting as recommended sound research practices that allow intelligibility and thus *transparency*, the practices of code obfuscation do exist as a perverse technique to make an open source code *less* intelligible, in the same competitive outlook. The notion of an open source *code* is thus not as straightforward as it seems.

Licensing policies

Licensing is what permits the distribution of software from developers to users under some form of law, by defining the contractual obligations of both parties. In that sense, business models are materialized into sharing practices by licenses. Licenses are also a way to materialise what it means to be “open” when referring to free software as a role model for open science. Licensing is a core practice of software developing and it is, in particular, fundamental in order to be able to embed the principles of free software into an actual package. Yet, it is frequently overlooked. A lot of scientific programs are "given away for free" without any licensing considerations, thus leaving in the dark the actual distribution strategies of the program owner (AlNoamany & Borghi, 2018).

The Gaussian software package example that we mentioned earlier allows to shed light on a blind spot in the open source narrative for reproducibility: it is often argued that the source *code* should be available, and the weakest sense (open as available to read) actually is problematic, not only because trying to read a source code raises its own issues of intelligibility, but also because, available to read may be not enough. If a source code is readable but not available to reuse (modify, compile, benchmark...), then its "openness" is debatable. Since *transparency* as a virtue of reproducibility often refers to the commitments of free software as a role model to follow, it is surprising that so little case is made for reuse as an "essential freedom". The tension here is between the readability of the source *code* (to ensure *transparency*) vs the control of the stability of the *package* and its versions (to ensure *consistency*).

On the other hand, the issues of licensing may be more complex than this Manichean view. Consider, for example, the parameterization of modelling methods. It is a fundamental step in the modelling process. The readability of parameters may be hidden away in the *code* (thus affecting *transparency***)**, but the licencing policy has also a role here, especially when it discriminates between academic and industrial users. Industrial users are for example sometimes given the possibility to alter modelling parameters (for a higher licensing price). The industrial license thus allows more *transparency* that the academic one, because the user can actually check and benchmark parameters (the black box is open). Yet, actors argue that published calculations from industrial licensees may have used altered parameters, thus revealing a problem of *consistency* in the *model* being used for the calculations, because of the then possible obfuscation of which modelling parameters have been actually used in a calculation (Wieber & Hocquet, 2020).

7. Software as Governance

Software is also a community of developers and users. The work of anthropologist Chris Kelty has focused on the practices of free software communities (Kelty, 2008). Even though scientists involved with software adopt a diversity of standpoints regarding what is open or not in software and how important it is, the cultural influence of the free software community shows in the debates around scientific software. It certainly has an influence on how software is tackled in computational science. Among the core practices of free software communities that Kelty (2008) defines (such as sharing source code or defining licensing policies), one of them is organising communities around the development and use of software. Light is thus shed on the importance of social aspects on software matters in scientific activity, and their materialization into bureaucratic policies of development. Defining licensing policies (as seen earlier) and organising bureaucratic forms of

governance are essential in these communities, according to Kelty, and these “core practices” are also salient in the computational scientific communities. As Spencer (2015) reminds us, “tendencies towards […] bureaucracy could be discounted as merely practical, having little bearing on the legitimacy of discourse. But they play a major role in conditioning the investigations through which such discourse is generated in the first place. They are the geological processes shaping the landscape of possibilities that scientists navigate when they put their techniques into action”.

Another way of addressing the governance of software projects is to focus on the interactions between developers and different groups of users. As “market oriented” software is distributed to outsiders, and at times customers, the relationship between developers and users as conceptualised by innovation theorists, such as von Hippel, becomes relevant for reproducibility matters. Not unlike the "lead users" in NMR as described by Reinhardt (2006), borrowing from Von Hippel's theory of user driven innovation, different categories of users are emerging or disappearing around software projects. These dynamics exert influence over reproducibility, especially on the issue of who is able to do what, who contributes to software development and how, which users are empowered and how.

Development Teams

The issue of recognition when contributing to a piece of software shines a light on the collective aspect of software contribution. Publication-like processes define how authorship and contributorship to software are addressed and this situation sometimes generates frustration in academic circles. Yet, beyond the issue of who deserves reward for software and how, the actual practices of working together towards designing, developing, distributing, maintaining and supporting a piece of software raises collaboration issues through the critical bureaucratic issue of governance. In the software industry, and especially in open source projects, the vitality of the community of contributors is key. Not only the workforce must be sufficient and sustained (and this is not obvious if the project is "more complex than a phone app"), the issue of governance is influential for reproducibility matters, when it comes to *inclusivity* and *consistency*.

Business competition as a driving force for scientific software improvement is logically criticized from the point of view of open source advocates. The complexity of quantum chemistry programs forces anyone willing to start a project to reinvent the wheel because basic computational subroutines are withheld into proprietary packages. The modularity of those *packages* could be used to collaboratively create better software and this prospect is impaired by the competitive stance of proprietary licences. Competition hinders here the *inclusivity* to access code subroutines or libraries that constitute state-of-the-art routine procedures.

We have mentioned earlier the question of division of labour. Professionalization is regarded as a warrant of *sustainability* and *consistency* for a software package on the one hand. On the other, empowerment of scientists is argued to lead to *inclusivity* as they are then able to both act on models and to write code. Among the issues of "the cost of open software", the community of contributors (its size, its vitality, its governance) is part of a "business model" and "recognition" issue. Unless the software package gets huge public funding (and it sometimes does), it is often difficult to get a team to extend beyond one research group, to form a lasting workforce beyond interns, and to rely on other governance forms than counting entirely on one principal investigator.

If the software *package* project is open source, then another governance issue may raise yet another problem: the forking of a project, a typical practice from the open source software world, may be a dividing threat for a community. Open source repositories are full of corpses of software projects that did not survive the withering of its community. It is a reproducibility issue because of the subsequent lack of *sustainability* of a project if a community fades away. It can also be an issue because of the proliferation of software versions that blurs its *consistency*. Spencer (2015), observing a team of computational fluid dynamics, has described the collaboration difficulties of contributing to a local piece of software, and how the developing team tried to address those bureaucratic issues by importing organisational practices from the free software world. Hey and Pane (2015) describe a successful example with the software involved in acquisition and treatment of data at the Large Hadron Collider, where all the worldwide community of high energy physics gathers in one place (and gets enormous funding). Those bureaucratic processes are a way of achieving *consistency* and *sustainability* by organising collectively the social rules of the versioning and maintenance around a *model* embedded in a software, at the scale of a laboratory (Spencer, 2015) or an entire community (Hey and Pane, 2015).

User-Developer Relationship

We have already described how the advocates of proprietary software argue that they enhance the reliability of their packages with the hiring of professional software developers. But we have also hinted that the actual choice of scientific *models* embedded in *packages* is also pivotal: if the software is in a dominant market position, it has the power indeed to define what scientific *model* will be used by the community of its users-customers. To be part of a developing team (not only the actual coders, but also the scientists involved in the design of the models embedded into the suite) is therefore a strategic choice for the promotion of a model from a research team, and thus for academic careers. Krylov et al. (2015) argue that, while the source code of one of their software suites is not available to the whole community, it is indeed available to the community of partners of the project, a community that can be seen as a kind of lead-users (as defined in Reinhardt, 2006). They call this community an "open teamware" (Krylov et al., 2015) and argue that this selection process is actually what makes the community vibrant. Their source code is also available to its user base, upon request. Jacob, in response, points out that the "on demand source code" is in fact subject to the signing of a disclosure agreement that is dictated by the corporation. Suggestions of improvements to the software suite from “lead-users” research groups are traded in exchange of their surrendering of property rights. This situation results for these lead users in the hampering of potential external collaborations with other scientists as a consequence of those agreements. These mechanisms of *inclusivity* and exclusivity and the associated governance dynamics are thus complex, and are criticized as gatekeeping, and perpetuating power positions (Jacob, 2016). It is striking that the co-authors of the Krylov et al. paper are for the most part well established scientists. They have managed to make a software *suite* prosper along decades, and, as oftentimes in self-made meritocratic narratives, they have inherited this situation by being the students (or the students of students) of founding pioneers. A software suite in this regard is very much akin to dynasty and the stakes of being in or out the contributing team resembles playing the role of a courtier.

Thus, in a Weltanschauung where users are customers, the actual access to *models* poses a problem of *inclusivity*. Moreover, beyond the theoretical possibility of access, the implementation or not within the *package* of so called "user-friendly" empowering interfaces to access the parameterisation of the model, or even the portability to different hardware or operating systems, is a way of defining which user can use the tools (and thus who is in principle able to reproduce the scientific results produced with that tools). In this regard, it is another expression of the lock-in problem that hinders reproducibility.

