Automated capture of experiment context for easier reproducibility in computational research

Andrew P. Davison

Unité de Neurosciences, Information et Complexité,
CNRS UPR 3293, Gif sur Yvette, France

Computing in Science and Engineering 14(4): 48-56, July-Aug. 2012
http://dx.doi.org/10.1109/MCSE.2012.41

Abstract

Reproducibility is part of the definition of the scientific method. In computational science, reproducibility has practical benefits in allowing code reuse and hence enabling more rapid progress and the ability to develop more complex models and analysis methods. However, it is widely recognized that published research in diverse scientific domains that rely on numerical computation is too infrequently reproducible, with some commentators speaking of a "credibility crisis" [4, 11]. In this article, I examine the reasons why reproducibility is difficult to achieve, argue that for computational research to become consistently and reliably reproducible requires that reproducibility become *easy* to achieve, as part of day-to-day research and not just for that subset of research that is published, and suggest a combination of best practices and automated tools that can make reproducible research easier.

1 The value of reproducibility

The term "reproducible research" in computational science covers a spectrum of activities [3], distinguished by who is doing the reproducing, and to what degree they have access to, or make use of access to, the materials developed by the original researchers. At one end of the spectrum, an independent repeat of the study by a third party, based purely on the description of the methodology in a published article, with no access to the original code or computing environment (and which hopefully comes up with the same results, or qualitatively similar results depending on how many important details were missing from the article and on how much small differences in numerical methods affect the results). At the other end, the original researcher re-running the original code with the original data (maybe in the same environment, maybe not) and in principle getting identical results. In the middle, having a declarative, machine-readable description of the methodology (for example, using SBML [7] in systems biology), someone other than the original author(s) re-running the code, or writing a new implementation from scratch, perhaps in a different language, but with access to the original source code. All points on the spectrum are valuable (although some have argued differently [5]), fully-independent reproduction being critical to the integrity of the scientific process, while being able to re-run

the original code to get the same results (sometimes called "replication" to distinguish it from independent reproduction [5]) is fundamental to software engineering: reuse of existing code (usually organized into libraries) hugely accelerates the process of developing new functionality and hence doing new science.

2 The difficulty of reproducibility

Although independent reproduction is understandably more difficult to achieve, even "mere" replication of scientific results using the original code is often far from trivial.

Why are so many scientific computational studies not reproducible/replicable in practice? Because, even when they make their code and data available to others, researchers do not record exactly what code was run in order to generate a given result. It is standard scientific procedure, especially in experimental sciences, to write down every critical detail of an experiment in a lab notebook. The problem with computational science is that the number and granularity of critical details is so high that exhaustively identifying them is a major challenge and writing them all down prohibitively time consuming.

A cartoon description of the process of a "typical" computational project, inspired by the author's experiences as a graduate student and postdoc and by many conversations with colleagues, may serve to illustrate the problem: The most important detail to record is the particular version of your source code that was used to obtain a particular result, which can be identified either using a version control system or, since your scientific computing education consisted of a single programming course in FORTRAN and did not include any exposure to software development tools, by making a copy of the program and changing the file name. Later on you tweak the file in some minor way but keep the same name (thinking up good, descriptive file names not being easy). You're certain this tweak will not affect the earlier results but you have no automated tests and this time it does. Later, you discover version control and things improve, but it is not sufficient to pin down the code – programs are also run with input files (containing parameters, configurations, data) and perhaps with command line parameters or with settings in a graphical configuration panel. And so all these small details should also be recorded but are not, either not at all or not consistently: you're tweaking a figure for a paper or abstract with a looming deadline, and so you're interactively, iteratively changing stuff to make it just right and you haven't really got time to note down every last detail. And then, three months later you get the reviews (or two years later you get an e-mail because someone can't reproduce your results) and so you need to re-run some simulations/analyses, but in the meantime you've got a new laptop and/or upgraded your operating system and several of the software libraries on which your code depends have changed, as has perhaps the processor architecture and now your code doesn't compile or run, or maybe (miracle) it does run, but gives subtly or not so subtly different results and we don't know what's changed because we didn't write down exactly which versions of the libraries were being used three months or two years ago, and perhaps we don't even know what libraries were being used (maybe we know the top-level ones, but not the libraries on which the top-level libraries depend).

