

Digital Scientific Notations as a Human-Computer Interface in Computer-Aided Research

Konrad Hinsen^{1,2}

¹Centre de Biophysique Moléculaire, CNRS UPR4301, Orléans, France

²Synchrotron SOLEIL, Division Expériences, Gif sur Yvette, France

Corresponding author:

Konrad Hinsen¹

Email address: konrad.hinsen@cnr.fr

ABSTRACT

Most of today's scientific research relies on computers and software not only for administrative tasks, but also for processing scientific information. Examples of such computer-aided research are the analysis of experimental data or the simulation of phenomena based on theoretical models. With the rapid increase of computational power, scientific software has integrated more and more complex scientific knowledge in a black-box fashion. As a consequence, its users do not know, and do not even have a chance of finding out, which assumptions and approximations their computations are based on. The black-box nature of scientific software has thereby become a major cause of mistakes. The present work starts with an analysis of this situation from the point of view of human-computer interaction in scientific research. It identifies the key role of digital scientific notations at the human-computer interface, reviews the most popular ones in use today, and describes a proof-of-concept implementation of Leibniz, a language explicitly designed as a digital scientific notation for models formulated as mathematical equations.

1 INTRODUCTION

Computers have profoundly changed the way scientific research is done. While the same statement can be made about many other human activities, the impact of computers on scientific research goes beyond their use as mere tools for managing data, writing articles, or communicating with colleagues. Computers process information, and information is the core resource of science. In the natural sciences, practically all results that are obtained from experimental or theoretical work are at some point processed by computers. Algorithms and their implementations in software have become an integral part of the models and methods that scientists apply and study.

The first few decades of computer-aided research have been marked by the development and application of new computational techniques. They have permitted the exploration of ever more complex systems with ever better precision, but also lead to completely new styles of scientific enquiry based on analyzing large amounts of data by statistical methods. However, the initial enthusiasm about the new possibilities offered by computer-aided research has been dampened in recent years as scientists began to realize that the new technology also brings new kinds of problems. Errors in software, or in the way software is applied, are the most obvious one (Merali, 2010; Soergel, 2014). A more subtle problem is the widespread non-reproducibility of computational results, in spite of the fact that computations are fully deterministic (Claerbout and Karrenbach, 1992; Stodden et al., 2016). But perhaps the most insidious effect of the use of computers is that scientists are losing control over their models and methods, which are increasingly absorbed by software and thereby opacified, to the point of disappearing from scientific discourse (Hinsen, 2014). As I will discuss in section 2.3, the consequence is that automated computations are often no longer verifiable, which is an important cause of errors in computer-aided research.

In the philosophy of science, these practical questions and more fundamental ones that practitioners

do not tend to worry about, are discussed in the context of the *epistemic opacity* of automated computation (Imbert, 2017). The overarching issue is that performing a computation by hand, step by step, on concrete data, yields a level of understanding and awareness of potential pitfalls that cannot be achieved by reasoning more abstractly about algorithms. As one moves up the ladder of abstraction from manual computation via writing code from scratch, writing code that relies on libraries, and running code written by others, to having code run by a graduate student, more and more aspects of the computation fade from a researcher's attention. While a certain level of epistemic opacity is inevitable if we want to delegate computations to a machine, there are also many sources of accidental epistemic opacity that can and should be eliminated in order to make scientific results as understandable as possible.

As an example, a major cause for non-reproducibility is the habit of treating an executable computer program, such as the Firefox navigator or the Python interpreter, as an abstraction that is referred to by a name. In reality, what is launched by clicking on the Firefox icon, or by typing "python" on a command line, is a complex assembly of software building blocks, each of which is a snapshot of a continuous line of development. Moreover, a complete computation producing a result shown in a paper typically requires launching many such programs. The complexity of scientific software stacks makes them difficult to document and archive. Moreover, recreating such a software stack identically at a later time is made difficult by the fast pace of change in computing technology and by lack of tool support. A big part of the efforts of the Reproducible Research movement consists of taking a step down on the abstraction ladder. Whereas the individual building blocks of software assemblies, as well as the blueprints for putting them together, were treated as an irrelevant technical detail in the past, this information is now realized as important for reproducibility. In order to make it accessible and exploitable, many support tools for managing software assemblies are currently being developed.

The problem of scientists losing control over their models and methods, leading to the non-verifiability of computations, has a similar root cause as non-reproducibility. Again the fundamental issue is treating a computer program as an abstraction, overlooking the large number of models, methods, and approximations that it implements, and whose suitability for the specific application context of the computation needs to be verified by human experts. To achieve reproducibility, we need to recover control over what software we are running precisely. We must describe our software assemblies in a way that allows our peers to *use* them on their own computers but also to *inspect* how they were built, for example to check if a bug detected in a building block affects a given published result or not. To achieve verifiability, we need to recover control over which models and methods the software applies. We must describe our model and method assemblies in a way that allows our peers to *apply* them using their own software but also to *inspect* them in order to verify that we made a judicious choice. Reproducibility is about the *technical* decomposition of a computation into software building block. Verifiability is about the *scientific* decomposition of a computation into models and methods. As I will show in section 2.4, these two decompositions do not coincide because they are organized according to different criteria.

In this article, I take the point of view that accidental epistemic opacity should be treated as an issue of human-computer interaction in computer-aided research. Since the problem of non-reproducibility is relatively well understood by now, even though effective solutions remain to be developed for many situations, I will focus on non-verifiability as the major unresolved problem, and in particular on the role of digital scientific notations.

2 BACKGROUND

2.1 Motivation

The topics I will cover in this article will probably seem rather abstract and theoretical to most practitioners of computer-aided research. The two personal anecdotes in this section should provide a more down-to-earth motivation for the analysis that follows. Readers who do not need further motivation can skip this section.

In 1997, I wrote an implementation of the popular AMBER force field for biomolecular simulations (Cieplak and Kollman, 1996) as part of a Python library that I published later (Hinsen, 2000). A force field is a function $U(X, \Phi, G)$ expressing the potential energy of a molecular system in terms of the positions of the atoms, X , a set of parameters, Φ , and a labelled graph G that has the atoms as vertices, the covalent bonds as edges, and an "atom type" label on each vertex that describes the chemical environment of the atom. Force fields are the main ingredients to the models used in biomolecular simulation, and the subject of much research activity, leading to frequent updates. The computation of a force field

100 involves non-trivial graph traversal algorithms that are habitually not documented, and in fact hardly even
101 mentioned, in the accompanying journal article, which concentrates on describing how the parameter
102 set Φ was determined and how well the force field reproduces experimental data. I quickly realized
103 that the publication mentioned above plus the publicly available parameter files containing Φ with their
104 brief documentation were not sufficient to re-implement the AMBER force field, so I started gathering
105 complementary information by doing test calculations with other software implementing AMBER, and
106 by asking questions on various mailing lists.

107 One of the features of AMBER that I discovered came as a surprise: its potential energy function
108 depends on the order in which the atoms were listed in the input file that defines the initial configuration
109 of the molecules. This makes no sense at all: interaction energies depend on the nature of the system,
110 describe by the graph G and the parameters Φ , and on its instantaneous configuration X , but not on
111 how the system is represented in a specific file format. I can only speculate about the cause of this
112 design decision, but it is probably the result of simplicity of implementation taking priority over physical
113 reasonableness, and the decision might well have been taken by an inexperienced graduate student. A
114 reviewer of the paper would surely have objected had the feature been described there. However, the
115 feature wasn't documented anywhere else than in the source code of a piece of software, which was
116 never peer reviewed at all.

117 Over the years, I have mentioned this feature to many colleagues, who were all as surprised as I was,
118 and often believed me only after checking for themselves. To the best of my knowledge, no paper and
119 no software documentation mentions this behavior. I can think of only three ways to stumble on it: by
120 experimenting with the atom order in input files, by reading the source code of a simulation program, or
121 by talking to someone who happens to know. In fact, I am not even sure that all software implementing
122 AMBER handles this feature in the same way, given that is in general impossible to obtain identical
123 numbers from different software packages for many other reasons.

124 Pragmatists might ask how important this effect is. I don't think anyone can answer this question in
125 general. The numerical impact on a single energy evaluation is very small. But Molecular Dynamics
126 is chaotic, meaning that small differences can be strongly amplified. There are examples of changes
127 assumed to be without effect on the results of MD simulations turning out to be important in the end (e.g.
128 Reißer et al. (2017)). The hypothesis that AMBER's atom order dependence has no practical importance
129 would have to be checked for all possible applications of the force field. It would clearly be less effort
130 for everybody to simply fix the force field definition.

