Building a scientific workbench in Pharo

Konrad Hinsen, Serge Stinckwich

Building a scientific workbench in Pharo

Konrad Hinsen
Centre de Biophysique Moléculaire
CNRS
Orléans, France
konrad.hinsen@cnrs.fr
Division Expériences
Synchrotron SOLEIL
Gif sur Yvette, France

Serge Stinckwich
Sorbonne Université, IRD, Unité de Modélisation
Mathématiques et Informatique des Systèmes
Complexes,UMMISCO, F-93143 Bondy, France.
Université de Yaoundé I, Yaoundé, Cameroon
Université de Caen Normandie, Caen, France
serge.stinckwich@ird.fr

Abstract

Over the last years, the need for better user interfaces in scientific computing has become apparent by the enormous growth in popularity of computational notebooks. However, the user experience provided by these notebooks is very limited compared to the live programming environments of Smalltalk, and first studies on how they are actually used in research studies point out various weaknesses. We present the first steps we have taken towards building a scientific workbench in Pharo, based on the Glamorous Toolkit, and outline future developments towards that goal.

Keywords Pharo, scientific computing, computational document

1 Introduction

Research in all domains of science is increasingly relying on computational methods. At the minimum, figures and tables used to communicate results in scientific publications are produced with the help of software, but more elaborate computational data processing techniques, often involving statistical methods, are routinely applied by scientists who consider their core work to be experimental or theoretical rather than computational. At the other extreme, whole subfields dedicated to the development and use of computational methods are firmly established in all the natural sciences and are becoming more and more important in the humanities.

The driving force behind the adoption of computational methods in scientific research is the new possibilities they offer for dealing with larger amounts of data and with more complex models. However, there is also a downside to the introduction of computers, which in the philosophy of science is referred to as the *epistemic opacity* of computation. Today, scientists routinely apply computational methods that they do not fully understand and may not even be aware of. One factor contributing to this problem is the increasing complexity of computational methods. However, a factor that in our opinion is much more important is the complexity and opacity of today's computing technology, in which understandability has so far not played a major role.

IWST 19, August 27–29, 2019, Cologne, Germany
2019.

Recent years have seen an increasing awareness of this problem in many scientific disciplines. Its most visible manifestation is the "reproducible research" movement, whose goal is to extend scientific publications by the inclusion of all the code and input data required to reproduce the presented results. Many obstacles of both technical and social nature make this a surprisingly difficult goal to achieve in practice. It requires that all research code is published in such a way that someone else can install and run it. But reproducibility is only one aspect of the problem. A computation can be fully reproducible and yet remain incomprehensible because of poorly written code or documentation.

The challenge for the coming years is to develop new ways of doing computational science that emphasize comprehensibility during all phases: the development of computational methods, their application to specific research questions, and the communication of the results. We believe that the Smalltalk tradition of encouraging transparency and explorability of computational systems is an excellent starting point for future work in this field. We describe the first steps that we have taken to design and implement a scientific workbench in Pharo (an open-source Smalltalk implementation), and we outline the future work that remains to be done.

2 Computational notebooks

For several decades, interactive scientific computing environments have been designed around variants of the Read-Eval-Print Loop (REPL) introduced in the 1950s with Lisp. In 1988, Mathematica [18] introduced the notebook interface, which can be considered a fusion of the REPL with Donald Knuth's concept of literate programming [20]. A computational notebook is a sequence of input, output, and text parts commonly called "cells" (see Fig. 1 for an example). Text cells have no impact on the computation, they can be considered rich-text comments. Input cells contain code snippets that can be fed into a language interpreter, either individually or in larger groups. An output cell is inserted immediately below each input cell upon execution, and contains the result of the code snippet, using a textual or a graphical representation.