To sum up, the difficulty in reproducing computational research is in large part due to the difficulty in capturing every last detail of the software and computing environment, which is what is needed to achieve reliable replication.

3 Requirements for reproducibility

The remedy to this problem contains two parts, both of which are needed, although the second is arguably the most important: (i) reduce the sensitivity of the results to the precise details of the code and environment; (ii) automate the process of capturing the code and environment in every detail.

The first part of the remedy, increasing the robustness of the scientific code, is an important one to pursue for other reasons than just reproducibility: a reliable scientific result should not depend on small details of a given implementation. Strategies for more robust code are widely employed in professional software development and have been described in many places [9]; they include reducing the tightness of the coupling between different parts of the code through a modular design and well-defined interfaces, building on established, widely-used and well-tested libraries, writing test suites. At the same time, making code more robust has costs in time and manpower, which may not be worthwhile incurring for scientific code with a limited number of users.

The second part, automatically capturing the details of the code and environment, may be approached in two ways: (i) taking a digital copy of the entire environment using a system virtual machine/hardware virtualization approach; (ii) capturing and storing metadata about the code and environment that allows it to be recreated later.

These approaches are complementary. Taking a snapshot of the entire environment [1, 12] has the advantages of simplicity (both in taking the snapshot and in later making use of it to replicate the results) and of guaranteed comprehensiveness. Capturing and storing metadata about the code and environment is more difficult, and reproducing the experiment later more involved, but this approach is (i) more lightweight, with less data to be stored, and hence can be used for every single exploratory computation, where the VM snapshot approach may be reserved only for those computations that produce final figures, (ii) makes it easier to change details of the code and environment that ought not to affect the result (whether for verification, extension or reuse of the code) and checking that the output is the same, (iii) is potentially more future-proof, being textual rather than binary data, (iv) makes it much simpler to search, index and compare previous computations. The ideal approach would be to combine both approaches, using the metadata for those cases where it is more accessible and maintaining the VM image for ease of exact replication and as a fall-back if not all metadata can be or has been captured. Lightweight equivalents to taking a full VM snapshot [6] would also allow using the dual approach for every exploratory computation.

The metadata that needs to be captured to ensure replication of a computational result (or at least to assist in tracking down the reason in the event replication cannot be achieved) includes at least the following: (i) the hardware platform on which a computation was performed, insofar as the details of the platform can affect the numerical outcome of the computation: for a distributed or otherwise parallelized computation, the number of nodes and how the computation was divided between them; for each node, the processor architecture, word size, byte order; (ii) the operating system: identity, version; (iii) for compiled code, the source code (or a stable URL where the specific version of the code may be obtained), the source code of any libraries used, the compilation procedure including details of compiler options, compilation order, etc.; (iv) for code run on a process virtual machine/run-time system such as the Java Virtual Machine, the precise version of the VM and ideally its source code and details of how it was compiled, (iv) for interpreted code, the code itself, the source code of any libraries/dependencies, the interpreter identity, version, how it was compiled, how any compiled dependencies (e.g. C extensions for Python) were built; (v) the identity of input data (including parameter/configuration files), which may include a stable URL where the file may be obtained or database accessed. Given the difficulty/impossibility of ascertaining that the content of source code or data files at a

given URL or in a given database has not been changed, it would be prudent to calculate a cryptographic hash/digest of the file/data contents (some version control systems such as Git already include such checks); (vi) the output data (including graphs, figures, etc., as well as any log files), carefully linked to the computation that produced them and again verified with a cryptographic hash (see [8] for the use of this approach in provenance tracking). For use in project management and in searching/indexing computations, it is very useful to understand the scientific context in which the calculation was performed: for what reason, as part of which project, what was the hypothesis the original experimenter had in mind when launching the computation?