8. Conclusion

Throughout our analysis of a corpus of opinion pieces dealing with openness in computational chemistry, we have depicted how computational reproducibility issues are more complex than a naive requirement for transparency as a sufficient and necessary condition, in the name of openness. We want to make clear, here, that we do not advocate for the futility of code transparency and open source scientific software as a source of reproducibility, quite the contrary. But we do regard this view of transparency as sufficient for computational reproducibility as superficial, and even detrimental to openness.

We have introduced four epistemic characteristics (namely transparency, consistency, sustainability and inclusivity) to account for a deeper understanding of the epistemic issues at stake within the computational reproducibility iceberg, of which transparency is only the most visible part. Alongside this epistemic account , we have shown that code is only a part of a much more complex concept that is summed up into software. Informed by a history of computing that recognizes the overlook of software in computational matters until recently, we have described software as a millefeuille composed of many layers. Within the domain of computer simulations, our general claim is that entanglements among these layers make models and software inextricable, both technically and epistemically. These entanglements has hardly been tackled by philosophers of science until now. From the formal model to the code, the computing environment, the package, the licensing and the community around software, all these entangled layers constitute our software millefeuille.

To grasp the multifaceted concept of software, we built on three perspectives on how to consider software: as a scientific instrument, as a commodity, and as a social project. These three perspectives allow us to highlight these entanglements, and to highlight how the four epistemic characteristics interplay in different manners in each of these three perspectives, and to which layers they relate in each case.

In particular, we have described how these characteristics sometimes conflict one with another. For example, transparency and consistency conflict within two Weltanschuungen of a scientific instrument. Sustainability and inclusivity also conflict when discussing the required workforce around a software project. These epistemic conflicts account for the tensions that are at play in the opinion pieces in computational chemistry we used as a corpus, and more generally within the issues of computational reproducibility.

Every scientific field possesses its own history as regards its relationship to software. Computational chemistry as a longue durée case study lends itself to study precisely the nuances of

these issues and allows to go further a superficial requirement for transparency. We argue that computational reproducibility is complex, its underlying epistemic issues are diverse, and they are deeply entangled with the concept of software, an ever present elephant in the room that still has to be acknowledged in its multiple dimensions as pivotal for computational reproducibility matters.

Acknowledgements

The authors gratefully acknowledge the Science History Institute (Philadelphia) for a fellowship during which part of this research had been achieved. This research project is being supported by a grant from the MSH Lorraine, France.
We are indebted to both anonymous reviewers for their valuable suggestions, which have clearly improved, in our view, this paper.

References

AlNoamany, Yasmin, and John A. Borghi. 2018. “Towards Computational Reproducibility: Researcher Perspectives on the Use and Sharing of Software.” PeerJ Computer Science 4: e163. https://peerj.com/articles/cs-163 (December 31, 2019).

Atmanspacher, Harald, and Sabine Maasen. 2016. Reproducibility: Principles, Problems, Practices, and Prospects. Hoboken, New Jersey: Wiley-Blackwell.

Baker, Monya. 2016. “1,500 Scientists Lift the Lid on Reproducibility.” Nature News 533(7604): 452. http://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970 (December 31, 2019).

Bhandari Neupane, J., Neupane, R. P., Luo, Y., Yoshida, W. Y., Sun, R., & Williams, P. G. (2019). “Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the “Willoughby–Hoye” Scripts for Calculating NMR Chemical Shifts”. *Organic Letters*, *21*(20), 8449–8453. https://doi.org/10.1021/acs.orglett.9b03216

Benureau, Fabien C. Y., and Nicolas P. Rougier. 2018. “Re-Run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions.” Frontiers in Neuroinformatics 11. https://www.frontiersin.org/articles/10.3389/fninf.2017.00069/full (December 31, 2019).

Chue Hong, Neil, Simon Hettrick, Andrew Jones, and Daniel Katz. 2015. “The Price of Open-Source Software – A Joint Response.” Software Sustainability Institute blog. https://www.software.ac.uk/blog/2016-09-22-price-open-source-software-joint-response (December 31, 2019).

Ensmenger, Nathan L. 2010. The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise. The MIT Press.

Geiger, Stuart, Dorothy Roe Howard, Lilly Irani, Nelle Varoquaux, Alexandra Paxton, and Chris Holdgraf. 2019. “Who Pays the Costs of Free and Open-Source Scientific Software?” 4S Annual Meeting. http://tinyurl.com/y2jqs4fb (December 31, 2019).

Gelfert, Axel. 2011. “Scientific Models, Simulation, and the Experimenter’s Regress”. In Models, Simulations, and Representations, edited by Paul Humphreys and Cyrille Imbert, 145-167. Routledge.

Gezelter, J. Daniel. 2015. “Open Source and Open Data Should Be Standard Practices.” The Journal of Physical Chemistry Letters 6(7): 1168–69. https://doi.org/10.1021/acs.jpclett.5b00285 (December 31, 2019).

Hatton, L., & van Genuchten, M. (2019). Computational Reproducibility: The Elephant in the Room. *IEEE Software*, *36*(2), 137–144. https://doi.org/10.1109/MS.2018.2883805

Hey, Tony, and Mike C. Payne. 2015. “Open Science Decoded.” Nature Physics 11(5): 367–69. https://www.nature.com/articles/nphys3313 (December 31, 2019).

Hinsen, Konrad. 2014. “Computational science: shifting the focus from tools to models [version 2; peer review: 2 approved]”. F1000Research 2014, 3:101. https://doi.org/10.12688/f1000research.3978.2

Hinsen, Konrad, and Nicolas Rougier. 2019. “Challenge to Test Reproducibility of Old Computer Code.” Nature 574(7780): 634–634. https://www.nature.com/articles/d41586-019-03296-8 (December 31, 2019).

Hocquet, Alexandre, and Frédéric Wieber. 2017. “’Only the Initiates Will Have the Secrets Revealed’: Computational Chemists and the Openness of Scientific Software”. IEEE Annals of the History of Computing 39(4): 40-58.

Horner, Jack, and John Symons. 2014. “Reply to Angius and Primiero on Software Intensive Science”. Philosophy & Technology 27 (3): 491-94. https://doi.org/10.1007/s13347-014-0172-9.

Humphreys, Paul. 2004. Extending Ourselves: Computational Science, Empiricism, and Scientific Method. Oxford University Press.

Jacob, Christoph R. 2016. “How Open Is Commercial Scientific Software?” The Journal of Physical Chemistry Letters 7(2): 351–53. https://doi.org/10.1021/acs.jpclett.5b02609 (December 31, 2019).

Kelty, Christopher M. 2008. Two Bits: The Cultural Significance of Free Software. Duke University Press.

Krylov, Anna I., John M. Herbert, Filipp Furche, Martin Head-Gordon, Peter J. Knowles, Roland Lindh, Frederick R. Manby, Peter Pulay, Chris-Kriton Skylaris, and Hans-Joachim Werner. 2015. “What Is the Price of Open-Source Software?” The Journal of Physical Chemistry Letters 6(14): 2751–54. https://doi.org/10.1021/acs.jpclett.5b01258 (December 31, 2019).

Lejaeghere, K., Bihlmayer, G., Björkman, T., Blaha, P., Blügel, S., Blum, V., Caliste, D., Castelli, I. E., Clark, S. J., Dal Corso, A., de Gironcoli, S., Deutsch, T., Dewhurst, J. K., Di Marco, I., Draxl, C., Dułak, M., Eriksson, O., Flores-Livas, J. A., Garrity, K. F., … Cottenier, S. (2016). “Reproducibility in density functional theory calculations of solids”. *Science (New York, N.Y.)*, *351*(6280), aad3000. https://doi.org/10.1126/science.aad3000

Lenhard, Johannes, and Uwe Küster. 2019. “Reproducibility and the Concept of Numerical Solution.” Minds and Machines 29 (1): 19–36. https://doi.org/10.1007/s11023-019-09492-9.

Leonelli, Sabina. 2019. “Rethinking Reproducibility as a Criterion for Research Quality.” In Including a Symposium on Mary Morgan: Curiosity, Imagination, and Surprise, Research in the

History of Economic Thought and Methodology, Emerald Publishing Limited, 129–46. https://doi.org/10.1108/S0743-41542018000036B009 (December 31, 2019).

Mahoney, Michael S. (2008). What Makes the History of Software Hard. IEEE Annals of the History of Computing 30(3): 8–18. https://doi.org/10.1109/MAHC.2008.55

Miletić, Vedran. 2015. "What Is the Price of Open-Source Fear, Uncertainty, and Doubt?" Nudged Elastic Band is my band name. https://nudgedelastic.band/2015/09/what-is-the-price-of-open-source-fear-uncertainty-and-doubt/ (December 31, 2019).