In the initial stage of exploring a dataset or a scientific model, a scientist creates new input cells for each command,

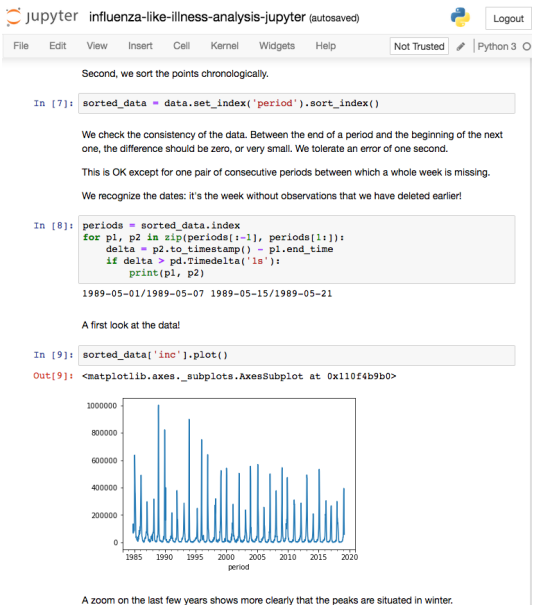

Figure 1. A Jupyter notebook consisting of input, output, and text cells. Input cells can be executed manually, one by one, meaning that they are not necessarily run in textual order. Therefore they are marked by 'In' followed by a number in brackets that indicates the order of execution. Output cells are marked by 'Out' plus that same number in execution order. When a notebook is run as a whole, input cells are executed in textual order. Readers must verify the execution order by inspecting the numbers, because results can depend on it.

using the notebook much like a traditional REPL. Text cells are used sparingly for adding comments where deemed useful for later work. As the scientist gains a better understanding of the problem, he or she starts to revise earlier input cells, typically resulting in more compact and more focused code. At the same time, text cells are expanded into a more complete documentation of the computation, intended initially for collaborators. Further editing iterations transform the notebook into a publishable document that a reader unfamiliar with the work should be able to understand.

The reader typically starts by reading the notebook in the order of presentation of the cells. He or she can also explore some aspect in more detail by modifying (or inserting) input cells and executing them. This does, however, require a prior re-execution of the complete notebook, in order to make all the intermediate results available in memory. An inconvenience of this exploration strategy is that it modifies the original notebook. After an intensive exploration session, it is often hard to see which input and output cells were part of the original notebook and which were added or modified by the reader. This is a direct consequence of the linear nature of the computational notebook.

This linear structure of a notebook, which follows from the linear structure of the computation even though, paradoxically, the latter is not enforced, also lies at the heart of other problems that become apparent when notebooks are used for anything but the simplest tasks. A recent study [25] of more than a million notebooks in public GitHub repositories found many indicators of poor quality, some of which support earlier criticisms of notebooks encouraging bad software engineering habits [11, 23, 27, 34]. Overall, the problems stemming from the imposed linearity can be classified as

1. Computational issues: the computation in a notebook is a linear sequence of commands modifying shared global state. This prevents the application of best practices from software engineering, in particular any attempt to modularize the code.
2. Documentation issues: the discussion of the science behind the computation must follow the order of the computation and thus cannot be structured for clarity of exposure. For example, the documentation of a data analysis must start with the rather boring details of preprocessing, which in a traditional paper would be relegated to an appendix.

Recently, the popularity of computational notebooks has surged in the wake of the Reproducible Research movement. It is often claimed that notebooks "improve reproducibility", but such claims rarely clarify what exactly they compare notebooks to. In fact, notebooks improve reproducibility compared to traditional scientific articles, which contain no code at all. However, notebooks are no more reproducible than plain scripts, given that the information they contain in addition to scripts (text and output cells) is for human consumption only and has no impact on reproducibility. In practice, today's notebook implementations require notebook authors to take various precautions to ensure reproducibility, which are exactly the same as required for making scripts reproducible: a detailed documentation of the software environment that was used, listing all dependencies with detailed version information.