4 Achieving reproducibility in day-to-day research

The majority of articles about, and tools to assist with, reproducible research focus on reproducibility of *published* research, e.g. [10, 12]. This is of course the most important use-case, but there is also a huge amount of value in making day-to-day exploratory research (some of which will later end up in publications of course) more reproducible. First, ease of reproducibility implies ease of reuse, with all the benefits that brings for efficiency, speed of development, finding bugs, etc. Second, having all the metadata about previous simulations/analyses easily accessible makes it much easier to do things like: identify when a time-consuming computation or part of a computation has been performed before *under identical conditions* and hence the result can be read from a file or database rather than recalculated; perform comparisons across different conditions/different code versions/different parametrizations; check that a result is robust (check that the results are the same when run on a different platform, with an updated OS, etc.). Third, experience with making day-to-day research reproducible will make it much more likely that published research will really be reproducible, rather than having reproducibility be just one more chore associated with submitting a manuscript, when any available shortcuts will be taken. One software tool that does focus on reproducibility of day-to-day, exploratory research is VisTrails (`http://www.vistrails.org`), which takes a visual, pipeline-based approach. The principal difference between VisTrails and the approach outlined below is that VisTrails provides a specific framework for performing computations, within which the context of computational experiments can be recorded and tracked, rather than aiming to capture this context with minimal changes to existing code or workflows. Which approach is most appropriate for a given project will depend on many factors, in particular on the amount of pre-existing code and applications and on the relative preference of the researcher for a graphical or scripting interface.

5 Best practices to simplify reproducibility

Before discussing tools to automate the capture of experimental context, it is worthwhile trying to identify best practices in running a computational science project that can make the work of such tools easier.

The first best practice is a consistent, repeatable computing environment. If moving a computation to a new system, it should be simple and straightforward to set up the environment identically (or nearly so) to that on the original machine(s). This suggests either using a package management system (for example apt, yum, MacPorts) and/or a configuration management tool (e.g. Puppet, Chef, Fabric). The former provide pre-built, well-tested software packages in central repositories, thus avoiding the vagaries of downloading and compiling packages from diverse sources and being faster to deploy. The latter enable the definition of recipes for machine setup, which is particularly important when using software that is not available in a package management system, whether because it has not been packaged (e.g. because it is not widely

used or is developed locally for internal use) or because the version in the package manager is too outdated. These tools are particularly useful when renting machine time. Clearly, if you have a recipe that specifies exactly which versions of which software are installed on a system, then recording this information is potentially as simple as recording the location and version of the recipe.

Second is use of version control. This should perhaps go without saying, but many, perhaps a majority of researchers using computational methods have had no training in software development and are unaware of the tools that are available, using instead the unreliable and inefficient "copy-and-rename" method of tracking which version of the code was used for a particular simulation. All files involved in the scientific process from conception to publication should be under version control. For this reason, modern distributed version control systems (Mercurial, Git, Bazaar, etc.) are advantageous because of the simplicity of setting up a new repository and putting files under version control, and of being able to commit changes without network access.

Third, it is good practice to cleanly separate code from configuration/parameters, for the following reasons: (i) the latter are changed more frequently than the former, so different recording tools are most appropriate for each, e.g. VCS for code, database for parameters; (ii) parameters are directly modified/controlled by the end user, code may not be; this means that parameters can be controlled through different interfaces (configuration files, graphical interfaces); (iii) it ensures that changes are made in a single place, rather than spread throughout a code base; (iv) parameters are useful for searching/filtering/comparing computations made with the same model or analysis method but with different parameters; storing the parameters separately makes this easier.

6 Automating the capture of experimental context

The major constraint in the design of a system for automated metadata/context capture is the diversity of tools, interfaces and workflows used: different researchers have very different ways of working: with the command line, with graphical interfaces, with batch jobs, or any combination of these for different components (simulation, analysis, graphing, etc.) and phases of a project; some projects are essentially solo endeavors, others collaborative projects, possibly distributed geographically. Clearly, a one-size-fits-all tool is not possible. At the same time, there is a large overlap in what needs to be recorded, and it would be wasteful for core, shared functionality to be reimplemented independently for each possible workflow. This suggests a design with a core library or libraries providing core functionality, and then a series of interfaces covering the different possible ways of working.