131 Nearly twenty years later, I was working on the gas-phase structure of small peptides. A nice study
132 combining experiments and simulation, on peptides similar to those I was interested in, caught my inter-
133 est (Jarrold, 2007). I decided to re-do the simulations, using my own software, as a warm-up exercise.
134 But I found very different structures. I must have done something differently... so let's find out what!
135 Unfortunately, I didn't get very far. The details about the simulations that are given in this paper are
136 above average for the field, and the simulation software was published. And yet, the available informa-
137 tion was completely insufficient for understanding what the author really did. Ten years after publication,
138 the precise software version used was no longer available. The description of the force field parameters
139 didn't match the files I could find in a later release. As for the simulation protocol, it didn't help me
140 much to learn that "in some cases ... more sophisticated methods were used", with no more detail given.

141 Readers familiar with the recent efforts to improve computational reproducibility might say that all
142 that's missing is a complete archive of the author's software and input files. But I doubt that would
143 be true. Comparing my calculations with the original author's work by going through hand-optimized
144 source code and machine-specific workflows might be possible in principle, but not in practice. The two
145 main ingredients to these computations are (a) a force field and (b) a heuristic search algorithm for a
146 global minimum. Extracting either one from optimized source code is as hopeless a task as extracting
147 high-level source code from binary executables. Comparing them, and swapping ingredients (e.g. use
148 my force field with his minimization algorithm) are not realistic endeavors at this time.

149 The long-term goal of the research described in this article is to be able to express complex scientific
150 data such as force fields and minimization algorithms in a way that permits their inspection and verifica-
151 tion by human readers. Force fields should be published in a form that gives peer reviewers a chance to
152 detect unphysical features, and users to perform comparisons with other force fields. Verifiability, like
153 reproducibility, should become a requirement for computations in order to ensure the transparency of
154 scientific research. But as a first step, it must become a possibility.

155 2.2 Verification and validation in science

156 The overall goal of science is to acquire reliable knowledge about the world around us. A major obstacle
157 to achieving reliability is the unreliability of the individual scientist. Like all humans, scientists make
158 mistakes, and the pursuit of non-scientific goals such as wealth or social status often interferes with the
159 quest for knowledge. The scientific community has therefore established an error-correction protocol
160 whose main ingredients are peer verification and continuous validation against new observations.

161 Peer verification consists of scientists inspecting their colleagues' work with a critical attitude, watch-
162 ing out for mistakes or unjustified conclusions. In today's practice, the first round of critical inspection
163 is peer review of articles submitted to a journal or conference. Peer review tends to be shallow, as few
164 reviewers re-do experiments or computations. But peer verification does not stop after publication. If
165 a contribution is judged sufficiently important, it will undergo continued critical inspection by other
166 scientists interested in building on its results.

167 Validation against new observations is a much slower process. It comes down to continuously check-
168 ing the coherence of all observations and models in a given domain of science. When contradictions
169 appear, additional experimental or theoretical work is required to figure out what went wrong. The out-
170 come can be a minor correction such as "observation X turned out to be faulty", or a major correction
171 such as "classical mechanics is insufficient to explain the behavior of matter at the atomic scale".

172 Descriptions of the scientific method tend to emphasize the role of validation as the main error
173 correction technique. It is true that validation alone would in principle be sufficient to detect mistakes.
174 However, the error correction process would be extremely inefficient without the much faster verification
175 steps. A missing factor 2 in a calculation can be found more rapidly and more reliably by re-doing the
176 calculation than by constructing an apparatus to validate the result experimentally. Verification gains in
177 importance as science moves on to more complex systems, for which the total set of observations and
178 models is much larger and coherence is much more difficult to achieve. As an illustration, consider a bug
179 in the implementation of a sequence alignment algorithm in genomics research. Sequence alignment
180 is not directly observable, it is merely a first step in processing raw genomics data in order to draw
181 conclusions that might then be amenable to validation. The path from raw data to the prediction of
182 an observable quantity is so long that finding the cause of a disagreement would be impossible if the
183 individual steps could not be verified.

184 Verification is possible only if individual scientific contributions are sufficiently complete that a com-
185 petent reader can follow the overall reasoning and evaluate the reliability of each piece of evidence that
186 is presented. For computer-aided research, this has become a major challenge. A minimal condition
187 for verifying computations is that the software is available for inspection, as K.V. Roberts called for as
188 early as 1969 in the first issue of the journal *Computer Physics Communication* (Roberts, 1969). His
189 advice was not heeded: most scientific software was not published at all, and sometimes even thrown
190 away by its authors at the end of a study. Many widely used software packages were distributed only
191 as executable binaries, with the explicit intention of preventing its users from understanding their inner
192 workings. This development has by now been widely recognized as a mistake and the Reproducible
193 Research movement has been making good progress in establishing best practices to make computations
194 in science inspectable (Stodden et al., 2016).

195 However, the availability of inspectable source code is only the first step in making verification
196 possible. Actually performing this verification is a complicated process in itself, which is often again
197 subdivided into a verification and a validation phase. In the context of software, verification is usually
198 defined as checking that the software conforms to its specification, whereas validation means checking
199 that the specification corresponds to the initial list of requirements. Since the nature of requirements
200 and specifications varies considerably between different domains of application, there is no consensus
201 about the exact borderline between verification and validation. However, as for the scientific method, the
202 general idea is that verification is a faster and more rigorous procedure that focuses on formal aspects,
203 with subsequent validation examining how the software fits into its application context.

204 Today's practice concerning verification and validation of scientific software varies considerably
205 between scientific disciplines. Well-established models and methods, widely used software packages,
206 and direct economic or societal relevance of results are factors that favor the use of verification and
207 validation. Independently of these domain-specific factors, the place of a piece of software in the full
208 software stack required for a computation determines which verification and validation techniques are
209 available and appropriate. The typical four-layer structure of this software stack is shown in Fig. 1. On

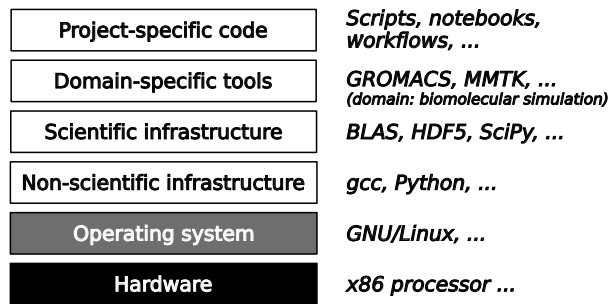


Figure 1. A typical software stack in scientific computing consists of four layers on top of hardware and systems software. The lower two layers contain widely used infrastructure software that can be verified using generic techniques from software engineering. The upper two layers are specified by scientific discourse and must be verified in its context.

210 a foundation consisting of hardware and systems software, the first layer consists of infrastructure that
 211 is not specific to scientific research, such as compilers. From the scientist's point of view, these are
 212 commodities provided by the outside world. The next layer consists of scientific infrastructure software
 213 that provides widely used algorithms, e.g. for numerical analysis or data management. This generally
 214 stable software is developed for research and in contact with scientists, but in terms of verification and
 215 validation can be handled like non-scientific infrastructure, because the scientific knowledge embedded
 216 into this software consists only of well-known models and methods and the software has a clear though
 217 typically informal specification.

218 The upper two layers are the most difficult ones to verify because there their specifications are in-
 219 complete or non-existent. The top layer consists of code written for a specific research project, with the
 220 goal of computing something that has never been computed before. It is also unlikely to be reused with-
 221 out modification. This makes it impossible to apply standard testing procedures. Moreover, quite often
 222 this top layer consists of scripts that launch executables written independently and in distinct program-
 223 ming languages, making it difficult to exploit language-centric verification approaches such as static type
 224 checking.

225 One level below, there is a more stable layer of domain-specific tools, which are developed by and
 226 for communities of scientists whose sizes range from one to a hundred research groups. In fundamental
 227 research, where models and methods evolve rapidly, this domain-specific software is almost as difficult
 228 to verify as the project-specific layer. Moreover, it is typically developed by scientists with little or no
 229 training in software engineering. For many years, verification and validation of domain-specific tools was
 230 very uncommon in fundamental research. Today, widely used community-supported software packages
 231 use quality assurance techniques such as unit testing and sometimes continuous integration. However,
 232 the scientific validity of the software is still not systematically evaluated, and it is not even clear how
 233 that could be achieved. Journals dedicated to the publication of scientific software such as the [Journal](#)
 234 [of Open Source Software](#) (Smith et al., 2017) or the [Journal of Open Research Software](#) do not even ask
 235 reviewers to comment on scientific correctness because such a request would be unreasonable given the
 236 current state of the art.