3 Leveraging the Smalltalk heritage to improve on the notebook

Two of the outstanding features that Smalltalk has had since its beginnings are a graphical user interface and an integrated development environment. The place of the terminal-based REPL is taken by workspaces (also called playgrounds) for editing and executing code snippets, and inspectors for exploring results. This toolset encourages experimentation by running small independent code snippets that have neither shared state (other than the global system environment) nor a natural order. In software development, these experimental code snippets are often the starting point of test cases, or find their way into the code documentation as examples.

Computer-aided research always involves some software development, but has some additional requirements. In particular, the focus of attention is not on code, but on data. This can be raw data from observations or experiments, but also pre-processed data or model-derived data (e.g. by way of simulation). The complete documentation of a scientific study consists of

1. Input data (observations, experiments)
2. Code written for processing or generating data
3. Processed and generated data
4. A narrative relating the data and code to the scientific context: motivation, references to earlier work, origin of experimental data, assumptions made, conclusions, etc.

Contrary to code snippets used for illustrating and testing software, the code snippets that process and generate data are not independent. Code snippets and datasets are nodes in a data dependency graph, which must be easy to follow for a reader [32]. What must be added to a Smalltalk environment to turn it into a scientific workbench is thus (1) data dependency management and (2) a documentation facility for writing narratives that can refer to or embed code and data.

One recent innovation in Smalltalk environments is the Glamorous Toolkit [10], which is a complete redesign of the traditional Smalltalk tools: browsers, inspectors, debuggers. A central feature is a rich document type based on the markup language Pillar [5]. Such documents can contain inline code snippets, much like computational notebooks, but they can also refer to arbitrary objects, including classes and individual methods. Objects referred to can be displayed and even edited right in the document. They can also be opened in separate inspector panes for more extensive exploration. These features make the Glamorous Toolkit a promising basis for a scientific workbench that combines, in a single environment, the development of scientific software and its application to the exploration of datasets and models. Moreover, the "moldable tools" approach [7] should make such an extension relatively straightforward.

The ActivePapers Pharo edition [14] is such an extension of the Glamorous Toolkit to data- or model-centric computations, building on the experience gained with earlier work in the ActivePapers project [12]. An ActivePaper is, by definition, a package combining the code, data, and documentation that describe together a computational scientific study. The first two editions of ActivePapers focused on reproducibility and high-performance computing, storing code and data in a single HDF5 file [21] for easy transferability. In the Pharo edition, an ActivePaper is a singleton, with the class containing the code and its unique instance holding the data. The documentation part takes the form of a Wiki whose individual pages are the rich documents introduced by the Glamorous Toolkit. The Wiki pages are stored in methods, borrowing a trick used by Pharo's help system. The main reason for this choice is the automatic inclusion of the Wiki pages into the version control repository. The example in appendix A illustrates further details of how an ActivePaper functions.

The integration of ActivePapers with the Glamorous Toolkit is mostly achieved using the latter's extension mechanisms (an example is shown in Fig. 2, a demo video is available as well [15]). The rich documents and the dependency graph are displayed using inspector plugins, with an additional Pillar annotation type providing links to Wiki pages. ActivePapers also adds a few user interface elements, in particular a specialized playground for working in the context of the object, with full access to its methods and instance variables. This playground is what readers of an ActivePaper use to explore its contents without having to modify them. It is also the tool that authors use for testing code before adding it to the ActivePaper's methods.

The linear command sequence with shared global state of a notebook is replaced by a workflow consisting of named code snippets called scripts, which are implemented as methods containing a special pragma. Their shared state is defined by the ActivePaper's instance variables, which are implemented as slots that track read and write accesses. This makes it possible to generate a data dependency graph during the execution of the scripts (the right-hand pane in Fig. 2). The workflow is completely independent of the documentation layer in the Wiki. Each layer can be structured according to its own logic. The Wiki pages can reference scripts and datasets via the standard Pillar annotations provided by the Glamorous Toolkit.