The core library should implement the following functionality: capture of the hardware environment (by querying the operating system(s)); capture of the general software environment (operating system identity and version); capture of the specific software environment (source code of main programs and libraries/dependencies, based on one or more of configuration management recipes, package-management systems, interaction with VCS, parsing/introspection of code to determine dependencies); capture of the inputs to and outputs from a computation (path/URL/other identifier of input and output data, cryptographic hash of input/output data, command-line arguments, parameter/configuration settings); capture of the scientific context (it is only this last part that cannot be automated, requiring input from the scientist to explain their motivations).

Interfaces may address different phases of the computation lifecycle – configuring, launching, reviewing outcomes – and couple the core library to the interface usually used for each phase (command-line, graphical interface, etc.)

Importantly, it is not necessary to capture *all* this information for such a system to be useful. The

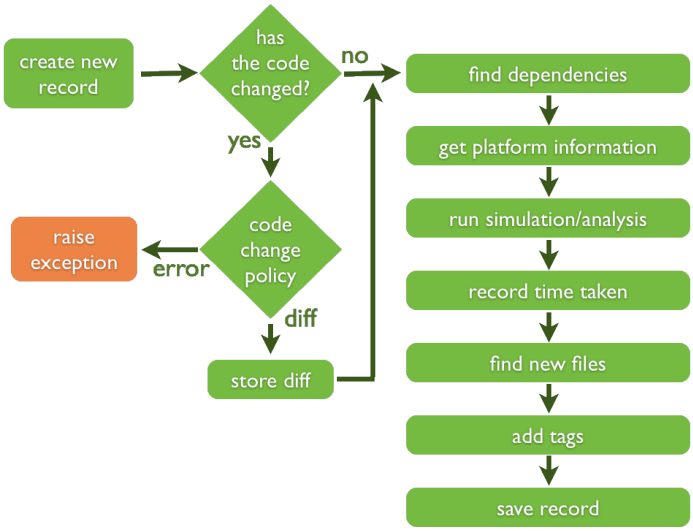

Figure 1: Steps in capturing experiment context when launching a computation with Sumatra. Figure licensed under the Creative Commons Attribution 3.0 Unported license (http://creativecommons.org/licenses/by/3.0/)

more that is captured, the easier future reproducibility will be, but it is not all-or-nothing; any information is better than none, and we can focus initial work on the most critical information (the scientists' own source code) and then improve the system incrementally. Critically, current ease of use should not be sacrificed to future ease of reproducibility, where these are in conflict: as noted above, these tools must be very easy to use and not appreciably slow the scientist's usual workflow (or even accelerate it), otherwise they will not be used consistently and will be put aside when faced with a tight deadline.

I have developed an implementation of a core library as described above in Python, together with two different interfaces: a command line tool which wraps the normal launching of a computation with phases of capturing a computation's context, inputs and outputs (illustrated in Figure 1) and also provides a simple facility for viewing the records of previous computations in text format on the console; and a web-browser-based interface with a built-in local webserver for viewing/searching/annotating the computation records. This software, Sumatra, is available from `http://neuralensemble.org/sumatra`.

I will now present an example of using the Sumatra command line interface, in a scenario of attempting to re-run some simulations that were originally performed several years ago. The code was taken from the SenseLab ModelDB database at `http://senselab.med.yale.edu/modeldb/ShowModel.asp?model=83319`, and consists of the implementation of a number of benchmark neuronal network models in several different simulators, associated with a review article on spiking neuronal network simulators [2]. Sumatra requires that the top-level code is under revision control, and so I imported the code from ModelDB into a Mercurial repository (available at `https://bitbucket.org/apdavison/spiking-network-benchmarks/`), and then cloned this repository into a local directory.