Peng, Roger D. 2011. "Reproducible Research in Computational Science." Science 334(6060): 1226–27. https://science.sciencemag.org/content/334/6060/1226 (December 31, 2019).

Reinhardt, Carsten. 2001. Chemical Sciences in the 20th Century : Bridging Boundaries. Weinheim; New York: Wiley-VCH.

Reinhardt, Carsten. 2006. "A Lead User of Instruments in Science: John D. Roberts and the Adaptation of Nuclear Magnetic Resonance to Organic Chemistry, 1955–1975." Isis 97(2): 205–36. https://www.journals.uchicago.edu/doi/abs/10.1086/504732 (November 29, 2019).

Schappals, M., Mecklenfeld, A., Kröger, L., Botan, V., Köster, A., Stephan, S., García, E. J., Rutkai, G., Raabe, G., Klein, P., Leonhard, K., Glass, C. W., Lenhard, J., Vrabec, J., & Hasse, H. (2017). "Round Robin Study: Molecular Simulation of Thermodynamic Properties from Models with Internal Degrees of Freedom". *Journal of Chemical Theory and Computation*, *13*(9), 4270–4280. https://doi.org/10.1021/acs.jctc.7b00489

Spencer, Matthew. 2015. "Brittleness and Bureaucracy : Software as a Material for Science." Perspectives on Science 23(4): 466–84. http://dx.doi.org/10.1162/POSC_a_00184 (December 31, 2019).

Stodden, Victoria, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P. A. Ioannidis, and Michela Taufer. 2016. "Enhancing Reproducibility for Computational Methods." Science 354(6317): 1240–41. https://science.sciencemag.org/content/354/6317/1240 (December 31, 2019).

Symons, John, and Ramón Alvarado. 2019. "Epistemic Entitlements and the Practice of Computer Simulation." Minds and Machines 29 (1): 37–60. https://doi.org/10.1007/s11023-018-9487-0.

Wieber, Frédéric, and Alexandre Hocquet. 2020. "Models, Parameterization, and Software: Epistemic Opacity in Computational Chemistry". Perspectives on Science 28(5): 610-629.

Winsberg, Eric. 2010. *S*cience in the Age of Computer Simulation, Chicago, Ill.: University of Chicago Press.

Winsberg, Eric. 2019. "Computer Simulations in Science", The Stanford Encyclopedia of Philosophy (Winter 2019 Edition), Edward N. Zalta (ed.). https://plato.stanford.edu/archives/win2019/entries/simulations-science/

Epistemic Issues in Computational Reproducibility: Software as the Elephant in the Room

ETHICAL STATEMENT:

• Funding: (if any) : none

• Conflict of Interest: The authors declare that they have no conflict of interest.

• Ethical approval: For this type of study formal consent is not required.

• Informed consent: For this type of study formal consent is not required.

Abstract

Computational reproducibility (i.e. issues of reproducibility stemming from the computer as a scientific tool) possesses its own dynamics and narratives of crisis. Alongside the difficulties of computing as an ubiquitous yet complex scientific activity, computational reproducibility suffers from a naive expectancy of total reproducibility and a moral imperative to embrace the principles of free software as a non-negotiable epistemic virtue. We argue that the epistemic issues at stake in actual practices of computational reproducibility are best unveiled by focusing on software as a pivotal concept, one that is surprisingly often overlooked in accounts of reproducibility issues. Software is not only about designing and coding but also about maintaining, supporting, distributing, licensing, and governance; it is not only about developers but also about users. We focus on openness debates among computational chemists involved in molecular modeling software packages as empirical grounding for our argument. We then identify and analyse four epistemic characteristics (transparency, consistency, sustainability and inclusivity) as key to the role of software in computational reproducibility.

Keywords

computational reproducibility, software, computational chemistry, transparency, consistency, sustainability, inclusivity

Article

1. Computational reproducibility

Is there “a growing sense that science has reached a reproducibility crisis” (Getzelter, 2015)? The tension between reproducibility as a general moral imperative of “the core principles of science”, and the acknowledgment in almost every quarter of scientific domains that reproducibility indeed

poses many problems, is key in the feeling that reproducibility in science might be in crisis (Baker, 2016). More often than not, this crisis is addressed in terms of epistemic transparency as a necessary and sufficient condition for science to be reproducible, in the (vague) name of open science.

Through the narratives of a reproducibility crisis in science throughout the last decade, there have been attempts to classify reproducibility into categories (for a review, see Atmanspacher and Maasen, 2016). The so-called "computational reproducibility" emerged as one of them. Computational reproducibility is naturally linked to the computer as a scientific tool: it "generally refers to the description and sharing of software tools and data in such a manner as to enable their use and evaluation by others" (AlNoamany & Borghi, 2018, p. 3). The notion has gained momentum because of the pervasiveness of the computer within scientific activity. In these narratives, computational reproducibility as a category is often contrasted with what is often referred to as experimental reproducibility. Computational reproducibility is, in this context, often naively perceived as the part of reproducibility that could (and should) be achieved exactly: "to re-run a computation however many times as wished for, in whichever context, without changes"[1].

Philosopher of science Sabina Leonelli, unsatisfied with an overarching definition of reproducibility across scientific disciplines, has convincingly defined six categories of reproducibility varying in scope and space (Leonelli, 2019). Her aim is to assert that scientific fields define reproducibility in diverse ways and that the very relevance of the concept of reproducibility is itself variable. The first of Leonelli's six categories is named "computational reproducibility" and is, according to her, the only one concerned with total reproducibility of datasets (as opposed to the plain reproduction of patterns or even inferences). Computational reproducibility is, in Leonelli's words, the only one that is agnostic towards the circumstances of data production. In this regard, Leonelli, in line with her description of data-centric science, suggests a view of computation as an activity that is limited to data treatment. Many accounts of reproducibility conform to this view when it comes to the computational part of reproducibility (see for example Peng, 2011). Computational reproducibility is defined in those accounts as wearing the burden of a requirement of total reproducibility. We are sympathetic to Leonelli's categorization because it offers a nuanced depiction of the reproducibility landscape in science, and we acknowledge the importance of computational reproducibility as a category of its own. Nevertheless, we argue that the landscape of computational reproducibility is itself more complex.

Practitioners such as computer scientists, computational scientists or software curators express concerns about computational reproducibility. For example, Benureau & Rougier (2018) consider that problems of replicability in computational software exist precisely because "it is easy to believe that if a program runs once and gives the expected results it will do so forever" (Benureau & Rougier, 2018, p. 1). AlNoamany & Borghi (2018) observe that in discussions about reproducibility, emphasis is most of the time put on data-related scientific practices but not sufficiently on software practices. Yet, computational reproducibility poses specific problems, such as "the lifetime reproducibility of a particular code" (Hinsen & Rougier, 2019, p. 634), or "the lack of transparency in disclosure of computational methods" (Stodden et al., 2016, p. 1240). What is at stake is thus not only to be able to reproduce data, but also to focus on the disclosure and sustainability of computational methods used to produce data in the first place.

[1]As per the wording of an anonymous reviewer.

To a certain extent, advocating that computational reproducibility issues are important is a way, for some practitioners, to stress that software has to be taken into account within the Open Science movement, along with publications (as in Open Access) and data (as in Open Data). As Hey & Payne (2015) write: "[The] global momentum towards ‘open science’ […] necessarily requires not only open access to research publications, but also to the metadata and data required to validate and make sense of the results of research. And improving the comprehensibility and reproducibility of computational science is an important step in this endeavour [by offering] access to the *software*, data and computer environment used to produce the results." (Hey & Payne, 2015, p. 367, our emphasis).

Thus, computational reproducibility cannot be reduced to reproducibility concerns associated with open or closed literature, even if they are linked to matters of publication's practices (like for example the availability of the code used within a scientific paper). Nor can it be reduced to reproducibility concerns associated with closed or open data, even if they are linked to issues of data disclosure (like for example the minutiae of model parameters used in a calculation). A common statement made by most practitioners discussing computational reproducibility is that computational methods have led to significant changes in actual scientific practices. Yet, they complain that the disclosure of models, computational steps and computing environment used to produce data is insufficient in the published scientific literature to achieve reproducibility. “A detailed understanding of what a given piece of software does is often limited to the software’s authors”, as Hinsen (2014, p. 2) summarizes. He even goes further as, for him, “there is no clear separation between the tool (software) and the model it operates on”. Hinsen considers that computational scientists should strive to explain to their peers the computational models and methods they use, because “scientific software is much too complicated to be an efficient way to communicate these models and methods” (Hinsen, 2014, p. 2). Computational transparency is not that easy to achieve, indeed.