In addition to having a more flexible structure, ActivePapers in Pharo have another major advantage over computational notebooks: they are standard Pharo objects. One consequence is that the data and code they contain can be used by other Pharo code, and in particular other ActivePapers. In contrast, notebooks are always top-level code that cannot be integrated into larger code assemblies. Moreover, there is no boundary in terms of development tools between an ActivePaper and the Pharo libraries it builds on. All the development, inspection, and refactoring tools of Pharo can therefore be applied. In contrast, most computational notebook implementations (in fact all except for Mathematica) treat notebooks as entities completely separate from library code. Moving code across the notebook-library boundary thus implies using different user interfaces, even if they are implemented in the same tool as is the case for Emacs or JupyterLab.

4 Existing computational platforms and frameworks in Pharo

A good scientific workbench must not only provide tools, but also support libraries for data management, data analysis,

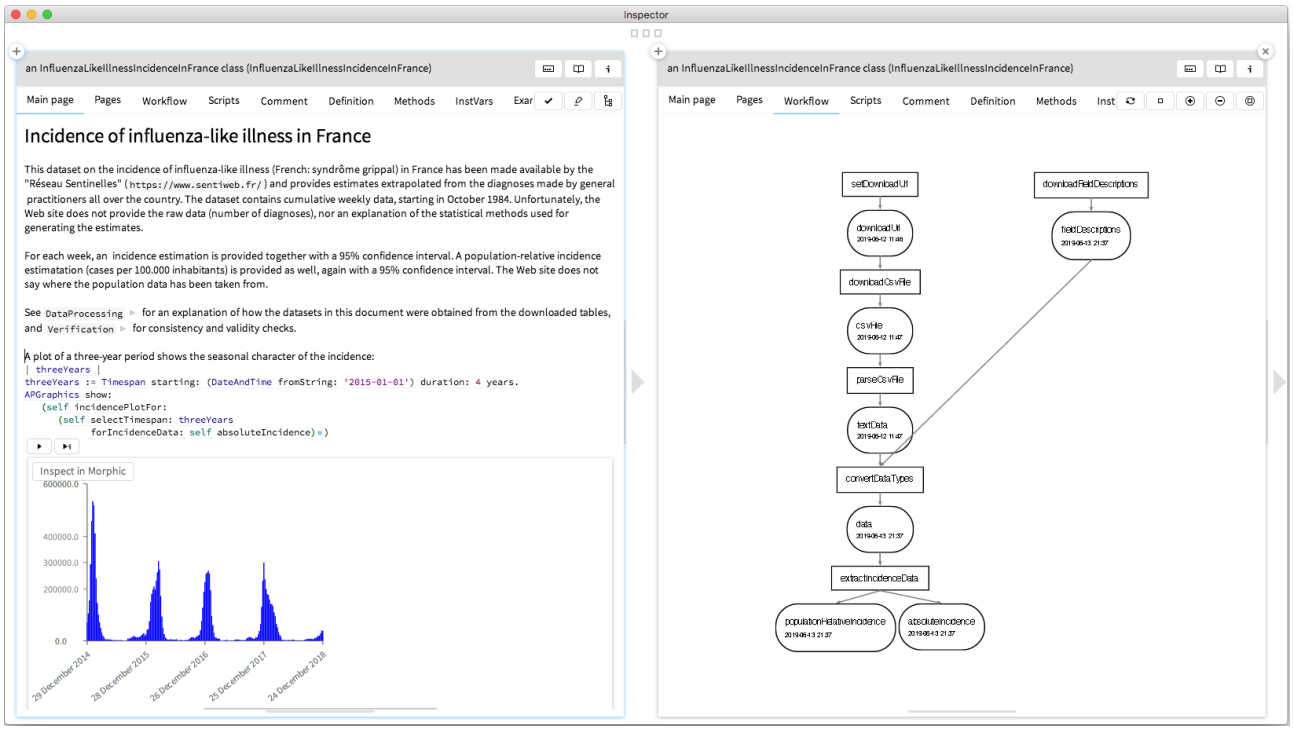

Figure 2. The document view and workflow view of an ActivePaper. These are two of the four views specific to the class `APActivePaper`, each of which being defined by a method containing the `<gtView>` pragma. The other two are "Pages" (a list of Wiki pages in the document) and "Scripts", a view permitting to read, edit, and run the scripts that make up the workflow.