In this directory, we create a Sumatra project using:

```
$ smt init --datapath=. --archive=./archive SpikingNetworkBenchmarks
```

The 'datapath' option is where I expect new data files to be generated (i.e. in or below the current working directory), while the 'archive' option ensures that repeated simulation runs do not overwrite existing data, by copying the output files to a time-stamped archive.

Now we run some simulations, starting with some Python code written for the Brian simulator (`http://www.briansimulator.org/`). Normally, we would run the simulation using:

```
$ python COBAHH.py
```

To run the simulation and also capture the experimental context, we use:

```
$ smt run -e python -m COBAHH.py
```

This displays a figure on the screen, but we would prefer to save the figure to a file, so we edit the code and run again. Assuming that we will use the same Python executable and the same script several times, we first set them as defaults:

```
$ smt configure -e python -m COBAHH.py
$ smt run
Code has changed. Please commit your changes.
```

The error arises because we have edited our script without committing the changes in version control, and by default Sumatra will refuse to run in this circumstance, so that we always know exactly which version of the code was used (Sumatra currently supports Subversion, Git, Mercurial and Bazaar). So now we commit the change, and run again:

```
$ smt run --reason="Run Benchmark 3 with Brian" --tag="Brian-1.0"
```

This time it finished successfully, and tells us what data files were generated by the script. We also took the opportunity to annotate our simulation record with some metadata – the reason why we ran the simulation and a tag – to help finding this record again in future. We can also add a qualitative assessment of the outcome of the simulation:

```
$ smt comment "Creates a colourful figure"
```

The original code dates from 2008, just after Brian 1.0.0 was released, and so this was the version we installed. In building on this model in future however, we would prefer to use a more up-to-date version, so let's install Brian 1.3.1 and check that we still get the same results. To do this, we need to know the label of the last simulation (since we didn't specify one, Sumatra generated one for us, based on the time stamp).

```
$ smt list
20120219-153526
20120219-153809
```

Now we can re-run it:

```
$ smt repeat 20120219-153809
The new record does not match the original. It differs as follows.
Record 1                : 20120219-153809
Record 2                : 20120219-153809_repeat
Executable differs      : no
Code differs            : yes
  Repository differs    : no
  Main file differs     : no
  Version differs       : no
  Non checked-in code   : no
  Dependencies differ   : yes
Launch mode differs     : no
Input data differ       : no
Script arguments differ : no
Parameters differ       : no
Data differ             : yes
run smt diff --long 20120219-153809 20120219-153809_repeat to see the
    differences in detail.
```

Label	Reason	Outcome	Duration	Processes	Executable		Script		
					Name	Version	Repository	Main file	Vers
20120219-155152	Run Benchmark 3 on four nodes	Success	24.38s	4	NEURON	7.1	/home/andrew/dev/spiking-network-benchmarks	init.hoc	690fce
20120219-154940	Run Benchmark 3 on four nodes	FAILED: delay needs to be non-zero for distributed simulations	2.72s	4	NEURON	7.1	/home/andrew/dev/spiking-network-benchmarks	init.hoc	690fce
20120219-154600	Check that Benchmark 3 runs on NEURON		53.84s	1	NEURON	7.1	/home/andrew/dev/spiking-network-benchmarks	init.hoc	690fce
20120219-153809_repeat	Repeat experiment 20120219-153809	Quantitative changes to output figures.	51.11s	1	Python	2.7.1	/home/andrew/dev/spiking-network-benchmarks	COBAHH.py	690fce
20120219-153809	Run Benchmark 3 with Brian	Creates a colourful figure	51.90s	1	Python	2.7.1	/home/andrew/dev/spiking-network-benchmarks	COBAHH.py	690fce
20120219-153526		Figure displayed on screen	55.62s	1	Python	2.7.1	/home/andrew/dev/spiking-network-benchmarks	COBAHH.py	63c4ec

Figure 2: Sumatra web interface showing summary information about all simulations.