237 The basic difficulty with verifying and validating the top layers of scientific software is the lack of
 238 clear specifications. The core of such a specification would be made up of the models and methods
 239 that are applied. They are, however, exactly what researchers modify in the course of their work. As
 240 a consequence, each computation requires its own *ad-hoc* specification that combines some established
 241 models and methods with some more experimental ones into a whole that is usually too complex to
 242 be written down in a useful way. The closest approximation to an informal specification is the journal
 243 article that describes the scientific work. An essential part of verification is therefore to check if the
 244 computation correctly implements the informal description given in the article, or inversely if the journal
 245 article correctly describes what the software does. To understand the challenges of this step, it is useful
 246 to take a closer look at the interface between scientific discourse and scientific software.

2.3 Informal and formal reasoning in scientific discourse

The main purpose of scientific discourse, whose principal elements today are journal articles and conference presentations, is to communicate new findings in a way that permits peer verification and re-use of the findings in later research. Another category of scientific discourse serves pedagogical purposes: review articles and monographs summarize the state of the art and textbooks teach established scientific knowledge to future generations of scientists. A common aspect of all these writings aimed at experts or future experts is an alternation of informal and formal reasoning. More precisely, formal deductions are *embedded* in an informal narrative.

Before the advent of computers, formal deductions were mainly mathematical derivations and occasional applications of formal logic. In this context, the formal manipulations are performed by the same people who write the informal narratives, with the consequence that the back and forth transitions between the two modes of reasoning, informal and formal, is often blurred. This explains why mathematical notation is often much less formal and precise than its users believe it to be (Boute, 2005; Sussman and Wisdom, 2002). An illustration is provided in Fig. 2, which shows a simple line of reasoning from elementary physics. Only a careful study of the text reveals that the parts typeset in blue correspond to formal reasoning. One way to identify these parts is to try to replace the textual description as much as possible by output from a computer algebra system. The parts typeset in black introduce the context (Newtonian physics) and define the formal symbols used in the equations in terms of physical concepts.

Motion of a mass on a spring

We consider a point-like object of mass m attached to a spring of force constant k whose mass we assume to be negligible. The other end of the spring is attached to a wall. When the particle is at position x , the force acting on it is given by

$$F = -k \cdot d, \tag{1}$$

where $d = x - l$ is the displacement of x relative to the spring's equilibrium length l . Newton's equation of motion for the mass takes the form

$$F = m\ddot{x} = -k \cdot (x - l). \tag{2}$$

This second-order ordinary differential equation, which can be rewritten as

$$\ddot{d} = -\frac{k}{m}d \tag{3}$$

in terms of the displacement $d = x - l$, has the solution

$$d(t) = A \cos(\omega t + \delta), \tag{4}$$

where $\omega = k/m$ is the angular frequency of the oscillatory motion, and the amplitude A and phase δ are arbitrary real numbers.

Figure 2. Mixing informal and formal reasoning in scientific discourse. The blue parts describe formal reasoning. The black parts establish the context and define the interpretation of the formal equations.

Computers have vastly broadened the possibilities of formal reasoning through automation. Moreover, the fact that computation enforces a clear distinction of formal and informal reasoning makes it a useful intellectual tool in itself (Knuth, 1974; Sussman and Wisdom, 2002). However, computing has led to a complete separation of automated formal reasoning from the informal narratives of scientific discourse. Even in the ideal case of a publication applying today's best practices for reproducible research, the reader has to figure out how text and mathematical formulas in the paper relate to the contents of the various files making up the executable computation.

This separation creates an important obstacle to verification. In the human-only scenario, both informal and formal reasoning are verified by a single person who, like the author, would not particularly care about the distinction. In the computer-assisted scenario, the narrative on its own cannot be verified because it is incomplete: the formal parts of the reasoning are missing. The computation on its own can be partially checked using software engineering techniques such as testing or static type checking, but in the absence of a specification, verification must remain incomplete. No amount of testing and verifying

278 on the software side can verify that the computation actually does what it is expected to do. In terms of
279 the illustration of Fig. 2, no verification restricted to the blue parts can establish that Eq. (2) is the cor-
280 rect equation to solve, and no human examination of the black parts can establish that the computation
281 correctly solves Eq. (2).

282 To the best of my knowledge, the causes of mistakes in computer-aided research have never been
283 analyzed systematically in a scientific study. However, my personal experience from 30 years of research
284 in computational physics and chemistry suggests that roughly half of the mistakes that persist in spite
285 of careful checks at all levels, in other words the mistakes that are detected after rather than before
286 the publication of the results, can be described as “the computation was perfectly reasonable but did
287 not correspond to the scientific problem as described in the paper”. One typical manifestation of this
288 problem is a mistake in a numerical constant in software source code, which includes the frequent case
289 of a wrong sign. This was the cause for a widely publicized series of retractions of published protein
290 structures, following the discovery of a sign error in data processing software (Miller, 2006). Another
291 variant is papers describing the computation only incompletely and in such a way that readers assume the
292 computation to be different from what it actually was. A well-known example is the paper by Harvard
293 economists Reinhart and Rogoff that supported austerity politics (Reinhart and Rogoff, 2010). It was
294 based on an initially unpublished Excel spreadsheet whose later inspection by independent researchers
295 revealed assumptions not mentioned in the paper, and mistakes in the implementation of some formulas
296 (Herndon et al., 2014). A third variant, perhaps even the most frequent one, is scientists using software
297 without knowing what it does exactly, leaving them unable to judge the well-foundedness of the methods
298 that they are applying. A high-impact example concerning the analysis of fMRI brain scans was recently
299 described (Eklund et al., 2016).

300 It is useful to look at this problem as a case of Human-Computer Interaction (HCI), the human part
301 being the informal scientific discourse and the computer part being the computation. This point of view
302 makes it clear that no improvement can be expected from focusing exclusively on scientific discourse or
303 on software. In order to prevent the kinds of mistakes that I have described, scientists must have precise
304 control over what software does when writing it, and a precise understanding of what software does
305 when they take the user’s or the verifier’s role.

306 The popularity of computational notebooks, introduced in 1988 with the computer algebra system
307 Mathematica (Wolfram Research Inc, 1988) and later implemented for a wide range of programming
308 languages by the Jupyter project (Kluyver et al., 2016), shows that the re-unification of informal and
309 formal reasoning into a single document corresponds to a real need in the scientific community. The
310 computational notebook is a variant of the earlier idea of literal programming (Knuth, 1984), which
311 differs in that the formal element embedded into the narrative is not a piece of software but a computation
312 with specific inputs and outputs. Its popularity is even more remarkable in view of the restrictions
313 that today’s implementations impose: a notebook contains a single linear computation, allowing neither
314 re-usable elements nor an adaptation of the order of presentation to the structure of the surrounding
315 narrative. Both restrictions are a consequence of the underlying computational semantics: the code cells
316 in a notebook are sent, one by one, as input to a command-line interpreter. Non-linear control flow can
317 only happen inside a cell, and there is no way to refer to another cell, or sequence of cells, in order to
318 re-use its contents. Notebooks can therefore capture only the top layer of the software stack shown in
319 Fig. 1, and even that only for relatively simple cases.

320 Another example for the re-unification of informal and automated formal reasoning is given by text-
321 books that include short computer programs to explain scientific concepts. Most of them deal specifically
322 with computational science and use the code as examples, e.g. Langtangen (2012), but some aim at con-
323 veying more fundamental scientific concepts using executable code for preciseness of notation (Sussman
324 et al., 2013; Sussman and Wisdom, 2014). Unfortunately, the code in such textbooks is in general not
325 directly executable because they were prepared using traditional editing technology in view of being
326 printed on paper. On the other hand, today’s computational notebooks are not flexible enough to handle
327 such more complex computational documents, which illustrates that the combination of narratives with
328 computational content is still in its infancy.