and for the implementation of scientific models and methods. For now, Pharo cannot compete with the large scientific ecosystems that have developed around popular languages such as Python or R. However, two key libraries provide sufficient support for simple data analysis tasks, which are extremely common in many scientific disciplines:

- PolyMath [26], a scientific library similar to NumPy or SciPy in Python. PolyMath provides a lot of functionalities: complex numbers, quaternions, random number generators, numerical methods, ordinary differential equations (ODE) solvers, and more.
- DataFrame [8], a tabular data structure for data analysis in Pharo. The DataFrame project is similar to the Pandas library in Python or the built-in data.frame class in R.

It is also worth mentioning more domain-specific libraries, which show the potential of this language to build a more versatile scientific workbench in the future:

- Pharo bindings to Tensorflow [31], which allow to use the TensorFlow C API and kernels. More elaborate frameworks like Keras are now also available from Pharo via PythonBridge [28]
- Kendrick [4], a domain-specific language focused on epidemiology modelling that supports modularity through clear separation of concerns, hence fostering reproducibility and reuse of models and simulations. It allows epidemiologists to craft their models and simulations more easily. Kendrick use PolyMath for solving ordinary differential equations.
- CORMAS [2] (for COmmon pool Resources and Multi-Agent Systems), an Agent-Based Modeling platform based on Pharo and dedicated to natural and common-pool resources management. CORMAS' main intent is to facilitate the design of ABM as well as the monitoring and analysis of agent-based simulation scenarios. This has been done by taking an innovative direction oriented towards participatory modeling, i.e. the collective design of models as an appropriate medium for fostering interdisciplinarity, and interactive simulations involving several stakeholders who interact with a simulation by acting directly on their agents.
- BioSmalltalk[22] and PhyloclassTalk, (Open-source phylogenetics workbench) which are bioinformatics platforms. They provide tokenizers, parsers and formatters to manipulate biological sequences and data from databases.

5 Future work

The ActivePapers Pharo edition is still at the prototype stage at this time. One issue that remains to be addressed is reproducibility. Pharo's package management infrastructure, which already provides dependency analysis, looks like a promising basis for a user interface that allows scientists to ensure the reproducibility of their work without having to go into the arcane technical details required by today's common approaches based on package managers at the operating system level. Another important issue is exporting a computational document in a format suitable for publishing. We expect the XDoc subsystem of the Glamorous Toolkit to be a good basis for this. Finally, the current implementation lacks a straightforward way to store the data held in an ActivePaper permanently. Code and documentation are easily stored in a version-controlled repository because they are defined by Pharo classes. Datasets, however, can be arbitrary Pharo objects, and can potentially be very big. Moreover, it is highly desirable to make these datasets accessible from other languages than Pharo. For this reason, efficient and general but Pharo-specific serialization protocols such as Fuel [9] are not a good choice.

The management of scientific data is currently undergoing a transition motivated by the increasing desire to publish and share datasets, and the simultaneous rise of distributed processing strategies in grid and cloud computing. For small datasets stored in text files using formats such as CSV, XML, or JSON, data management is not an issue because Pharo already supports these formats and the publication of the data is handled via standard Web techniques that are also already implemented in Pharo. Larger datasets are typically stored in specific binary formats such as HDF5 [21], which Pharo does not currently support (though an existing HDF5 interface for Squeak [6] could probably be adapted). However, these datasets are also most affected by the ongoing transition, and it might be more profitable for the Pharo community to concentrate on future distributed storage technologies such as IPFS [1] or dat [29].