Whether they are writing, using or reviewing scientific software, actors express their dismay about how to achieve reproducibility in practice. We reckon that the issue of computational reproducibility linked to software is complex and we aim to address this complexity, beyond the mere requirement for code transparency. To this end, we argue in this paper that epistemic issues beyond transparency are key to the role of software. In addition to transparency, we offer a categorization of three more epistemic characteristics (namely consistency, sustainability and inclusivity) to depict a more complex epistemic landscape. We also argue that this complexity is to be understood within the entanglements of the different layers of what we call a software millefeuille, beyond its reduction to “code”. To put it bluntly, software, even though ubiquitous in the practices, is surprisingly missing in the analyses, like the elephant in the room (Hatton and van Genuchten, 2019).

2. Computational Science and Software

Our way to deal with computational reproducibility is to focus on an important subpart of computational science, one that is concerned with calculations based on scientific models translated into computer programs. This activity of modeling as well as simulations has received considerable

interest from philosophers of science. It even possesses its own "epistemology of computer simulations" (Winsberg, 2019, 2010).

One of its features, according to Winsberg, is that computer simulations are "motley" in the sense of that they may depend not "just on theory but on many other model ingredients and resources as well, including parameterisations, numerical solution methods, mathematical tricks, approximations and idealizations, outright fictions, ad hoc assumptions, function libraries, compilers and computer hardware, and perhaps most importantly, the blood, sweat, and tears of much trial and error" (Winsberg, 2019, part 4.1). Procedures of validation (between theory and model) and verification (between model and program) are not that easily discernable, according to Winsberg. For instance, discretisation techniques, as a classical example of epistemic trade-offs, rely more on engineering and programming practices than on pure formalization of theories. In the same vein, Lenhardt and Küster (2019) explicitly express computational reproducibility concerns in terms of this kind of imperfect "numerical solutions". Building on this acknowledgment of the entanglement of engineering practices and formal theories, Symons and Alvarado (2019) discuss the concern for trust in computer simulations. They argue that "computer simulations and computational methods in general—as instruments deployed in scientific inquiry—are neither reliably transparent conveyors in all contexts, nor can they be regarded as equivalent to expert sources of testimony" (p. 47). "Thus, when computer simulations can be trusted it is because of their adherence to theoretical principles, empirical evidence, or engineering best practices and not because of their output alone" (p. 52). Winsberg (2019) adds that "the credentials [of computer simulations] develop over an extended period of time and become deeply tradition-bound. In Hacking's language, the techniques and sets of assumptions that simulationists use become 'self-vindicating'. […] the locus of our trust in simulations [resides] in practical aspects of the craft of modeling and simulation, rather than in any features of the models themselves" (part 4.2). These descriptions and analysis of computer simulations clearly emphasise the complexity of computational methods. They show that to trust a computational simulation, one needs more than the validation of a simulated model.

Some authors express this idea in terms of software. Gelfert (2011) distinguishes two aspects concerned with computational reproducibility. First, so-called "software" aspects involve the implementation of a model into a computational template[2] that makes it computationnally tractable. Gelfert considers that "this stage of setting up simulations often involves tacit know-how on the part of the investigator and is rarely fully documented" (p. 154). He then adds that "replicability in the case of computer simulation, however, also has a 'hardware' component, […] [that] concerns the further problem of the reliability of the concrete device, on which a simulation is being run" (p. 154). Focusing on the influence of software on these reproducibility matters, Horner and Symons (2014) posit that "the high conditionality of typical [scientific] software […] implies that [...] there is no known effective method for characterizing […] the error distribution in software" (p. 491), thereby asserting the difficulty and specificity of the task.

Thus, engineering practices of programming, tacit know-how and entrenched craft, software error management, implementation on concrete devices highlight issues that belong to the realm of software. The requirement for transparency as a simple way to achieve total reproducibility of computational methods, either at the level of computational templates or within code, appears then

[2] On computational templates and computational tractability, see Humphreys (2004).

naive. Yet, we consider that software, in those accounts (as algorithms in the case of Gelfert, or as programming in the case of Symons and Horner), is viewed as too narrow and not sufficiently historicized. Building on these works, but also on the field of the history of software, we thus aim to expand on the multifaceted nature of software to better apprehend the complexity of computational reproducibility issues, as a simplistic moral imperative for more transparency may be not enough.

As the entanglement between all these aspects is of tremendous importance for the issue of computational reproducibility, we are choosing, as a case study, the field of computational chemistry to shed light on this entanglement. Computational chemistry is an interesting computational science, with peculiarities concerning software. Because of its particular historical context of development, because of its proximity to the pharmaceutical industry, computational chemistry has had to deal with academic norms as well as business norms (Anonymised, 2017). Software packages, in this context, are devised to produce novel scientific results. It is also important to acknowledge that they are distributed, maintained and licensed under these sometimes conflicting norms. We thus argue that computational chemistry is an interesting field to engage in, so as to articulate multiple dimensions of software within the problem of computational reproducibility.

3. Computational chemistry as a case study

Computational chemistry is a scientific community involved in designing models to explore the physico-chemical properties of molecules (and materials). It is a field where reproducibility concerns are expressed in numerous “benchmarks” to assess the validity of computer simulations. These “benchmark” publications (some of them contain the word “reproducibility” in their very title) strive to compare scientific models such as density functionals (Lejaeghere et al. 2016) or molecular dynamics force fields (Schappals et al., 2017). These benchmarks include their mathematical expressions, the parameterization of these expressions, their translation into code, and their embedding into software packages. As an extreme example, a recent publication has even highlighted a Nuclear Magnetic Resonance (NMR) chemical shift calculation that depends on the operating system the software is installed on! (Bhandari Neupane et al., 2019)

Our way to study computational chemistry is to focus on application software designed by computational chemists to model physico-chemical properties, a genre of software designed within the community and for the community. While some of the members of this community do program software as developers, others are only users of these application software packages, either in Academia or in the industry. Some even sell or market software packages. The profiles in this community are thus extremely diverse when it comes to using or developing software. It is also notable that computational chemistry has emerged in times of mobilization of American universities to produce innovation and was involved with two major industries: the computer manufacturers and the pharmaceutical industry, the latter becoming a potential market for the former through molecular modelling software packages. In this specific context, tensions between academic norms and business norms arise (Anonymised, 2017).

In another publication (Anonymised, 2020), we have argued that opacity in computational chemistry models lies in different layers: in the parameters of the mathematical formalization of the model itself, in the code into which these models are translated, in the packages where these

implementations are embedded, in the licensing policies that define how software is shared and used, and in the communities of developers and users. We have also argued that these layers are entangled and not easily separated. Our point is that models and software have to be addressed together: beyond the entanglement of theories and motley models, and models and code, we argue that other dimensions of software are to be studied.

Hence, our aim in the present paper is to move a step further by shedding light on the incompleteness of expressing reproducibility in terms of mere transparency of code. To better comprehend the complexity of computational reproducibility, we describe different layers of what we call a software millefeuille. We build on this description to depict three more epistemic characteristics (namely consistency, sustainability and inclusivity) beyond transparency.
As empirical grounding for our argument, we are focusing in this paper on a timely series of five opinion pieces evolving into an asynchronous debate within the computational chemistry community. They have been written and published in 2015 and 2016, and they explicitly refer to a "reproducibility crisis". The articles have been published in "Journal of Physical Chemistry Letters" for three of them and in blog posts for three of them (one was a blog post eventually turned into a paper). We make use of these five papers and we also at times refer to debates within the Computational Chemistry List along the years, especially the so-called 1993, 2001 and 2011 flame wars we have analyzed before (Anonymised, 2017).

The first paper (Gezelter, 2015) has a self-explaining title: "Open Source and Open Data Should Be Standard Practices". Gezelter, as a computational chemist and an Open Science activist, argues for code sharing as desirable transparency, especially for peer reviewing, encourages versioning to improve code sustainability, and laments the lack of academic recognition for the activity of developing software. It is followed by a paper by ten co-authors (all involved in the development of commercial software packages among the most renowned in their field) entitled "What is the price of open source software?" that counters Getzelter's arguments (Krylov et al., 2015). The aim of this reply is to stand up for their business and epistemic model, and to question the feasibility and sustainability of open source software in computational chemistry. This second paper is in turn replied to by a blog post co-authored by four ("The Price of Open-Source Software – A Joint Response") (Chue Hong et al., 2015) and another paper by Jacob ("How Open Is Commercial Scientific Software?") (Jacob, 2016) that mitigate both views. A fifth blog post by Miletic, "What is the price of open-source fear, uncertainty, and doubt?", criticizes Krylov et al. from the viewpoint of libre activism (Miletic, 2015). Chue Hong and his co-authors are software engineers and work in the field of scientific software sustainability. Jacob is a computational chemist and part of the development team of yet another commercial package. Miletic is a contributor to a decentralized open source package project. Throughout these five pieces, typical software and epistemic issues related to computational reproducibility are discussed, a diversity of viewpoints are opposed, arguments are put forward, countered and debated.