Sumatra re-runs the original simulation (which, if we had made any changes, would include checking out the correct version of our code from version control) and then compares the context of the two experiments. Here we can see that the version number of one of the dependencies has changed, and so have the output data files. A visual inspection of the figures shows qualitatively similar, but quantitatively different results. Here we know that we changed only one module, but in a more realistic scenario the information given here about what exactly is different would be potentially very useful in tracking down the possible reasons for the changes.

To get more information about our simulations, we could use the "`smt list --long`" command, but the terminal is a limited environment for displaying information, so we will switch to the web interface:

```
$ smtweb
```

This starts a simple webserver and opens a web-browser tab showing a list of the simulations we have performed, see Figure 2. Here I should note that Sumatra supports both local and remote storage of computation records. The default is local storage using an SQLite database accessed via the Django web framework (on which the "smtweb" interface is built); there is also the option to store records on a remote server using the Sumatra web API (the API definition and a server implementation are available from `https://bitbucket.org/apdavison/sumatra_server/`). Both local and remote stores allow multiple projects and multiple users.

The record list shows a summary of metadata about the experiment context. To see more details, we can navigate to the detail view, see Figure 3. This view shows all information that has been captured about the simulation run, including the location and version of the Python interpreter, the version number of the user's code, the path to the output data files (including an SHA-1 hash of the file contents), a list of the Python modules that were imported by the main script, together with version numbers where these could be obtained, information about the platform on which the simulation was run, and a capture of the terminal output.

While most of the metadata shown here can be captured for any command-line driven computation, the list of dependencies requires that Sumatra has some knowledge of the programming language(s) used. Sumatra can currently find dependencies for Python scripts and for scripts written for two popular simulators in computational neuroscience, NEURON and GENESIS. Dependency finders for Matlab, Octave and R scripts, and for C, C++ and Java source code, are planned. Once a list of dependencies has been obtained, Sumatra uses a series of heuristics to obtain version numbers, some language-independent (such as obtaining the version number from a revision control system), some language-dependent (such as searching for variables named "VERSION", "__version__" or "get_version" in Python). This is a good example of graceful degradation in context capture – even if Sumatra cannot obtain all the version numbers or all the dependencies, some information is better than none.

Continuing with our scenario, we turn from the Brian simulator to NEURON (`http://www.`