The main limitation of Pharo as a scientific workbench is the relatively small number of libraries supporting scientific computing, compared to established ecosystems such as Scientific Python. However, there are many computationally simple use cases in data analysis for which today's libraries, together with currently on-going work on PolyMath and DataFrames, are fully sufficient. These application domains are therefore ideal targets to test and improve the user interface framework provided by ActivePapers. For more domain-specific work, a case by case analysis must be conducted to identify the most promising strategy: implement well-known methods in Pharo, or provide interfaces to existing code in other languages via the FFI (Foreign Function Interface) or higher-level techniques such as PythonBridge.

Finally, Pharo is particularly suited to the implementation of new approaches to scientific computing that require both fundamental algorithms and corresponding user interfaces. An example is the reification of scientific models, which is the goal of the digital scientific notation Leibniz whose Pharo implementation is work in progress [16]. Prior experiments using traditional programming systems [13] have shown the importance of having editors, inspectors, and debuggers for scientific models, all of which are very difficult to implement in programming systems based on text files and compilers.

6 Related work

The idea of using a Smalltalk live programming environment as a user interface for scientific computing has come up before [24] and its utility was demonstrated with a few applications [3, 33]. This happened at the same time that Mathematica introduced the computational notebook. Both approaches to introducing interactivity were revolutionary at a time when the state of the art in scientific computing was compiled languages with their edit-compile-run cycles, and graphical user interfaces limited to plotting and other simple visualization tasks.

Recent popular notebook implementations are Jupyter [19], which supports a wide range of programming languages (including Pharo[1]), and RMarkdown [17] for the R language that is widely used for data analysis tasks. The Org-Mode package of the Emacs editor provides an implementation of computational notebooks as part of a powerful general document processing system [30], which makes for a flexible and highly customizable computing environment which however suffers from poor support for graphics. Two more recent and more experimental notebook implementations are purely browser-based and thus implemented in JavaScript: Iodide (https://github.com/iodide-project/iodide) aims at multi-language scientific computation via WebAssembly, and Observable (https://observablehq.com/) makes the code and data in notebooks re-usable through a modularized data-flow approach.

7 Conclusion

We hope to have demonstrated the potential of Smalltalk-style live programming environments in general, and of Pharo plus Glamorous Toolkit in particular, as a powerful user interface for computational science. Furthermore, we hope to develop this idea into a practically usable tool, while at the same time making the live programming approach more generally known and potentially adopted by other languages and environments.

In addition to the benefits we have outlined in this paper, we expect Smalltalk environments to improve computational science in another, less obvious way: by reducing the epistemic opacity of computational methods that we mentioned

[1] https://github.com/jmari/JupyterTalk/

in the introduction. In all of today's scientific computing environments, there is a strong boundary between notebooks, scripts, and interactive REPLs on one hand, and library code on the other hand. A notebook that applies a statistical method from a code library has no way to include or refer to that code in order to explain the method. It is up to the reader to figure out where the corresponding source code is (assuming it is available at all), and use an appropriate tool to inspect it. In this way, today's environments actively discourage readers from exploring the models and methods underlying computational work. Smalltalk systems do not have any such barrier, and in the contrary assist their users with exploring all the code in the system as far as they like. The computational documents implemented by the Glamorous Toolkit perpetuate this tradition by permitting links to arbitrary objects and even embedding arbitrary object inspectors. Unlike a notebook, an ActivePaper can easily contain a page describing a model or method implemented in library code, adding examples if necessary. In the long run, this may well be the most important contribution of Smalltalk to computational science.