4. Software and epistemic issues of reproducibility

Software is a difficult topic to grasp, first of all because its very definition is elusive. As historian of software Nathan Ensmenger puts: "Although the idea of software is central to our modern

conception of the computer as a universal machine, defining exactly what software is can be surprisingly difficult" (Ensmenger, 2010, p. 7).

In many accounts, software and code are interchangeably used, yet software encompasses many dimensions that go beyond what is understood as code. Ensmenger summons the notion of socio-technical object to circumscribe these many dimensions. "Unlike hardware, which is almost by definition a tangible thing that can readily be isolated, identified, and evaluated, software is inextricably intertwined with the larger sociotechnical system of computing that includes machines [...], people (users, designers, and developers), and processes [...]" (Ensmenger, 2010, pp. 7-8).

To address computational reproducibility, we argue that a focus not only on the design of programs but also on the implementation, distribution, maintenance, support and uses of software is necessary. For an operational assessment of the different yet entangled layers of the software millefeuille, we will in this paper try to stick to a categorization that distinguishes between the model, the code, the computing environment, the package, the licence and the community, even though each of these categories are not impermeable, and even though other phrasings exist to designate layers within or in between those categories, such as methods, algorithms, or programs. We are trying here to design an operational categorization to highlight the diversity of layers at stake.

We thus are choosing the term *"model"* as encompassing the methods for which scientists envision the possibility to compute "numerical solutions" out of them. By *"code"*, we mean the translation of those models into a language that a computer comprehends. Both layers are entangled. *"Computing environment"* refers to the interactions of code with other computer layers such as operating systems, or libraries dependencies, or the compiling into an executable. It is also entangled with the previous layers because some hardware evolution like parallelization or the use of graphical processing units need reassessment of programming and calculation techniques. *"Package*" illustrates that part of software that is designed as something to be distributed. It might embed several different scientific models into an assemblage of code sub-units, possibly tied together within a user interface, and possibly evolving into a full software suite. Support and maintenance, when they exist, apply mainly to this package layer[3]. The *"licence"* is the contract that links the package and its users. It defines under which conditions the package is distributed and used. Finally, the *"community"* embraces scientists, developers, programmers, maintainers, professionals, possible marketers and sellers, but also users. Users are in turn potentially further divided into categories such as lay users or lead users.

We argue that not only transparency is difficult to achieve indeed throughout this software millefeuille, it is also not enough to account for the reproducibility problems at stake. We thus introduce three other epistemic characteristics, namely consistency, sustainability and inclusivity that are distinct from transparency and yet play a significant role in computational reproducibility issues.

[3]We thank a anonymous reviewer for suggesting "documentation" as yet another layer. While we agree that a variety of epistemic concerns may arise from such a layer, we have chosen, based on our case study corpus, to account for the documentation layer within what we have named the "package" layer.

We define *"transparency"* as the ability to check how a tool functions and what it does. As far as software is concerned, transparency is obviously linked to open-source code as a commonly-viewed necessary prerequisite to any reproducibility attempt. Yet, transparency extends well beyond the scope of open-source code as we clarify further down in this paper. We define *"consistency"* as the ability to be certain about which tool is actually used in which computing environment, transparent or not. Transparency may not be enough for replication, if one cannot ascertain which transparent tool is used in which computing environment, and therefore ensure that said tool does what it claims it does. Consistency is linked for example to the practices of versioning, or to software dependency problems. We define *"sustainability"* as the guarantee that the tool works the same way over time, or with different hardware configurations, or with different compilers. Sustainability is linked to support, maintenance but also portability. Sustainability is an essential matter for results to be replicated over time or different computing environments, something that transparency and consistency alone cannot offer. We define *"inclusivity"* as the ability for anyone to use, benchmark or develop the tool. Transparency, consistency and sustainability may be not enough for replication if some people, but not all, are actually allowed, or empowered to test or try to replicate. Evidently, these characteristics are interrelated. Yet, they are also at times conflicting, and we argue that this categorization is useful to illustrate the tensions within computational reproducibility. Consistency, sustainability and inclusivity are in a way the hidden part of the iceberg of computational reproducibility. Beyond the sole transparency, taking those four characteristics into account altogether allow for a deeper understanding of the epistemic complexity of computational reproducibility.

In the following sections, we describe three different perspectives about how the concept of software can be grasped in the scientific milieu. First of all, software is akin to a scientific instrument, a tool that is used by scientists to perform scientific calculations. Secondly, as scientific software development norms are at times at odds with academic norms, software is also a business model, being actually either pursued or criticized as such. Finally, software introduces a social dimension. First, development involves many interacting humans. Software also defines a relationship between users on the one hand and developers on the other. This social dimension raises the bureaucratic question of governance as a reproducibility matter.

In these three different perspectives, the way the four previously defined epistemic characteristics are interrelated is diverse, bringing a kaleidoscopic view of computational reproducibility epistemic issues. We will now assess software as instrument, software as distribution and software as governance in terms of these four epistemic characteristics linked to the six layers of software millefeuille (model, code, computing environment, package, licence and community), in order to stress how computational reproducibility is impacted.

5. Software as Instrument

Scientists construct models in ways that are linked to the phenomena their models try to represent, to the theories they can use, but also to the technological, professional, economic and political context in which they work. These epistemic roots are a pivotal step to understand how the models they construct are then translated into software, and how these models influence the actors' discourses and the tensions at stake among them. As the historian of software Michael Mahoney

writes, computational scientists “have put their portion of the world into the computer” (Mahoney, 2008, p. 8). These models, and their translations into software, are “operative” representations. “Through the computer, we can manipulate them, and they in turn can trigger actions in the world” (Mahoney, 2008, p. 13).

In a publication describing the designing of a piece of software within a computational fluid dynamics academic laboratory, Spencer (2015) points out this performativity: “The focus on the framework, on keeping it workable, portable, durable, arises from the intersection of the properties of expanding software with its conditions of application as a particular kind of medium for research”. Scientific software thus shares many aspects with scientific instrumentation. It is a tool, designed by scientists for scientists, part and parcel of the scientific lab.

Typically, the actors of our corpus view software as a complex scientific instrument. The Nuclear Magnetic Resonance (NMR) apparatus is a frequent metaphor they employ. Not only NMR and molecular modelling are the bread and butter of many chemistry labs instrumentation, the metaphor is also used because NMR is exemplar of modernity and complexity in the chemistry lab, at the crossroads between Big Science and Entrepreneurship Science, as narrated by Reinhardt (2006).

Two different, almost opposed, Weltanschauungen regarding scientific instrumentation are referred to in the papers of the corpus we explore. In the first one, the instrument is home-made. The practitioner knows the instrument well. The trust of the community is based on the *transparency* of the instrument: it is open for all to see. The claim here is that transparency is the epistemic virtue that entails reproducibility. Failing to achieve transparency leads to the instrument being accused of being a so called "black box", i.e. an instrument where it is not known what is going on "under the hood".

In the second one, the instrument is manufactured by a corporation. The trust of the practitioner and the community is based on an industrial commodity, that is supposed to be calibrated, standardised and even liable. The corporation is in charge of providing a reliable instrument. It is not challenged to open the black box. Reproducibility is here based on the stability and dissemination of a (industrial) standard. Failure to warrant it leads to issues of *consistency*, as in the proliferation of non-certified versions of the instrument, and *sustainability*, as for example in the lack of its maintenance.

Of course, these are ideal types: software, as instruments in general, may belong to an intermediate category, or be as complex as to mix parts belonging to each category. Instruments may even share characteristics of one category and the other at the same time (for a review of epistemic instrumentation issues in chemistry, see Reinhardt, 2001). Yet, these two ideal types function in the narratives of the actors to oppose two Weltanschauungen of how reproducibility is built.

In our case study, the two types are exemplified in advocacy for open source software vs proprietary software. The proponents of open source software as an imperative typically argue that transparency should be a must for "taxpayers funded" science. Science as a public good should not be enclosed into proprietary software. Transparency must therefore be achieved by publication of the details of the *model*, the source *code* being open (and intelligible), accessible to reviewers, to users, or to any "taxpayer". According to whom it is actually transparent, *inclusivity* concerns are here at stake.

These arguments are countered by Krylov et al. (2015) by referring to the complexity of a quantum chemistry program. The metaphor of the "phone app" is used to describe what in contrast "complex quantum chemsitry code" is: it is slow to write, develop, maintain and support and the community of contributors is small, akin to an NMR apparatus, whereas the "phone app" can afford to be "free" (the narrative is replete with confusion between free beer and free speech) because the phone app, in their words, is easy to write, is disposable, and the community of contributors can be large. Therefore, proprietary licensing is used by "complex quantum chemistry code" developers to generate revenue that allows to finance the required workforce to develop, maintain and support the software *package* in order to achieve *sustainability* and *consistency*. Reproducibility is to be attained thanks to this proprietary financing.