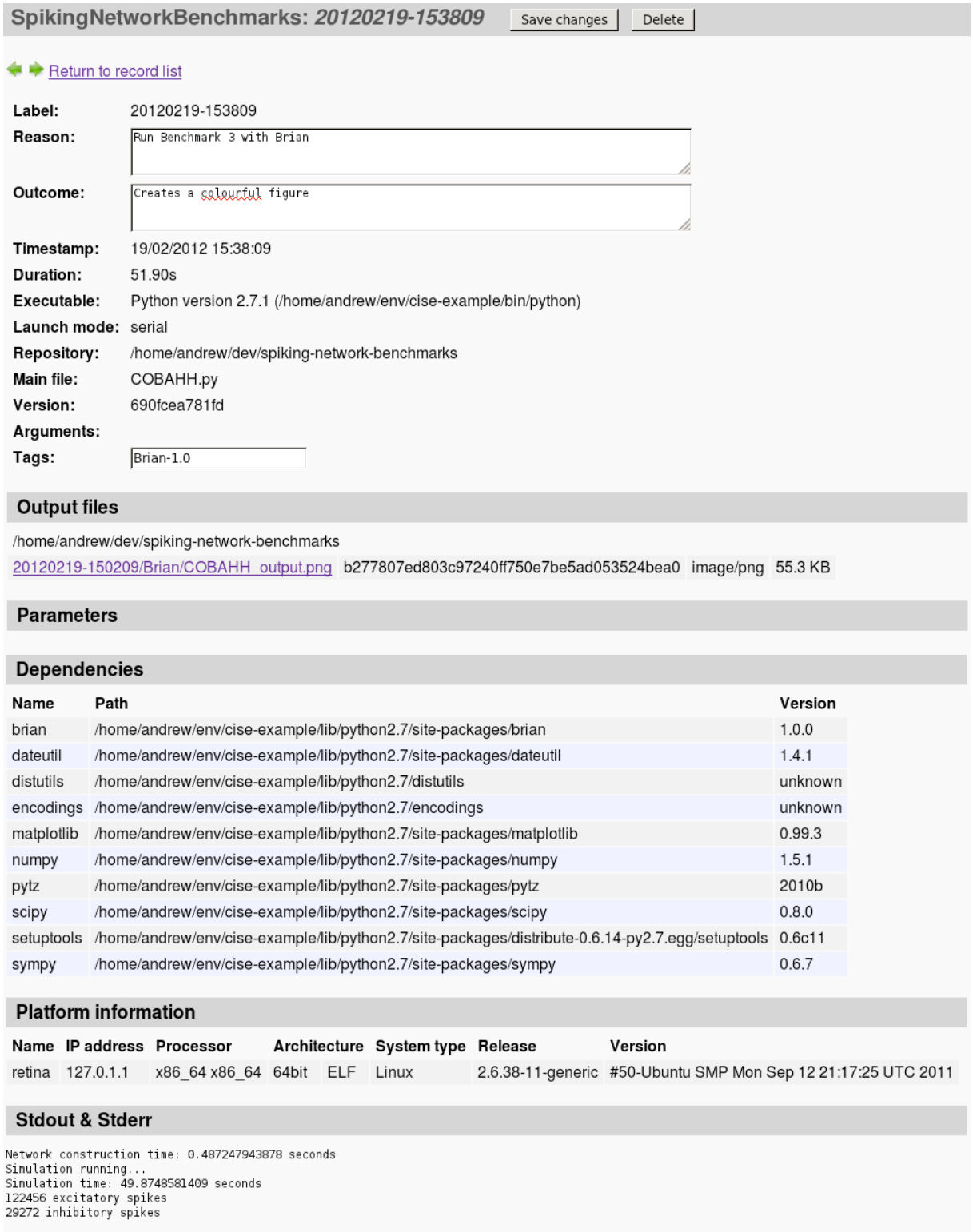

Figure 3: Sumatra web interface showing a detailed view of the metadata associated with a single simulation.

neuron.yale.edu/). We change to the NEURON/cobahh directory, where we would normally run:

```
$ nrniv init.hoc
```

Switching the project defaults, we now run with Sumatra using:

```
$ smt configure -e nrniv -m init.hoc
$ smt run -r "Check that Benchmark 3 runs on NEURON"
```

NEURON supports parallelisation of neuronal network simulations using MPI, so we can also run as:

```
$ mpirun -n 4 nrniv -mpi init.hoc
```

or, with context capture:

```
$ smt run -n 4 -r "Run Benchmark 3 on four nodes"
```

This failed, but is nonetheless captured by Sumatra. Looking at the error messages, we can make a note of why it failed in case we encounter the same error again in the future.

```
$ smt comment "FAILED:  delay needs to be non-zero for distributed simulations"
```

So we have to change the value of the "delay" parameter, and try again. Now it fails because we changed the code and did not commit the changes. But I don't really want to commit such a small change at this stage – I may find other problems later – so I ask Sumatra to store the diff and then continue:

```
$ smt configure --on-changed=store-diff
$ smt run -n 4 -r "Run Benchmark 3 on four nodes"
```

Now it runs successfully. Note that, leaving aside the optional annotation, the command for running the MPI simulation with Sumatra is actually shorter than that for running it stand-alone. This is an example of minimising the impact of reproducibility measures on the day-to-day workflow. In this case, using Sumatra even requires slightly less typing, as some of the 'boilerplate' elements from the command-line are stored as configuration options.