References

[1] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *arXiv:1407.3561 [cs]* (July 2014). arXiv:cs/1407.3561

[2] Pierre Bommel, Nicolas Becu, Christophe Le Page, and François Bousquet. 2016. Cormas: An Agent-Based Simulation Platform for Coupling Human Decisions with Computerized Dynamics. In *Simulation and Gaming in the Network Society*, Toshiyuki Kaneda, Hidehiko Kanegae, Yusuke Toyoda, and Paola Rizzi (Eds.). Vol. 9. Springer Singapore, Singapore, 387–410. https://doi.org/10.1007/978-981-10-0575-6_27

[3] Toufic I. Boubez, Andy M. Froncioni, and Richard L. Peskin. 1992. A Prototyping Environment for Differential Equations. *ACM Trans. Math. Softw.* 18, 1 (March 1992), 1–10. https://doi.org/10.1145/128745.128746

[4] Mai Anh Bui T., Nick Papoulias, Serge Stinckwich, Mikal Ziane, and Benjamin Roche. 2019. The Kendrick Modelling Platform: Language Abstractions and Tools for Epidemiology. *BMC Bioinformatics* 20, 1 (Dec. 2019), 312. https://doi.org/10.1186/s12859-019-2843-0

[5] Damien Cassou, Cyril Ferlicot Delbecque, Yann Dubois, and Thibault Arloing. 2015. Documenting and Presenting with Pillar. In *Enterprise Pharo*. Square Bracket Associates.

[6] Nicolas Cellier. 2018. Smalltalk Tools for Engineering and Mathematics. http://www.squeaksource.com/STEM.html.

[7] Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel. 2017. Moldable Tools for Object-Oriented Development. In *Present and Ulterior Software Engineering*, Manuel Mazzara and Bertrand Meyer (Eds.). Springer International Publishing, Cham, 77–101. https://doi.org/10.1007/978-3-319-67425-4_6

[8] DataFrame contributors. 2019. DataFrame. https://github.com/PolyMathOrg/DataFrame.

[9] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. 2014. Fuel: A Fast General Purpose Object Graph Serializer. *Softw. Pract. Exper.* 44, 4 (April 2014), 433–453. https://doi.org/10.1002/spe.2136

[10] Feenk. 2019. Glamorous Toolkit. https://gtoolkit.com/.

[11] Joel Grus. 2018. I Don't like Notebooks. https://conferences.oreilly.com/jupyter/jupny/public/schedule/detail/68282.

[12] Konrad Hinsen. 2015. ActivePapers: A Platform for Publishing and Archiving Computer-Aided Research. *F1000Research* 3, 289 (July 2015). https://doi.org/10.12688/f1000research.5773.3

[13] Konrad Hinsen. 2018. Verifiability in Computer-Aided Research: The Role of Digital Scientific Notations at the Human-Computer Interface. *PeerJ Computer Science* 4 (July 2018), e158. https://doi.org/10.7717/peerj-cs.158

[14] Konrad Hinsen. 2019. The ActivePapers Pharo Edition. http://www.activepapers.org/pharo-edition/.

[15] Konrad Hinsen. 2019. The Computational Notebook of the Future, Part 2. https://vimeo.com/339361206.

[16] Konrad Hinsen. 2019. Leibniz Runtime Library for Pharo. https://github.com/khinsen/leibniz-pharo/.

[17] RStudio Inc. 2016. R Markdown. https://rmarkdown.rstudio.com/.

[18] Wolfram Research Inc. 1988. Mathematica, Version 1.0. (1988). Champaign, IL, 1988.

[19] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a Publishing Format for Reproducible Computational Workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, IOS Press, Amsterdam, Netherlands, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

[20] Donald E Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (1984), 97–111.

[21] Quincey Koziol and Dana Robinson. 2018. HDF5. Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States). https://doi.org/10.11578/dc.20180330.1

[22] Hernán F. Morales and Guillermo Giovambattista. 2013. BioSmalltalk: A Pure Object System and Library for Bioinformatics. *Bioinformatics* 29, 18 (2013), 2355–2356.

[23] Alexander Mueller. 2018. 5 Reasons Why Jupyter NoteBooks Suck. https://towardsdatascience.com/5-reasons-why-jupyter-notebooks-suck-4dc201e27086.