329 **2.4 Human-computer interaction in computer-aided research**

330 Computer programs are predominantly viewed as tools, not unlike physical devices such as cars or mi-
331 crosopes. Humans interacting with tools can take on different roles. For software, the main roles are

332 “developer” and “user”, whereas for physical devices the number of roles tends to be larger: “designer”,
 333 “producer”, “maintenance provider”, and “user”. Software developers interact with software at the source
 334 code level using software development tools. Software users interact with the same software via a user
 335 interface designed by its developers. Some software provides several levels of user interfaces, for exam-
 336 ple by providing a scripting or extension programming language for an intermediate category of “power
 337 users”. Each role in interacting with software is associated with a different mental model of how the
 338 software works. The basic user’s mental model is restricted to *what* the software does. A power user
 339 knows *how* the software accomplishes its tasks, i.e. what the basic algorithms and data structures are.
 340 Developers also need to be aware of the software’s architecture and of many implementation details.

341 In the development of scientific software, these different roles and the associated mental models
 342 have so far hardly been taken into account. In fact, in many scientific disciplines there are no clearly
 343 defined roles yet that people could adopt. More generally, human-computer interaction in computer-
 344 aided research has been shaped by historical accidents rather than by design. In particular, most software
 345 user interfaces are the result of a policy giving highest priority to rapid development and thus ease of
 346 implementation.

347 The analysis of verification that I have given in sections 2.2 and 2.3 suggests that the users’ minimal
 348 mental model of scientific software must include everything that may affect the results of a computation.
 349 This is a condition for scientists being able to verify the interface between informal and formal reasoning,
 350 i.e. to judge if a computation provides an answer to the scientific question being asked. Technical
 351 details can then be safely left to specialists. An important kind of specialist knowledge in many fields
 352 of scientific computing concerns performance characteristics such as the use of CPU time and memory.
 353 Performance experts would choose the best hardware and software for a given task, and work with
 354 developers on fine-tuning software. The extreme agility requirements in scientific research may well
 355 require many people to adopt multiple roles, such as user and performance expert, or performance expert
 356 and developer. However, verifying and working with the results should never require more than user-
 357 level knowledge.

358 In the rest of this article, I will concentrate on human-computer interaction at the user level, focusing
 359 on the interplay between informal and computer-assisted formal reasoning in scientific discourse. The
 360 overall goal is to explore how computations can be defined in such a way that human scientists can most
 361 easily understand and verify them.

362 **Case study: simulating the predator-prey equations**

363 A simple example will help to illustrate the interface between informal and formal reasoning and between
 364 humans and computers. The predator-prey equations, also known as the Lotka-Volterra equations, de-
 365 scribe the dynamics of the populations of two interacting species in an ecosystem in terms of non-linear
 366 differential equations. They take the form

$$367 \frac{dx}{dt} = \alpha x - \beta xy \quad (5)$$

$$368 \frac{dy}{dt} = -\gamma y + \delta xy, \quad (6)$$

369 where x is the number of prey, y the number of predators, and the positive constants α , β , γ , and δ
 370 describe the events that change these numbers: α is the birth rate of prey, β the rate at which prey are
 371 eaten by predators, γ the death rate of predators, and δ the food-dependent birth rate of predators. These
 equations are based on a number of assumptions that a textbook or journal article –informal reasoning –
 would discuss for each potential application.

For given parameters α , β , γ , δ , and given initial values $x(t_0), y(t_0)$, the predator-prey equations fully
 define $x(t)$ and $y(t)$ for all $t > t_0$, i.e. they are a complete specification for their solution. However, the
 equations do not provide an algorithm for actually finding a solution. To this day, no closed-form solution
 is known. Numerical solutions can be obtained after an approximation step that consists of discretization.
 For simplicity of presentation, I will use the simple Euler discretization scheme, even though it is *not* a
 good choice in practice. This scheme approximates a first-order differential equation of the form

$$372 \frac{dz}{dt} = f(t, z(t)) \quad (7)$$

by the discretized equation

$$z(t+h) = z(t) + hf(t, z(t)), \quad (8)$$

372 which can be iterated, starting from $t = t_0$, to obtain $z(t)$ for a discrete set of time values $t + nh$ for any
 373 natural number n . For the predator-prey equations, $z(t)$ is replaced by the two components $x(t), y(t)$,
 374 which does not entail any fundamental change to the Euler method.

375 The discretized equation can be solved exactly using rational arithmetic. However, for performance
 376 reasons, rational arithmetic is usually approximated by inexact floating-point arithmetic. This approxi-
 377 mation involves two steps:

- 378 1. The choice of a floating-point representation with associated rules of arithmetic. The most popular
 379 choices are the single- and double-precision binary representations of IEEE standard 754-2008.
- 380 2. The choice of the order of the floating-point arithmetic operations, which due to rounding errors
 381 do not respect the usual associativity rules for exact arithmetic.

382 The scientific decomposition of this computation thus consists of five parts, each of which requires a
 383 justification or discussion in an informal narrative:

- 384 (A) The description of the scientific question by the predator-prey equations.
- 385 (B) The values of the constant parameters.
- 386 (C) The values of the initial values $x(t_0)$ and $y(t_0)$.
- 387 (D) The discretization using the Euler method and the choice of h .
- 388 (E) The choices concerning the floating-point approximation: precision, rounding mode, order of arith-
 389 matic operations.

390 The technical decomposition of a typical implementation of this computation looks very different:

- 391 1. A program that implements an algorithm derived from the predator-prey equations, using partially
 392 specified floating-point arithmetic. This program also reads numerical parameters from a file, and
 393 calls a function from an ODE solver library.
- 394 2. An ODE solver library implementing the Euler method in partially specified floating-point arith-
 395 metic.
- 396 3. An input file for the program that provides the numerical values of the parameters, the initial values
 397 $x(t_0)$ and $y(t_0)$, and the step size h .
- 398 4. A compiler defining the precise choices for floating-point arithmetic.

399 The transitions from A to 1, from B/C/D to 3, and from D/E to 1/2/4 require human verification
 400 because they represent transitions from informal to formal reasoning. The two approximations that
 401 require scientific validation are A→D (the Euler method) and D→E (floating-point approximation). In
 402 this validation, formal reasoning (running the code) is an important tool. The first validation is in practice
 403 done empirically, by varying the step size h and checking for convergence. The many subtleties of this
 404 procedure are the subject of numerical analysis. The validity of the floating-point approximation would
 405 be straightforward to check if the computation could be done in exact rational arithmetic for comparison.
 406 This is unfortunately not possible using the languages and libraries commonly used for numerical work,
 407 which either provide no exact rational arithmetic at all or require the implementation 1/2 to be modified,
 408 introducing another opportunity for mistakes.

2.5 Digital Scientific Notations

A *digital scientific notation* is a formal language that is part of the user interface of scientific software. This definition includes, but is not limited to, formal languages embedded into informal scientific discourse, as in the case of computational notebooks. Another important category contains the file formats used to store scientific datasets of all kinds. As I have explained in section 2.4, the scientific information to be expressed in terms of digital scientific notations includes everything that is relevant at the user level, i.e. everything that has an impact on the results of a computation. This includes in particular scientific models and computational methods, but also more traditional datasets containing, for example, experimental observations, simulation results, or fitted parameters.

Digital scientific notations differ in two major ways from traditional mathematical notation:

1. They must be able to express algorithms, which take an ever more important role in scientific models and methods.
2. They must be adapted to the much larger size and much more complex structure of scientific models and data that can be processed with the help of computers.

The first criterion has led to the adoption of general-purpose programming languages as digital scientific notations. Most published computational notebooks, for example, use either Python or R for the computational parts. The main advantage of using popular programming languages in computational documents is good support in terms of tools and libraries. On the other hand, since these are programming rather than specification languages, they force users to worry about many technical details that are irrelevant to scientific knowledge. For example, the Python language has three distinct data types for sequences: lists, tuples, and arrays. There are good technical reasons for having these separate types, but for scientific communication, the distinction between them and the conversions that become necessary are only a burden. Another disadvantage is that the source code of a programming language reduces all scientific knowledge to algorithms for computing specific results. This process implies a loss of information. For example, the predator-prey equations from Eq. 5 contain information that is lost in a floating-point implementation of its discrete approximation because the latter can no longer be used to deduce general properties of exact solutions.

Domain-specific languages (DSLs) are another increasingly popular choice for representing scientific knowledge. In contrast to general-purpose programming languages, DSLs are specifically designed as digital scientific notations, and usually avoid the two main disadvantages of programming languages mentioned above. Most scientific DSLs are embedded in a general-purpose programming language. A few almost arbitrarily selected examples are Liszt, a DSL for mesh definitions in finite-element computation, embedded in Scala (DeVito et al., 2011), Kendrick, a DSL for ODE-based models in epidemiology, embedded in Smalltalk (Bui et al., 2016), and an unnamed DSL for micromagnetics, embedded in Python (Beg et al., 2017). The choice for an embedded DSL is typically motivated by simpler implementation and integration into an existing ecosystem of development tools and libraries. On the other hand, embedded DSLs are almost impossible to re-implement in a different programming language with reasonable effort, which creates a barrier to re-using the scientific knowledge encoded using them. A stand-alone DSL is independent of any programming language, as is illustrated by Modelica (Fritzson and Engelson, 1998), a general modeling language for the natural and engineering sciences for which multiple implementations in different languages exist. However, each of these implementations is a rather complex piece of software.