Although I have focused here on interfaces, I would like to emphasize the design of Sumatra as a an easily-extensible library that can be used by others to build their own interfaces (for example, a desktop application built using any language that has Python bindings) or incorporated within existing tools (for example workflow engines) to extend their capabilities. A good example is that for scientific code written in Python, it is possible to use the Sumatra API directly and then run scripts as normal, rather than using the 'smt' launcher.

Sumatra is developed with an open development model: several users of the software have contributed code back to the project, and further contributions are welcomed. Specific planned improvements are dependency-finding support for a greater number of languages and for mixed-language projects, and search capabilities: filtering the database of previous computations based on tags is already possible, but filtering on other metadata, particularly parameter values, would be very useful.

7 Conclusions

In this article I have tried to establish (i) that reproducibility/replicability is both necessary in ensuring the credibility of scientific research (as has been noted elsewhere [4, 11]) and useful in

promoting software reuse and hence accelerating scientific progress; (ii) that it should be part of day-to-day research, not done post-hoc only for published results, and so needs to be made much, much easier to achieve; (iii) that an important part of achieving it is to meticulously record every factor that could conceivably have an impact on our results, (iv) that to perform this meticulous record-keeping manually would be tedious beyond endurance, not to say impossible in practice due to time constraints and the limits of the human attention span. The upshot of all this is that we need automated tools for the record keeping, since after all automating mundane, repetitive, tedious tasks is why digital computers were invented in the first place. I have enumerated the factors that need to be recorded by such tools, presented some considerations in their design, and briefly described one implementation of such a system. Since automated tools work best in a consistent and somewhat predictable environment I have attempted to identify some best practices that will make the life of both the researcher and the tool easier.

Finally, I would like to reemphasize that if reproducibility is not made easier to achieve, only the very conscientious will achieve it. While having journals and funding agencies require demonstration of reproducibility will certainly increase its level in the published computational science literature, contriving for reproducibility to become of net benefit in day-to-day research is likely to have a more profound effect.

References

[1] G.R. Brammer, R.W. Crosby, S. Matthews, and T.L. Williams. Paper Mâché: Creating dynamic reproducible science. *Procedia CS*, 4:658–667, 2011.

[2] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James Bower, Markus Diesmann, Abigail Morrison, Philip Goodman, Frederick Harris, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of Computational Neuroscience*, 23:349–398, 2007. 10.1007/s10827-007-0038-6.

[3] S. Crook, H.E. Plesser, and A.P. Davison. Lessons from the past: approaches for reproducibility in computational neuroscience. In J.M. Bower, editor, *20 Years of Computational Neuroscience*. Springer-Verlag, in press.

[4] D.L. Donoho, A. Maleki, I.U. Rahman, M. Shahram, and V. Stodden. 15 years of reproducible research in computational harmonic analysis. *Computing in Science and Engineering*, 11:8–18, 2009.

[5] C. Drummond. Replicability is not reproducibility: nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the* 26^{th} *ICML*, Montreal, CA, 2009.

[6] P.J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, June 2011.

[7] M. Hucka, A. Finney, H.M. Sauro, H. Bolouri, J.C. Doyle, H. Kitano, and A.P. Arkin. The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models. *Bioinformatics*, 19:524–531, 2003.

[8] David Koop, Emanuele Santos, Bela Bauer, Matthias Troyer, Juliana Freire, and Cláudio Silva. Bridging workflow and data provenance using strong links. In Michael Gertz and

Bertram Ludäscher, editors, *Scientific and Statistical Database Management*, volume 6187 of *Lecture Notes in Computer Science*, pages 397–415. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13818-8_28.

[9] S. McConnell. *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004.

[10] P. Nowakowski, E. Ciepiela, D. Harezlak, J. Kocot, M. Kasztelnik, T. Bartynski, J. Meizner, G. Dyk, and M. Malawski. The Collage authoring environment. *Procedia CS*, 4:608–617, 2011.

[11] V. Stodden. Trust your science? Open your data and code. *Amstat News*, July 1st 2011.

[12] P. Van Gorp and S. Mazanek. SHARE: a web portal for creating and sharing executable research papers. *Procedia CS*, 4:589–597, 2011.