Moving beyond the open/proprietary dichotomy, the well known (in the field) Gaussian *package* is subject to a similar tension. The company is priding itself on providing a readable source *code* (to allow *transparency*), yet it forbids its compiling with unsupported compilers. It does so to avoid the proliferation of "suboptimal" versions of the official *package*, expressing concerns of *consistency*. It also does so to avoid the portability to unsupported hardware, thus expressing concerns of *sustainability* regarding the *computing environment*. The debates around this very tension have been analysed in Anonymised (2017). As Chue Hong et al. (2015) point out, the mere publication of equations, parameters or algorithms or even source code is not enough to warrant reproducibility. The mere compiling of a piece of software or the dependence on evolving libraries may jeopardize the reproducibility of a calculation. Hence, epistemic characteristics are clashing here: *transparency* on the one hand, as expressed by the fact that a readable source code may not be transparent enough for reproducibility of calculations, and *consistency* on the other, as expressed by the fact that a modifiable source code may lead to a proliferation of versions that hamper the consistency of results.

6. Software as Distribution

Software is also what turns code into an activity linked to the rest of the world, sometimes an industrial one. Nathan Ensmenger argues that software, as the interface between computer and society, defines our relationship to the computer. “Software sits uncomfortably at the intersection of science, engineering and business, a heterogeneous technology that blurs the boundaries between the technological and organizational, fraught with disputes over conflated social, political and technological agendas” (Ensmenger, 2010, p. 8).

In the professional history of software, the phrasing "software crisis" describes a major disenchantment that occurred soon after software was envisioned as a business opportunity: in a nutshell, from the late nineteen sixties onwards, the computing industry realised that while the performance of hardware went up and its cost went down, the software followed the exact opposite trend, precisely because of the increased difficulty to keep up with the hardware development pace, with increasing demands for workforce, for investment and for technology, especially in the domains of maintenance and support more than the actual design or programming. More than a turning point, the "software crisis" has been known more as an art-de-vivre ever since (Ensmenger, 2010). In his description of the academic developing of a piece of software, Spencer (2015) similarly emphasises on the tensions introduced by time: “Both brittleness and the accumulation of

supporting processes make demands on time, whether it is time spent figuring out how to make changes or fix bugs, or time spent adhering to new administrative routines for tracking changes and logging bugs".

Software has also to be understood as something to share or distribute. More than fifty years ago, scientists began to write programs to be used by others. "Market-oriented" scientific software thus shares concerns that are similar to the general software industry. Even if scientific software is not necessarily a commercial product (and it sometimes is), it is prone to commodification. Miletic (2015) argues that the limit of the NMR instrument metaphor shows when distribution issues arise: from a certain open point of view, the sharing of a piece of software is unlimited. Yet, issues of (lack of) academic recognition regarding the labour of programming can be found in scientific fields where software is a tool and not the object of research itself. The notion of a (lack of) "business model" is ubiquitous in actor phrasings, exemplifying the tensions between academic norms and business norms.

Business models

Software developing oftentimes is an unrewarded scientific activity within Academia. The traditional publication reward of scientific activity fails to account for the amount of work, energy, time and money invested in developing a scientific method turned into a piece of software. It is an acute problem in computational science because developing, and even more so maintaining, is barely the core activity of computational scientists, in contrast to computing scientists. In this regard, computational scientists are often portrayed as "bad programmers". Many computational reproducibility issues are summed up into a lack of sound programming practices (AlNoamany & Borghi, 2018). In the same vein, the lack of incentives and rewards, or mechanisms of recognition for software developing (quite similarly to data curating) is a well known problem mentioned in computational reproducibility pamphlets (Stodden et al., 2016). What is much less mentioned however is the fact that a "sustainable business model" is one of the way computational scientists try to address the issue.

Not unexpectedly, proprietary software is criticized by open source promoters. Not only do proprietary *packages* turn software into epistemically opaque scientific instruments that lack *transparency*, Miletic also points out that they trap their users-customers in a "vendor lock-in" situation: price is not negligible, users cannot benchmark, *portability* is discouraged, and above all, users are restricted to only make use of the scientific methods that are validated by the software corporation. The market is such that distributors (industrial or academic turned into companies) embed several (but not all) molecular modeling methods into competing software *suites* that are mutually exclusive in terms of the choice of *models*. Access to the tools is thus restricted, expressing an issue of *inclusivity*.

Even though it is common to find arguments in favour of alternative business models in the realm of general open source software (like paid services, assistance, support, maintenance...), it is much more difficult for a "complex quantum chemistry code" with a limited community of developers and users, in contrast to the (in)famous phone app. Actually, even the most hardcore promoters of open source in our case study acknowledge the need for a sustainable business model. Still, their suggested business models are vague or surprising (like a proprietary GUI around an open source

engine, for example). The “cost of open software” in general is actually an issue that is gaining attention (Geiger et al., 2019).

Proprietary packages rely on a professional workforce to ensure "strict coding guidelines" (in Krylov et al. words), support and maintenance (at least in their rhetorics). Linked to the issue of industry, recognition, market and workforce, they express a concern about the reliance on interns, graduates or post-docs to code (an activity that is detrimental to their career), de facto transforming junior researchers into the lumpenproletariat of scientific software in "free software" development. The portability of the *package* into diverse pieces of hardware, the rewriting of the *code* with regard to vectorisation, parallelisation or GPU processors, are issues of *computing environment sustainability* that, they argue, can only be addressed by a (well paid) professional team of programmers. Yet, it is argued on the opposing side, that this labour is also part and parcel of the computational scientist toolkit. Such understanding, leanrt by experience, of the entanglements of model and code are empowering the coding scientist. The division of labour is thus criticized as an issue of *inclusivity*, since it lowers the number of people able to delve into these entanglements.

It is also argued that equating *transparency* of software as reproducible good practice with free software principles may lead to conceptual dead ends. As scientists turned into software developers commonly fear for the stealing of their code (precisely because this activity requires a lot of investment and gives only minuscule rewards in return), some are wary to give away their code to the reviewer, even though the code is supposed to be licensed under an open licence. Delaying for as long as possible the sharing of ones code is a technique used to minimize the dangers of being stolen of a cutting edge scientific tool in a competitive scientific world. The actual practices of publishing and reviewing in competitive research are surely key here, and this participates in a wider wariness of scientists towards the publishing process. In contrast to code commenting as recommended sound research practices that allow intelligibility and thus *transparency*, the practices of code obfuscation do exist as a perverse technique to make an open source code *less* intelligible, in the same competitive outlook. The notion of an open source *code* is thus not as straightforward as it seems.

Licensing policies

Licensing is what permits the distribution of software from developers to users under some form of law, by defining the contractual obligations of both parties. In that sense, business models are materialized into sharing practices by licenses. Licenses are also a way to materialise what it means to be “open” when referring to free software as a role model for open science. Licensing is a core practice of software developing and it is, in particular, fundamental in order to be able to embed the principles of free software into an actual package. Yet, it is frequently overlooked. A lot of scientific programs are "given away for free" without any licensing considerations, thus leaving in the dark the actual distribution strategies of the program owner (AlNoamany & Borghi, 2018).

The Gaussian software package example that we mentioned earlier allows to shed light on a blind spot in the open source narrative for reproducibility: it is often argued that the source *code* should be available, and the weakest sense (open as available to read) actually is problematic, not only because trying to read a source code raises its own issues of intelligibility, but also because, available to read may be not enough. If a source code is readable but not available to reuse (modify,

compile, benchmark...), then its "openness" is debatable. Since *transparency* as a virtue of reproducibility often refers to the commitments of free software as a role model to follow, it is surprising that so little case is made for reuse as an "essential freedom". The tension here is between the readability of the source *code* (to ensure *transparency*) vs the control of the stability of the *package* and its versions (to ensure *consistency*).

On the other hand, the issues of licensing may be more complex than this Manichean view. Consider, for example, the parameterization of modelling methods. It is a fundamental step in the modelling process. The readability of parameters may be hidden away in the *code* (thus affecting *transparency*)**,** but the licencing policy has also a role here, especially when it discriminates between academic and industrial users. Industrial users are for example sometimes given the possibility to alter modelling parameters (for a higher licensing price). The industrial license thus allows more *transparency* that the academic one, because the user can actually check and benchmark parameters (the black box is open). Yet, actors argue that published calculations from industrial licensees may have used altered parameters, thus revealing a problem of *consistency* in the *model* being used for the calculations, because of the then possible obfuscation of which modelling parameters have been actually used in a calculation (Anonymised, 2020).