Looking at how scientific DSLs are used in practice, it turns out that both embedded and stand-alone DSLs end up being a user interface for a single software package, or at best a very small number of packages. Adopting an existing DSL for a new piece of software is very difficult. One obstacle is that the existing DSLs can be too restrictive, having been designed for a narrowly defined domain. For embedded DSLs, interfacing the embedding language with the implementation language of the new software can turn out to be a major obstacle. Finally, the complexity of a DSL can be prohibitive, as in the case of Modelica. In all these scenarios, the net result is a balkanization of digital scientific knowledge because for each new piece of software, designing a new DSL is often the choice of least effort.

These considerations lead to two important criteria for good digital scientific notations that existing ones do not satisfy at the same time:

- **Generality.** While it is unrealistic to expect that a single formal language could be adequate for

462 computer-aided research in all scientific disciplines, it should be usable across narrowly defined
463 domains of research, and be extendable to treat newly discovered scenarios.

- 464 • Simplicity. The implementation of user interfaces based on a digital scientific notation should not
465 require a disproportionate effort compared to the implementation of the scientific functionality of
466 a piece of software.

467 In the next section, I will describe an experimental digital scientific notation that was designed with
468 these criteria in mind, and report on first experiences with simple toy applications. While it is too early
469 to judge if this particular notation will turn out to be suitable for real-life applications, it illustrates that
470 better digital scientific notations can be designed if their role at the human-computer interface is fully
471 taken into account.

472 3 LEIBNIZ, A DIGITAL SCIENTIFIC NOTATION FOR CONTINUOUS MATH- 473 EMATICS

474 An important foundation of many scientific theories is the mathematics of smoothly varying objects such
475 as the real numbers. This foundation includes in particular geometry, analysis, and linear algebra. In
476 some scientific disciplines, such as physics and chemistry, this is the dominant mathematical foundation.
477 In other disciplines, such as biology, it is one important foundation among others, notably discrete math-
478 ematics. The digital scientific notation *Leibniz*, named after 17th-century polymath Gottfried Wilhelm
479 Leibniz, focuses on continuous mathematics and its application. Like many computer algebra systems,
480 but unlike common programming languages, it can express functions of real numbers and equations
481 involving such functions, in addition to the discrete approximations using rational or floating-point num-
482 bers that are used in numerical work.

483 The design priorities for Leibniz are:

- 484 • Embedding in narratives such as journal articles, textbooks, or software documentation, in order
485 to act as an effective human-computer interface. The code structure is subordinate to the structure
486 of the narrative.
- 487 • Generality and simplicity, as discussed in section 2.5.

488 Before discussing how Leibniz achieves these goals, I will present the language through a few illus-
489 trative examples.

490 Figs. 3, 4, and 5 show three views of a Leibniz document introducing the predator-prey equations
491 from section 2.4. This and other examples are also available on-line in the [Leibniz example collection](#).
492 Fig. 3 shows the author view. Leibniz is implemented as an extension to the document language Scribble
493 (Flatt et al., 2009), which is part of the Racket ecosystem (Felleisen et al., 2015). Scribble source
494 code is a mixture of plain text and commands, much like the better known document language \LaTeX
495 (Lamport, 1994). Commands start with an @ character. Leibniz adds several commands such as @op and
496 @equation, which define elements of Leibniz code. The Leibniz processing tool generates the two
497 other views from the author's input document. Fig. 4 shows the reader view, a rendered HTML page, in
498 which the Leibniz code is typeset on a blue background. This makes the transition between informal and
499 formal reasoning visible at a glance. The machine-readable view, shown in Fig. 5, is an XML file that
500 represents the code in a very rigid format to facilitate processing by scientific software.

501 In terms of semantics, Leibniz' main source of inspiration has been the OBJ family of algebraic spec-
502 ification languages (Goguen et al., 2000), and in particular its most recent incarnation, Maude (Clavel
503 et al., 2002). In fact, the semantics of the current version of Leibniz are a subset of Maude's functional
504 modules, the main missing features being conditional sort membership and the possibility to declare
505 operators as commutative and/or associative. As I will discuss later, this minimalist language is not suffi-
506 cient for dealing with the complex scientific models used in real research. Various features will be added
507 in the future as the need becomes evident in practical applications.

508 As the reference to the OBJ family suggests, Leibniz is based on term rewriting. The code units in
509 Leibniz are called *contexts* (corresponding essentially to Maude's functional modules) and consist of (1)
510 the definition of an order-sorted term algebra, (2) a list of rewrite rules, and (3) any number of assets,
511 which are arbitrary values (terms or equations) identified by unique labels. A Leibniz document, such as

```

#lang leibniz

@title{The predator-prey equations}
@author{Konrad Hinsen}

@import["functions" "functions.xml"]

@context["predator-prey" #:use "functions/derivatives- $\mathbb{R}$ - $\mathbb{R}$ "]{

The predator-prey equations, also known as the Lotka-Volterra
equations, describe the dynamics of two interacting species in an
ecosystem in terms of non-linear differential equations.

The two interacting time-dependent observables are the number of prey,
 $\text{@op}\{\text{prey} : \mathbb{R} \rightarrow \mathbb{R}\}$ , and the number of predators,  $\text{@op}\{\text{predators} : \mathbb{R} \rightarrow \mathbb{R}\}$ .
Although the number of individuals of a species is really an integer,
it is taken to be a real number for the benefit of using differential
equations. The two coupled equations for  $\text{@term}\{\text{prey}\}$  and
 $\text{@term}\{\text{predators}\}$  are
@inset{
  @equation[pp1]{ $\frac{d}{dt}(\text{prey}) =$ 
    (prey-growth-rate  $\times$  prey)
    - (predation-rate  $\times$  predators  $\times$  prey)}
  @equation[pp2]{ $\frac{d}{dt}(\text{predators}) =$ 
    (predator-growth-rate  $\times$  predators  $\times$  prey)
    - (predator-loss-rate  $\times$  predators)}}

These equations are based on a few assumptions:
@itemlist[
  @item{In the absence of predators, the prey exhibits exponential
    growth described by  $\text{@op}\{\text{prey-growth-rate} : \mathbb{R}\}$ .}
  @item{The number of prey decreases by predation, which is
     $\text{@op}\{\text{predation-rate} : \mathbb{R}\}$  times the number of encounters
    between individuals of each species. The latter is taken to
    be proportional to both  $\text{@term}\{\text{prey}\}$  and  $\text{@term}\{\text{predators}\}$ .}
  @item{In the absence of prey, the number of predators decreases by
    starvation, described by  $\text{@op}\{\text{predator-loss-rate} : \mathbb{R}\}$ .}
  @item{The number of predators grows with the availability of food,
    which, like predation, is proportional to both  $\text{@term}\{\text{prey}\}$  and
     $\text{@term}\{\text{predators}\}$  with the proportionality constant
     $\text{@op}\{\text{predator-growth-rate} : \mathbb{R}\}$ .}
}

```

Figure 3. The author view of a Leibniz document shows Leibniz code embedded in a narrative. Most of the commands (starting with @) are inherited from the Scribble document language, only @context, @op, @term, and @equation are added by Leibniz. This example defines a single context called predator-prey that uses another context called derivatives- $\mathbb{R} \rightarrow \mathbb{R}$ that is defined in an imported Leibniz document functions.

512 the one shown in Figs. 3, 4, and 5, is a sequence of such contexts, each of which is identified by a unique
513 name. A context can *use* another context, inheriting its term algebra and its rewrite rules, or *extend* it, in
514 which case it also inherits its variables and assets. In a typical Leibniz document, each context extends
515 the preceding one, adding or specializing scientific concepts. This corresponds to a frequent pattern of
516 informal reasoning in scientific discourse that starts with general concepts and assumptions and then
517 moves on to more specific ones. The “use” relation typically serves for references to contexts imported
518 from other documents that treat a more fundamental theory or methodology. In the example, the context
519 predator-prey uses a context from another document (shown partially in Fig. 6 and [available online](#))
520 called functions that defines real functions of a single real variable having derivatives. When using
521 or extending contexts, it is possible to specify transformations, in particular renaming to avoid name
522 clashes. A small number of builtin contexts defines booleans and a hierarchy of number types with
523 associated arithmetic operations.