[24] Richard L. Peskin, Sandra S. Walther, and Andy M. Froncioni. 1989. Smalltalk — the next Generation Scientific Computing Interface? *Mathematics and Computers in Simulation* 31, 4-5 (Oct. 1989), 371–381. https://doi.org/10.1016/0378-4754(89)90131-6

[25] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. [n. d.]. A Large-Scale Study about Quality and Reproducibility of Jupyter Notebooks. ([n. d.]), 11.

[26] PolyMath contributors. 2019. PolyMath. https://github.com/PolyMathOrg/PolyMath.

[27] Kirill Pomogajko. 2015. Why I Don't Like Jupyter (FKA IPython NoteBook). http://opiateforthemass.es/articles/why-i-dont-like-jupyter-fka-ipython-notebook/.

[28] Object Profile. 2018. PythonBridge. https://objectprofile.github.io/PythonBridge/.

[29] Danielle C. Robinson, Joe A. Hand, Mathias Buus Madsen, and Karissa R. McKelvey. 2018. The Dat Project, an Open and Decentralized Research Data Tool. *Sci Data* 5 (Oct. 2018). https://doi.org/10.1038/sdata.2018.221

[30] Eric Schulte and Dan Davison. 2011. Active Documents with Org-Mode. *Computing in Science & Engineering* 13, 3 (May 2011), 66–73. https://doi.org/10.1109/MCSE.2011.41

[31] Serge Stinckwich. 2019. Pharo Bindings for TensorFlow. https://github.com/PolyMathOrg/libtensorflow-pharo-bindings.

[32] Marijn van Vliet. 2019. Guidelines for Data Analysis Scripts. *arXiv:1904.06163 [cs]* (April 2019). arXiv:cs/1904.06163

[33] Sandra S. Walther and Richard L. Peskin. 1991. Object-Oriented Visualization of Scientific Data. *Journal of Visual Languages & Computing* 2, 1 (March 1991), 43–56. https://doi.org/10.1016/S1045-926X(05)80051-5

[34] Yihui Xie. 2018. The First Notebook War. https://yihui.name/en/2018/09/notebook-war/.

A Appendix: a simple example

The following commented Pharo code constructs a simple ActivePaper with two datasets and two scripts computing their values. It is provided as an illustration of how ActivePapers is implemented. Note that only the first step would normally be executed by running code from a playground, the remaining steps being performed using Pharo's development tools.

1. Create an ActivePaper as a subclass of `APActivePaper`:

   ```
   APActivePaper subclass: #ActivePaperExample.
   ```

2. Add a class comment containing the main page of the ActivePaper's Wiki, using Pillar markup:

   ```
   ActivePaperExample comment: '!A simple ActivePaper'
   ```

3. Add two instance variables:

   ```
   ActivePaperExample addInstVarNamed: 'x y'.
   ```

   The superclass `APActivePaper` converts instance variables to dataset slots whose read and write access it observes and regulates. All read accesses are traced in order to construct a data dependency graph. The superclass also creates read accessors for each dataset.

4. Add a script that defines dataset `x`:

   ```
   ActivePaperExample compile: 'setX
   <computes: #x>
   x := 42'
   classified: 'scripts'.
   ```

   What makes this method a script is the pragma `computes:`, which also states the dataset that is computed by this script. A script can contain multiple pragmas to signal that it computes multiple datasets. It is not allowed to change the value of any other dataset.

5. Add a script that defines dataset `y`:

   ```
   ActivePaperExample compile: 'computeY
   <computes: #y>
   y := x / 2'
   classified: 'scripts'.
   ```

   This script reads the value of `x`, creating a dependence of `y` on `x`.

6. Look up the value of dataset `y`:

   ```
   ActivePaperExample data y
   ```

   This code snippet invokes the read accessor for `y` that has been created automatically. It retrieves the script computing `y` (searching for the pragma) and executes it. When this script attempts to read `x`, which has not been set, the same mechanism is applied to retrieve the script computing `x` and execute it. All dataset computations are thus performed lazily.