7. Software as Governance

Software is also a community of developers and users. The work of anthropologist Chris Kelty has focused on the practices of free software communities (Kelty, 2008). Even though scientists involved with software adopt a diversity of standpoints regarding what is open or not in software and how important it is, the cultural influence of the free software community shows in the debates around scientific software. It certainly has an influence on how software is tackled in computational science. Among the core practices of free software communities that Kelty (2008) defines (such as sharing source code or defining licensing policies), one of them is organising communities around the development and use of software. Light is thus shed on the importance of social aspects on software matters in scientific activity, and their materialization into bureaucratic policies of development. Defining licensing policies (as seen earlier) and organising bureaucratic forms of governance are essential in these communities, according to Kelty, and these “core practices” are also salient in the computational scientific communities. As Spencer (2015) reminds us, “tendencies towards […] bureaucracy could be discounted as merely practical, having little bearing on the legitimacy of discourse. But they play a major role in conditioning the investigations through which such discourse is generated in the first place. They are the geological processes shaping the landscape of possibilities that scientists navigate when they put their techniques into action”.

Another way of addressing the governance of software projects is to focus on the interactions between developers and different groups of users. As “market oriented” software is distributed to outsiders, and at times customers, the relationship between developers and users as conceptualised by innovation theorists, such as von Hippel, becomes relevant for reproducibility matters. Not unlike the "lead users" in NMR as described by Reinhardt (2006), borrowing from Von Hippel's theory of user driven innovation, different categories of users are emerging or disappearing around software projects. These dynamics exert influence over reproducibility, especially on the issue of

who is able to do what, who contributes to software development and how, which users are empowered and how.

Development Teams

The issue of recognition when contributing to a piece of software shines a light on the collective aspect of software contribution. Publication-like processes define how authorship and contributorship to software are addressed and this situation sometimes generates frustration in academic circles. Yet, beyond the issue of who deserves reward for software and how, the actual practices of working together towards designing, developing, distributing, maintaining and supporting a piece of software raises collaboration issues through the critical bureaucratic issue of governance. In the software industry, and especially in open source projects, the vitality of the community of contributors is key. Not only the workforce must be sufficient and sustained (and this is not obvious if the project is "more complex than a phone app"), the issue of governance is influential for reproducibility matters, when it comes to *inclusivity* and *consistency*.

Business competition as a driving force for scientific software improvement is logically criticized from the point of view of open source advocates. The complexity of quantum chemistry programs forces anyone willing to start a project to reinvent the wheel because basic computational subroutines are withheld into proprietary packages. The modularity of those *packages* could be used to collaboratively create better software and this prospect is impaired by the competitive stance of proprietary licences. Competition hinders here the *inclusivity* to access code subroutines or libraries that constitute state-of-the-art routine procedures.

We have mentioned earlier the question of division of labour. Professionalization is regarded as a warrant of *sustainability* and *consistency* for a software package on the one hand. On the other, empowerment of scientists is argued to lead to *inclusivity* as they are then able to both act on models and to write code. Among the issues of "the cost of open software", the community of contributors (its size, its vitality, its governance) is part of a "business model" and "recognition" issue. Unless the software package gets huge public funding (and it sometimes does), it is often difficult to get a team to extend beyond one research group, to form a lasting workforce beyond interns, and to rely on other governance forms than counting entirely on one principal investigator. If the software *package* project is open source, then another governance issue may raise yet another problem: the forking of a project, a typical practice from the open source software world, may be a dividing threat for a community. Open source repositories are full of corpses of software projects that did not survive the withering of its community. It is a reproducibility issue because of the subsequent lack of *sustainability* of a project if a community fades away. It can also be an issue because of the proliferation of software versions that blurs its *consistency*. Spencer (2015), observing a team of computational fluid dynamics, has described the collaboration difficulties of contributing to a local piece of software, and how the developing team tried to address those bureaucratic issues by importing organisational practices from the free software world. Hey and Pane (2015) describe a successful example with the software involved in acquisition and treatment of data at the Large Hadron Collider, where all the worldwide community of high energy physics gathers in one place (and gets enormous funding). Those bureaucratic processes are a way of achieving *consistency* and *sustainability* by organising collectively the social rules of the versioning

and maintenance around a *model* embedded in a software, at the scale of a laboratory (Spencer, 2015) or an entire community (Hey and Pane, 2015).

User-Developer Relationship

We have already described how the advocates of proprietary software argue that they enhance the reliability of their packages with the hiring of professional software developers. But we have also hinted that the actual choice of scientific *models* embedded in *packages* is also pivotal: if the software is in a dominant market position, it has the power indeed to define what scientific *model* will be used by the community of its users-customers. To be part of a developing team (not only the actual coders, but also the scientists involved in the design of the models embedded into the suite) is therefore a strategic choice for the promotion of a model from a research team, and thus for academic careers. Krylov et al. (2015) argue that, while the source code of one of their software suites is not available to the whole community, it is indeed available to the community of partners of the project, a community that can be seen as a kind of lead-users (as defined in Reinhardt, 2006). They call this community an "open teamware" (Krylov et al., 2015) and argue that this selection process is actually what makes the community vibrant. Their source code is also available to its user base, upon request. Jacob, in response, points out that the "on demand source code" is in fact subject to the signing of a disclosure agreement that is dictated by the corporation. Suggestions of improvements to the software suite from “lead-users” research groups are traded in exchange of their surrendering of property rights. This situation results for these lead users in the hampering of potential external collaborations with other scientists as a consequence of those agreements. These mechanisms of *inclusivity* and exclusivity and the associated governance dynamics are thus complex, and are criticized as gatekeeping, and perpetuating power positions (Jacob, 2016). It is striking that the co-authors of the Krylov et al. paper are for the most part well established scientists. They have managed to make a software *suite* prosper along decades, and, as oftentimes in self-made meritocratic narratives, they have inherited this situation by being the students (or the students of students) of founding pioneers. A software suite in this regard is very much akin to dynasty and the stakes of being in or out the contributing team resembles playing the role of a courtier.

Thus, in a Weltanschauung where users are customers, the actual access to *models* poses a problem of *inclusivity*. Moreover, beyond the theoretical possibility of access, the implementation or not within the *package* of so called "user-friendly" empowering interfaces to access the parameterisation of the model, or even the portability to different hardware or operating systems, is a way of defining which user can use the tools (and thus who is in principle able to reproduce the scientific results produced with that tools). In this regard, it is another expression of the lock-in problem that hinders reproducibility.

8. Conclusion

Throughout our analysis of a corpus of opinion pieces dealing with openness in computational chemistry, we have depicted how computational reproducibility issues are more complex than a naive requirement for transparency as a sufficient and necessary condition, in the name of openness. We want to make clear, here, that we do not advocate for the futility of code transparency and open

source scientific software as a source of reproducibility, quite the contrary. But we do regard this view of transparency as sufficient for computational reproducibility as superficial, and even detrimental to openness.

We have introduced four epistemic characteristics (namely transparency, consistency, sustainability and inclusivity) to account for a deeper understanding of the epistemic issues at stake within the computational reproducibility iceberg, of which transparency is only the most visible part. Alongside this epistemic account , we have shown that code is only a part of a much more complex concept that is summed up into software. Informed by a history of computing that recognizes the overlook of software in computational matters until recently, we have described software as a millefeuille composed of many layers. Within the domain of computer simulations, our general claim is that entanglements among these layers make models and software inextricable, both technically and epistemically. These entanglements has hardly been tackled by philosophers of science until now. From the formal model to the code, the computing environment, the package, the licensing and the community around software, all these entangled layers constitute our software millefeuille.

To grasp the multifaceted concept of software, we built on three perspectives on how to consider software: as a scientific instrument, as a commodity, and as a social project. These three perspectives allow us to highlight these entanglements, and to highlight how the four epistemic characteristics interplay in different manners in each of these three perspectives, and to which layers they relate in each case.

In particular, we have described how these characteristics sometimes conflict one with another. For example, transparency and consistency conflict within two Weltanschuungen of a scientific instrument. Sustainability and inclusivity also conflict when discussing the required workforce around a software project. These epistemic conflicts account for the tensions that are at play in the opinion pieces in computational chemistry we used as a corpus, and more generally within the issues of computational reproducibility.

Every scientific field possesses its own history as regards its relationship to software. Computational chemistry as a longue durée case study lends itself to study precisely the nuances of these issues and allows to go further a superficial requirement for transparency. We argue that computational reproducibility is complex, its underlying epistemic issues are diverse, and they are deeply entangled with the concept of software, an ever present elephant in the room that still has to be acknowledged in its multiple dimensions as pivotal for computational reproducibility matters.