524 Like in the OBJ family, a term algebra is defined by operator declarations, which in turn refer to
525 sorts defined by sort declarations. All these declarations can be inserted in arbitrary order into the nar-
526 rative contained in the body of a @context declaration, and can also be repeated. The predator-prey
527 example contains no sort or subsort declarations of its own, but it adds six nullary operators, represent-

The predator-prey equations

by Konrad Hinsén

Context *predator-prey*
uses
functions/derivatives-
 $\mathbb{R} \rightarrow \mathbb{R}$

The predator-prey equations, also known as the Lotka-Volterra equations, describe the dynamics of two interacting species in an ecosystem in terms of non-linear differential equations.

The two interacting time-dependent observables are the number of prey, $\text{prey} : \mathbb{R} \rightarrow \mathbb{R}$, and the number of predators, $\text{predators} : \mathbb{R} \rightarrow \mathbb{R}$. Although the number of individuals of a species is really an integer, it is taken to be a real number for the benefit of using differential equations. The two coupled equations for prey and predators are

$$\begin{aligned} \text{pp1: } \mathcal{D}(\text{prey}) &= (\text{prey-growth-rate} \times \text{prey}) - (\text{predation-rate} \times \text{predators} \times \text{prey}) \\ \text{pp2: } \mathcal{D}(\text{predators}) &= (\text{predator-growth-rate} \times \text{predators} \times \text{prey}) - (\text{predator-loss-rate} \times \text{predators}) \end{aligned}$$

These equations are based on a few assumptions:

- In the absence of predators, the prey exhibits exponential growth described by $\text{prey-growth-rate} : \mathbb{R}_p$.
- The number of prey decreases by predation, which is $\text{predation-rate} : \mathbb{R}_p$ times the number of encounters between individuals of each species. The latter is taken to be proportional to both prey and predators.
- In the absence of prey, the number of predators decreases by starvation, described by $\text{predator-loss-rate} : \mathbb{R}_p$.
- The number of predators grows with the availability of food, which, like predation, is proportional to both prey and predators with the proportionality constant $\text{predator-growth-rate} : \mathbb{R}_p$.

Figure 4. The reader view of a Leibniz document. All code is shown on a blue background.

528 ing the predator and prey populations and the four rate constants, to the term algebra it inherits from
529 *functions/derivatives- $\mathbb{R} \rightarrow \mathbb{R}$* . The sort \mathbb{R} stands for the real numbers, the sort \mathbb{R}_p for the positive
530 real number, and the sort $\mathbb{R} \rightarrow \mathbb{R}$ for real functions of one real variable. Other than these nullary
531 operators, the context *predator-prey* only defines two assets, the two differential equations for the
532 predator and prey populations, identified by the labels *pp1* and *pp2*. An example for binary infix op-
533 erators with rewrite rules can be found in the context *functions/derivatives- $\mathbb{R} \rightarrow \mathbb{R}$* partially
534 shown in Fig. 6. It defines rules for the sum, difference, and product of real functions. Contrary to sort
535 and operator declarations, the order of rewrite rules is important because when multiple rules are eligible
536 for rewriting a term, the textually first one is selected.

537 The context *functions/derivatives- $\mathbb{R} \rightarrow \mathbb{R}$* also illustrates one of the few special operators in
538 Leibniz, the bracket operator, which is merely syntactic sugar, the term $a[b]$ being treated like $[\](a,$
539 $b)$, except that $[\]$ is not legal operator syntax. The two other special operators are superscript and
540 subscript: a_b is equivalent to $_{(a, b)}$ and a^b becomes $\hat{(a, b)}$. In addition to these three special
541 operators, contexts can define arbitrary prefix operators of the form $\text{op}(a, b)$, with any number of
542 arguments, and arbitrary binary infix operators of the form $a \text{ op } b$. There are no precedence rules
543 for infix operators in Leibniz, the use of round brackets is obligatory to resolve ambiguities. The sole
544 exception is a chain of identical binary operators at the same level of the expression. For example, $a +$
545 $b + c$ is allowed and equivalent to $(a + b) + c$. This rule has been inspired by the Pyret language
546 (Pro, 2018) and is a compromise between the familiarity of the precedence rules in mathematics and the
547 ease of not having to remember precedence values for a potentially large number of infix operators. The
548 current Leibniz examples also make extensive use of mathematical Unicode symbols in order to define
549 familiar-looking operators. This is, however, a question of style rather than a language feature.

550 Another [online example](#) illustrates how Leibniz can help to document and validate approximations.
551 The example implements Heron's algorithm for computing square roots, which is a special case of the
552 Newton-Raphson method for finding the roots of a function. The algorithm is first developed for real
553 numbers, with test computations using exact rational number arithmetic. A conversion tool that is part of
554 the Leibniz implementation then derives a floating-point version of the algorithm using the 64-bit binary
555 representation of IEEE standard 754-2008, keeping the order of arithmetic operations from the original

```

1 <leibniz-document>
2 <context id="predator-prey">
3 <includes>
4 <use>functions/derivatives-R-R</use>
5 </includes>
6 <sorts>
7 <sort id="Rq" />
8 <sort id="R-r" />
9 </sorts>
10 <subsorts />
11 <vars />
12 <ops>
13 <op id="predator-loss-rate">
14 <arity />
15 <sort id="Rq" />
16 </op>
17 <op id="predators">
18 <arity />
19 <sort id="R-r" />
20 </op>
21 <op id="predation-rate">
22 <arity />
23 <sort id="Rq" />
24 </op>
25 <op id="prey-growth-rate">
26 <arity />
27 <sort id="Rq" />
28 </op>
29 <op id="prey">
30 <arity />
31 <sort id="R-r" />
32 </op>
33 <op id="predator-growth-rate">
34 <arity />
35 <sort id="Rq" />
36 </op>
37 </ops>
38 </rules />
39 <assets>
40 <asset id="pp2">
41 <equation>
42 <vars />
43 <left>
44 <term op="q">
45 <term-or-var name="predators" />
46 </term>
47 </left>
48 <condition />
49 <right>
50 <term op="_">
51 <term op="x">
52 <term op="x">
53 <term-or-var name="predator-growth-rate" />
54 <term-or-var name="predators" />
55 </term>
56 <term-or-var name="prey" />
57 </term>
58 <term op="_">
59 <term-or-var name="predator-loss-rate" />
60 <term-or-var name="predators" />
61 </term>
62 </right>
63 </equation>
64 </asset>
65 </assets>
66 <asset id="pp1">
67 <equation>
68 <vars />
69 <left>
70 <term op="q">
71 <term-or-var name="prey" />
72 </term>
73 </left>
74 <condition />
75 <right>
76 <term op="_">
77 <term op="x">
78 <term-or-var name="prey-growth-rate" />
79 <term-or-var name="prey" />
80 </term>
81 <term op="x">
82 <term op="x">
83 <term-or-var name="predation-rate" />
84 <term-or-var name="predators" />
85 </term>
86 <term-or-var name="prey" />
87 </term>
88 </right>
89 </equation>
90 </asset>
91 </assets>
92 </context>
93 </leibniz-document>
94

```

Figure 5. The machine-readable view of the predator-prey example.

556 algorithm, which is unambiguous in Leibniz. This example also showcases another user interface feature:
 557 in the reader view, computationally derived information is typeset on a green background, making it easy
 558 to distinguish from the human input typeset on a blue background.

559 A final feature of Leibniz that deserves discussion is its type system, or rather sort system as it is
 560 habitually called in the context of formal logic and term algebras. In this system, directly taken over
 561 from Maude, sort and subsort declarations define a directed acyclic sort graph, which in general consists
 562 of multiple connected components called kinds. Operator declarations assign a sort to each term and a
 563 required sort to each argument position of a non-nullary operator. Mismatches at the kind level, i.e. an
 564 argument sort not being in the same connected component as a required sort, lead to a rejection of a term
 565 in what resembles static type checking in programming languages. Mismatches inside a kind, however,
 566 are tolerated. The resulting term is flagged as potentially erroneous but can be processed normally by
 567 rewriting. If in the course of rewriting the argument gets replaced by a value that is a subsort of the
 568 required type, the error flag is removed again. The presence of an error flag on a result of a computation
 569 thus resembles a runtime error in a dynamically typed language. This mixed static-dynamic verification
 570 system offers many of the benefits of a static type checker, but also allows the formulation of constraints
 571 on values that cannot be verified statically. Leibniz uses this feature to define fine-grained subsorts on
 572 the number sorts, e.g. “positive real number” or “non-zero rational number”, which turn out to be very
 573 useful in many scientific models.

574 4 DISCUSSION

575 The three main goals in the development of Leibniz are (1) its usability as a digital scientific notation
 576 embedded in informal narratives, (2) generality in not being restricted to a narrowly defined scientific
 577 domain, and (3) simplicity of implementation in scientific software. While the current state of Leibniz,
 578 and in particular the small number of test applications that have been tried, do not permit a final judgment
 579 on how well these goals were achieved, it is nevertheless instructive to analyze *which* features of Leibniz
 580 are favorable to reaching these goals and how Leibniz compares to the earlier digital scientific notations
 581 reviewed in section 2.5.

582 The key feature for embedding is the highly declarative nature of Leibniz. The declarations that
 583 define a context and the values that build on them (terms and equations) can be inserted in arbitrary order
 584 into the sentences of a narrative. Verification at the informal-formal borderline is as well supported by
 585 Leibniz as by traditional mathematical notation. None of the digital scientific notations in use today
 586 shares this feature. Order matters only for rewrite rules, which has not appeared to be a limitation in the
 587 experiments conducted so far. Leibniz permits to write rules as assets identified by unique labels, and
 588 then assemble a list of named assets into a rule set for rewriting, but so far this feature has not found a

Functions

1 Real functions of one variable

The sort $\mathbb{R} \rightarrow \mathbb{R}$ describes real functions of one real variable. Function application is defined by $\mathbb{R} \rightarrow \mathbb{R}[\mathbb{R}] : \mathbb{R}$. Note that this implies that the domain of the function is the full set of real numbers, which excludes functions with singularities as well as partial functions.

It is convenient to provide basic arithmetic on functions:

- $f: \mathbb{R} \rightarrow \mathbb{R} + g: \mathbb{R} \rightarrow \mathbb{R} : \mathbb{R} \rightarrow \mathbb{R}$ with
 $(f + g)[x] \Rightarrow f[x] + g[x]$
 $\forall x: \mathbb{R}$
- $f: \mathbb{R} \rightarrow \mathbb{R} - g: \mathbb{R} \rightarrow \mathbb{R} : \mathbb{R} \rightarrow \mathbb{R}$ with
 $(f - g)[x] \Rightarrow f[x] - g[x]$
 $\forall x: \mathbb{R}$
- $f: \mathbb{R} \rightarrow \mathbb{R} \times g: \mathbb{R} \rightarrow \mathbb{R} : \mathbb{R} \rightarrow \mathbb{R}$ with
 $(f \times g)[x] \Rightarrow f[x] \times g[x]$
 $\forall x: \mathbb{R}$
- $s: \mathbb{R} \times g: \mathbb{R} \rightarrow \mathbb{R} : \mathbb{R} \rightarrow \mathbb{R}$ with
 $(s \times g)[x] \Rightarrow s \times g[x]$
 $\forall x: \mathbb{R}$

We do not define division as this requires more elaborate definitions to handle the case of functions with zeros.

Function composition is defined by

$$f: \mathbb{R} \rightarrow \mathbb{R} \circ g: \mathbb{R} \rightarrow \mathbb{R} : \mathbb{R} \rightarrow \mathbb{R} \text{ with}$$

$$(f \circ g)[x] \Rightarrow f[g[x]]$$

$$\forall x: \mathbb{R}$$

by Konrad Hinsén

Context $\mathbb{R} \rightarrow \mathbb{R}$
 uses *builtins/real-numbers*

Figure 6. The beginning of the document `functions` referred to by the predator-prey example. The full version is [available online](#).

589 good use.

590 Visual highlighting of the formal parts of a narrative (the blue and green background colors) allows
 591 readers to spot easily which parts of a narrative can affect a computation. Moreover, the reader can be
 592 assured that the internal coherence of all such highlighted information has been verified by the Leibniz
 593 authoring tool. For example, an equation typeset on a blue background is guaranteed to use only oper-
 594 ators declared in the context and the sorts of all terms have been checked for conformity. In this way,
 595 Leibniz actively supports human verification by letting the reader concentrate on the scientific aspects.

596 Generality is achieved by Leibniz not containing any scientific information and yet encapsulating
 597 useful foundations for expressing it. These foundations are term algebras, equational logic, and num-
 598 bers as built-in terms. This is an important difference in comparison to scientific DSLs. In fact, the
 599 analogue of a scientific DSL is not Leibniz, but a set of domain-specific Leibniz contexts. The common
 600 foundation makes it possible to combine contexts from different domains, which is difficult with DSLs
 601 designed independently. General-purpose programming languages follow the same approach as Leibniz
 602 in providing domain-neutral semantic foundations for implementing algorithms. These foundations are
 603 usually lambda calculus, algebraic data types, and a handful of built-in basic data types such as numbers
 604 and character strings. Leibniz' main advantage is in this respect is that equational logic is a more useful
 605 foundation for scientific knowledge than lambda calculus.

606 The principle of factoring out application-independent structure and functionality has a practically
 607 successful precedent in data languages such as XML (Bray et al., 2006). The foundation of XML is a
 608 versatile data structure: a tree whose nodes can have arbitrary labels and textual content. XML defines
 609 nothing but the syntax for this data structure, delegating the domain-specific semantics to schemas. The
 610 combination of data referring to different schemas is made possible by the XML namespace mechanism.
 611 The machine-facing side of Leibniz can be thought of as a layer in between the pure syntax of XML

612 and domain-specific scientific knowledge, providing a semantic foundation for scientific models and
613 methods.

614 The separation of syntax and semantics in XML is reflected by tools that process information stored
615 in XML-based formats. Domain-specific tools can delegate parsing and a part of validation to generic
616 parsers and schema validators. This same principle is expected to ensure simplicity of implementation for
617 Leibniz. Validating and rewriting terms are generic tasks that can be handled by a domain-independent
618 Leibniz runtime library. Assuming Leibniz is widely adopted, optimized Leibniz runtimes will become
619 as ubiquitous as XML parsers.

620 Another aspect of Leibniz that makes it easier to use in scientific software is its separation of a
621 machine-oriented syntax based on XML from the representations that human users interact with. In
622 contrast, for general-purpose programming languages and stand-alone DSLs, syntax is designed to be
623 an important part of the user interface. This makes it difficult to extract and analyze information stored
624 in a program, because any tool wishing to process the source code must deal with the non-trivial syntax
625 designed for human convenience. Moreover, suitable parsers are rarely available as reusable libraries.

626 5 CONCLUSIONS AND FUTURE WORK

627 The work reported in this article has focused on the human-computer interaction aspect of digital scien-
628 tific notations, and in particular on the embedding of digital scientific notations in scientific discourse
629 with the goal of facilitating verification by humans. First experiments with Leibniz have shown that it
630 can be embedded in informal discourse much like traditional mathematical notation. It can therefore
631 be expected that human verification will work in the same way, at least for code that can be structured
632 as a sequence of sufficiently short sections. Only further practice can show if this is possible for more
633 complex scientific models.

634 The semantics of the initial version of Leibniz were somewhat arbitrarily chosen to be a subset of
635 Maude, which looked like a good starting point for first experiments. However, these experiments sug-
636 gest that the language is currently too minimalist for productive use in computer-aided research. In
637 particular, the lack of predefined collections (lists, sets) makes it cumbersome to use for many applica-
638 tions, such as the force fields mentioned as a motivation in section 2.1. Another aspect of the language
639 that deserves further attention is the sort system summarized in section 3. Many common value con-
640 straints in scientific applications would require value-dependent sorts, in the spirit of dependent types.
641 Examples are the compatibility of units of measure, or of the dimensions of matrices.

642 Another aspect that will require further work is the definition of the role of digital scientific notations
643 such as Leibniz in the ecosystem of scientific software. A theoretically attractive but at this time not
644 very feasible approach would have software tools read specifications from Leibniz documents and per-
645 form the corresponding computations. In addition to the fundamental issue that we do not have general
646 automatic methods for turning specifications into efficient implementations, there is the practical issue
647 that today's scientific computing universe is very tool-centric, with users often adapting their research
648 methodology to their tools rather than the inverse. A more realistic short-term scenario sees Leibniz used
649 in the documentation of software packages, which could then contain a mixed informal/formal specifica-
650 tion of the software's functionality. This specification could be verified scientifically by human readers,
651 and the software could be verified against it using techniques such as testing or formal verification. A sci-
652 entific study would be documented in another Leibniz document that uses contexts from the software's
653 specification.