Acknowledgements

The authors gratefully acknowledge the Science History Institute (Philadelphia) for a fellowship during which part of this research had been achieved. This research project is being supported by a grant from the MSH Lorraine, France.
We are indebted to both anonymous reviewers for their valuable suggestions, which have clearly improved, in our view, this paper.

References

AlNoamany, Yasmin, and John A. Borghi. 2018. “Towards Computational Reproducibility: Researcher Perspectives on the Use and Sharing of Software.” PeerJ Computer Science 4: e163. https://peerj.com/articles/cs-163 (December 31, 2019).

Atmanspacher, Harald, and Sabine Maasen. 2016. Reproducibility: Principles, Problems, Practices, and Prospects. Hoboken, New Jersey: Wiley-Blackwell.

Baker, Monya. 2016. “1,500 Scientists Lift the Lid on Reproducibility.” Nature News 533(7604): 452. http://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970 (December 31, 2019).

Bhandari Neupane, J., Neupane, R. P., Luo, Y., Yoshida, W. Y., Sun, R., & Williams, P. G. (2019). “Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the “Willoughby–Hoye” Scripts for Calculating NMR Chemical Shifts”. *Organic Letters*, *21*(20), 8449–8453. https://doi.org/10.1021/acs.orglett.9b03216

Benureau, Fabien C. Y., and Nicolas P. Rougier. 2018. “Re-Run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions.” Frontiers in Neuroinformatics 11. https://www.frontiersin.org/articles/10.3389/fninf.2017.00069/full (December 31, 2019).

Chue Hong, Neil, Simon Hettrick, Andrew Jones, and Daniel Katz. 2015. “The Price of Open-Source Software – A Joint Response.” Software Sustainability Institute blog. https://www.software.ac.uk/blog/2016-09-22-price-open-source-software-joint-response (December 31, 2019).

Ensmenger, Nathan L. 2010. The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise. The MIT Press.

Geiger, Stuart, Dorothy Roe Howard, Lilly Irani, Nelle Varoquaux, Alexandra Paxton, and Chris Holdgraf. 2019. “Who Pays the Costs of Free and Open-Source Scientific Software?” 4S Annual Meeting. http://tinyurl.com/y2jqs4fb (December 31, 2019).

Gelfert, Axel. 2011. “Scientific Models, Simulation, and the Experimenter’s Regress”. In Models, Simulations, and Representations, edited by Paul Humphreys and Cyrille Imbert, 145-167. Routledge.

Gezelter, J. Daniel. 2015. “Open Source and Open Data Should Be Standard Practices.” The Journal of Physical Chemistry Letters 6(7): 1168–69. https://doi.org/10.1021/acs.jpclett.5b00285 (December 31, 2019).

Hatton, L., & van Genuchten, M. (2019). Computational Reproducibility: The Elephant in the Room. *IEEE Software*, *36*(2), 137–144. https://doi.org/10.1109/MS.2018.2883805

Hey, Tony, and Mike C. Payne. 2015. “Open Science Decoded.” Nature Physics 11(5): 367–69. https://www.nature.com/articles/nphys3313 (December 31, 2019).

Hinsen, Konrad. 2014. “Computational science: shifting the focus from tools to models [version 2; peer review: 2 approved]”. F1000Research 2014, 3:101. https://doi.org/10.12688/f1000research.3978.2

Hinsen, Konrad, and Nicolas Rougier. 2019. “Challenge to Test Reproducibility of Old Computer Code.” Nature 574(7780): 634–634. https://www.nature.com/articles/d41586-019-03296-8 (December 31, 2019).

Horner, Jack, and John Symons. 2014. “Reply to Angius and Primiero on Software Intensive Science”. Philosophy & Technology 27 (3): 491-94. https://doi.org/10.1007/s13347-014-0172-9.

Humphreys, Paul. 2004. Extending Ourselves: Computational Science, Empiricism, and Scientific Method. Oxford University Press.

Jacob, Christoph R. 2016. “How Open Is Commercial Scientific Software?” The Journal of Physical Chemistry Letters 7(2): 351–53. https://doi.org/10.1021/acs.jpclett.5b02609 (December 31, 2019).

Kelty, Christopher M. 2008. Two Bits: The Cultural Significance of Free Software. Duke University Press.

Krylov, Anna I., John M. Herbert, Filipp Furche, Martin Head-Gordon, Peter J. Knowles, Roland Lindh, Frederick R. Manby, Peter Pulay, Chris-Kriton Skylaris, and Hans-Joachim Werner. 2015. “What Is the Price of Open-Source Software?” The Journal of Physical Chemistry Letters 6(14): 2751–54. https://doi.org/10.1021/acs.jpclett.5b01258 (December 31, 2019).

Lejaeghere, K., Bihlmayer, G., Björkman, T., Blaha, P., Blügel, S., Blum, V., Caliste, D., Castelli, I. E., Clark, S. J., Dal Corso, A., de Gironcoli, S., Deutsch, T., Dewhurst, J. K., Di Marco, I., Draxl, C., Dułak, M., Eriksson, O., Flores-Livas, J. A., Garrity, K. F., … Cottenier, S. (2016). “Reproducibility in density functional theory calculations of solids”. *Science (New York, N.Y.)*, *351*(6280), aad3000. https://doi.org/10.1126/science.aad3000

Lenhard, Johannes, and Uwe Küster. 2019. “Reproducibility and the Concept of Numerical Solution.” Minds and Machines 29 (1): 19–36. https://doi.org/10.1007/s11023-019-09492-9.

Leonelli, Sabina. 2019. “Rethinking Reproducibility as a Criterion for Research Quality.” In Including a Symposium on Mary Morgan: Curiosity, Imagination, and Surprise, Research in the History of Economic Thought and Methodology, Emerald Publishing Limited, 129–46. https://doi.org/10.1108/S0743-41542018000036B009 (December 31, 2019).

Mahoney, Michael S. (2008). What Makes the History of Software Hard. IEEE Annals of the History of Computing 30(3): 8–18. https://doi.org/10.1109/MAHC.2008.55

Miletić, Vedran. 2015. “What Is the Price of Open-Source Fear, Uncertainty, and Doubt?” Nudged Elastic Band is my band name. https://nudgedelastic.band/2015/09/what-is-the-price-of-open-source-fear-uncertainty-and-doubt/ (December 31, 2019).

Peng, Roger D. 2011. “Reproducible Research in Computational Science.” Science 334(6060): 1226–27. https://science.sciencemag.org/content/334/6060/1226 (December 31, 2019).

Reinhardt, Carsten. 2001. Chemical Sciences in the 20th Century : Bridging Boundaries. Weinheim; New York: Wiley-VCH.

Reinhardt, Carsten. 2006. “A Lead User of Instruments in Science: John D. Roberts and the Adaptation of Nuclear Magnetic Resonance to Organic Chemistry, 1955–1975.” Isis 97(2): 205–36. https://www.journals.uchicago.edu/doi/abs/10.1086/504732 (November 29, 2019).

Schappals, M., Mecklenfeld, A., Kröger, L., Botan, V., Köster, A., Stephan, S., García, E. J., Rutkai, G., Raabe, G., Klein, P., Leonhard, K., Glass, C. W., Lenhard, J., Vrabec, J., & Hasse, H. (2017). “Round Robin Study: Molecular Simulation of Thermodynamic Properties from Models with Internal Degrees of Freedom”. *Journal of Chemical Theory and Computation*, *13*(9), 4270–4280. https://doi.org/10.1021/acs.jctc.7b00489

Spencer, Matthew. 2015. “Brittleness and Bureaucracy : Software as a Material for Science.” Perspectives on Science 23(4): 466–84. http://dx.doi.org/10.1162/POSC_a_00184 (December 31, 2019).

Stodden, Victoria, Marcia McNutt, David H. Bailey, Ewa Deelman, Yolanda Gil, Brooks Hanson, Michael A. Heroux, John P. A. Ioannidis, and Michela Taufer. 2016. “Enhancing Reproducibility for Computational Methods.” Science 354(6317): 1240–41. https://science.sciencemag.org/content/354/6317/1240 (December 31, 2019).

Symons, John, and Ramón Alvarado. 2019. “Epistemic Entitlements and the Practice of Computer Simulation.” Minds and Machines 29 (1): 37–60. https://doi.org/10.1007/s11023-018-9487-0.

Winsberg, Eric. 2010. *S*cience in the Age of Computer Simulation, Chicago, Ill.: University of Chicago Press.

Winsberg, Eric. 2019. “Computer Simulations in Science”, The Stanford Encyclopedia of Philosophy (Winter 2019 Edition), Edward N. Zalta (ed.). https://plato.stanford.edu/archives/win2019/entries/simulations-science/