654 Leibniz and digital scientific notations similar to it are also promising candidates for unifying sym-
655 bolic and numerical computation in science. As the example of the predator-prey equations shows,
656 Leibniz can represent not only computations, but also equations. A computer algebra system could pro-
657 cess equations formulated in Leibniz, producing results such as analytical solutions, approximations, or
658 numerical solution algorithms, which could all be expressed in Leibniz as well. Corresponding Leibniz
659 contexts could be derived automatically from the OpenMath standard (OpenMath society, 2018).

660 Finally, there is obviously a lot of room for improvement in the tools used by authors and readers
661 for interacting with Leibniz content. Ideally, the author and reader would work with identical or very
662 similar views, which should be more interactive than plain text or HTML documents. Much inspiration,
663 and probably also implementation techniques, can be adopted from computational notebooks and other
664 innovations in scientific publishing that are currently under development.

665 ACKNOWLEDGMENTS

666 I am grateful to Prof. Shriram Krishnamurthi for recommending the adoption of the infix operator rules
667 from the Pyret language, and for suggesting the predator-prey equations as an example for demonstration.

668 REFERENCES

- 669 (2018). Programming in Pyret.
- 670 Beg, M., Pepper, R. A., and Fangohr, H. (2017). User interfaces for computational science: A domain
671 specific language for OOMMF embedded in Python. *AIP Advances*, 7(5):056025.
- 672 Boute, R. (2005). Functional declarative language design and predicate calculus: A practical approach.
673 *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):988–1047.
- 674 Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2006). Extensible
675 Markup Language (XML) 1.1 (Second Edition).
- 676 Bui, T. M. A., Ziane, M., Stinckwich, S., Ho, T. V., Roche, B., and Papoulias, N. (2016). Separation of
677 concerns in epidemiological modelling. In *Companion Proceedings of the 15th International Confer-*
678 *ence on Modularity*, pages 196–200. ACM Press.
- 679 Cieplak, P. and Kollman, P. (1996). A second generation force field for the simulation of proteins, nucleic
680 acids, and organic molecules (vol 117, pg 5179, 1995). *J Am Chem Soc*, 118(9):2309–2309.
- 681 Claerbout, J. and Karrenbach, M. (1992). Electronic documents give reproducible research a new mean-
682 ing. In *SEG Technical Program Expanded Abstracts 1992*, SEG Technical Program Expanded Ab-
683 stracts, pages 601–604. Society of Exploration Geophysicists.
- 684 Clavel, M., Durán, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J., and Quesada, J. F.
685 (2002). Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*,
686 285(2):187–243.
- 687 DeVito, Z., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P., Joubert, N., Palacios, F., Oakley, S., Med-
688 ina, M., Barrientos, M., Elsen, E., Ham, F., and Aiken, A. (2011). Liszt: A domain specific language
689 for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for*
690 *High Performance Computing, Networking, Storage and Analysis*, page 1. ACM Press.
- 691 Eklund, A., Nichols, T. E., and Knutsson, H. (2016). Cluster failure: Why fMRI inferences for spatial
692 extent have inflated false-positive rates. *PNAS*, 113(28):7900–7905.
- 693 Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt,
694 S., and Herbstritt, M. (2015). The Racket Manifesto. Technical report, Schloss Dagstuhl - Leibniz-
695 Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.
- 696 Flatt, M., Barzilay, E., and Findler, R. B. (2009). Scribble: Closing the book on ad hoc documentation
697 tools. In *ACM Sigplan Notices*, volume 44, pages 109–120. ACM.
- 698 Fritzon, P. and Engelson, V. (1998). Modelica — A unified object-oriented language for system mod-
699 eling and simulation. In Goos, G., Hartmanis, J., van Leeuwen, J., and Jul, E., editors, *ECOOP'98*
700 *— Object-Oriented Programming*, volume 1445, pages 67–90. Springer Berlin Heidelberg, Berlin,
701 Heidelberg.
- 702 Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J.-P. (2000). Introducing OBJ. In
703 Hinchey, M., Goguen, J., and Malcolm, G., editors, *Software Engineering with OBJ*, volume 2, pages
704 3–167. Springer US, Boston, MA.
- 705 Herndon, T., Ash, M., and Pollin, R. (2014). Does high public debt consistently stifle economic growth?
706 A critique of Reinhart and Rogoff. *Cambridge Journal of Economics*, 38(2):257–279.
- 707 Hinsen, K. (2000). The molecular modeling toolkit: A new approach to molecular simulations. *J Comput*
708 *Chem*, 21(2):79–85.
- 709 Hinsen, K. (2014). Computational science: Shifting the focus from tools to models. *FI1000Research*,
710 3:101.
- 711 Imbert, C. (2017). Computer Simulations and Computational Models in Science. In Magnani, L. and
712 Bertolotti, T., editors, *Springer Handbook of Model-Based Science*, pages 735–781. Springer Interna-
713 tional Publishing, Cham.
- 714 Jarrold, M. F. (2007). Helices and Sheets in vacuo. *Phys Chem Chem Phys*, 9(14):1659–1671.
- 715 Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick,
716 J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., and Willing, C. (2016). Jupyter Notebooks
717 – a publishing format for reproducible computational workflows. In Loizides, F. and Schmidt, B.,

- 718 editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90.
719 IOS Press.
- 720 Knuth, D. E. (1974). Computer Science and its Relation to Mathematics. *The American Mathematical*
721 *Monthly*, 81(4):323–343.
- 722 Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97–111.
- 723 Lamport, L. (1994). *LATEX: A Document Preparation System: User's Guide and Reference Manual*.
724 Addison-Wesley Pub. Co, Reading, Mass, 2nd ed edition.
- 725 Langtangen, H. P. (2012). *A Primer on Scientific Programming with Python*. Number 6 in Texts in
726 Computational Science and Engineering. Springer, Heidelberg, third edition edition.
- 727 Merali, Z. (2010). Computational science: ...Error. *Nature*, 467(7317):775–777.
- 728 Miller, G. (2006). A Scientist's Nightmare: Software Problem Leads to Five Retractions. *Science*,
729 314(5807):1856–1857.
- 730 OpenMath society (2000-2018). OpenMath. <https://openmath.github.io/>.
- 731 Reinhart, C. M. and Rogoff, K. S. (2010). Growth in a Time of Debt. *American Economic Review*,
732 100(2):573–578.
- 733 Reißer, S., Poger, D., Stroet, M., and Mark, A. E. (2017). Real Cost of Speed: The Effect of a Time-
734 Saving Multiple-Time-Stepping Algorithm on the Accuracy of Molecular Dynamics Simulations. *J.*
735 *Chem. Theory Comput.*, 13(6):2367–2372.
- 736 Roberts, K. V. (1969). The publication of scientific Fortran programs. *Computer Physics Communica-*
737 *tions*, 1(1):1–9.
- 738 Smith, A. M., Niemeyer, K. E., Katz, D. S., Barba, L. A., Githinji, G., Gymrek, M., Huff, K. D., Madan,
739 C. R., Mayes, A. C., Moerman, K. M., Prins, P., Ram, K., Rokem, A., Teal, T. K., Guimera, R. V.,
740 and Vanderplas, J. T. (2017). Journal of Open Source Software (JOSS): Design and first-year review.
741 *arXiv:1707.02264 [cs]*.
- 742 Soergel, D. A. W. (2014). Rampant software errors undermine scientific results. *F1000Research*, 3:303.
- 743 Stodden, V., McNutt, M., Bailey, D. H., Deelman, E., Gil, Y., Hanson, B., Heroux, M. A., Ioannidis,
744 J. P. A., and Tauber, M. (2016). Enhancing reproducibility for computational methods. *Science*,
745 354(6317):1240–1241.
- 746 Sussman, G. J. and Wisdom, J. (2002). The Role of Programming in the Formulation of Ideas. Technical
747 report, MIT Artificial Intelligence Laboratory.
- 748 Sussman, G. J. and Wisdom, J. (2014). *Structure and Interpretation of Classical Mechanics*. The MIT
749 Press, Cambridge, Massachusetts, second edition edition.
- 750 Sussman, G. J., Wisdom, J., and Farr, W. (2013). *Functional Differential Geometry*. The MIT Press,
751 Cambridge, MA.
- 752 Wolfram Research Inc (1988). Mathematica, Version 1.0. Champaign, IL, 1988.