

# 1

---

## *Sumatra: a toolkit for reproducible research*

---

**Andrew P. Davison**

*UNIC, CNRS UPR 3293, Gif sur Yvette, France*

**Michele Mattioni**

*European Bioinformatics Institute, Hinxton, UK*

**Dmitry Samarkanov**

*L2EP Optimization Team, Ecole Centrale de Lille, Villeneuve d'Ascq, France*

**Bartosz Teleńczuk**

*UNIC, CNRS UPR 3293, Gif sur Yvette, France*

*Institute for Theoretical Biology, Humboldt University, Berlin*

## CONTENTS

1.1	Introduction .....	1
1.2	Using Sumatra .....	2
1.3	Design criteria .....	8
1.4	Architecture .....	9
1.4.1	Code versioning and dependency tracking .....	10
1.4.2	Data handling .....	11
1.4.3	Storing provenance information .....	12
1.4.4	Parameter handling .....	12
1.4.5	Launching computations .....	13
1.4.6	Putting it all together .....	13
1.4.7	Search/query/reuse .....	14
1.5	Discussion .....	15
	Acknowledgments .....	16
	Short biographies .....	16

---

## 1.1 Introduction

Lack of replicability in computational studies is, at base, a problem of shortcomings in record keeping. In laboratory-based experimental science, the tradition is to write down all experimental details in a paper notebook. This approach is no longer viable for many computational studies, as the number of details that could have an impact on the final result is so large. Automated or semi-automated tools for keeping track of all the experimental details – the scientist's own code, input and output data, supporting software, the computer hardware used, etc. – are therefore needed.

For the busy scientist, the time investment needed to learn to use these tools, or to

adapt their workflow so as to make use of them, may be one they are reluctant to make, especially since the problems of lack of reproducibility often take some time to manifest themselves. To achieve wide uptake among computational scientists, therefore, tools to support reproducible research should aim to minimise the effort required to learn, adopt and use them (see [1] for a more detailed version of this argument).

Sumatra is a software tool to support reproducible computational research, which aims to make reproducible computational science as easy to achieve (or easier) than non-reproducible research, largely by automating the process of capturing all the experimental details. In practice, this means that using Sumatra should require minimal changes to existing workflows and, given the wide diversity in workflows for computational science, Sumatra should be easy to adapt to different computational environments.

This chapter is intended for two groups of people:

- scientists who are interested in using Sumatra to track the details of their own research;
- developers who are interested in using Sumatra as a library in their own software for reproducible research.

The first section is an extended case study, illustrating how Sumatra may be of use in day-to-day research. This is followed by an in-depth explanation of Sumatra's architecture, including examples of how to use Sumatra as a Python library and how to extend and customize Sumatra.

---

## 1.2 Using Sumatra

We will illustrate one way to use Sumatra, and why you might want to use Sumatra, with a story about Alice and Bob. Bob is a graduate student in Alice's lab. When Alice was a graduate student herself, she kept track of the evolution of her code by giving each significant version a different file name, and she included the file name as a label in every figure she generated. Alice used to be quite confident she could, if it were ever necessary, go back and recreate the results from her earlier papers, since she has the original data carefully archived on CD-ROMs. However, after her recent experience with Charlie, she is not so sure. Charlie was a postdoc in Alice's lab, who got some great results, which they wrote up and submitted to a high-profile journal. The reviews were quite positive, but the reviewers asked for some new figures and a change to one of the existing figures. The problem was that when they tried to generate the modified figure, they could not get the results to match: the new graph looked significantly different, and no longer showed the effect they had found. Although Charlie had used the Subversion version control system for his code, he had not been so careful about keeping track of which version of the code had been used for each figure in the manuscript: several of the figures had originally been generated for a poster, and in the rush to get the poster finished in time to send to the printers, Charlie had not had time to keep such careful notes as usual, and had not always remembered to check-in changes in his code to the Subversion repository. Now Charlie has left science for a job with a major bank, and the manuscript is languishing in a drawer.

As a consequence of these experiences, Alice asked Bob, her new graduate student, to try out Sumatra. Sumatra automates the necessary, but tedious and error-prone process of keeping track of which code version was used to produce which output. Bob has his code in a Mercurial version control repository (for the purposes of this chapter, we will use a simplified version of Bob's code. If you would like to follow along, the repository is available

at <http://bitbucket.org/apdavison/ircr2013>). Bob downloaded and installed Sumatra according to the instructions at <http://neuralensemble.org/sumatra>.

Bob normally runs his analysis (of scanning electron microscope images of glass samples) as follows:

```
$ python glass_sem_analysis.py MV_HFV_012.jpg
1699.875 65.0
```

This analyses the image specified on the command line, generates some further images, and prints out some statistics (see the SciPy tutorial at <http://scipy-lectures.github.com/> for more details). The output images are saved to a specific subdirectory labelled according to the day on which the code is run, and the individual files are labelled with a timestamp, e.g. “Data/20121025/MV\_HFV\_012\_163953\_phases.png”.

He creates a new Sumatra project in the same directory, using the `smt` command-line tool:

```
$ smt init ProjectGlass
$ smt configure -e python -m glass_sem_analysis.py -i . -d Data
```

This creates a new project, and sets “python” as the default executable to be used, “glass\_sem\_analysis.py” as the default script file, the current directory (“.”) as the place to look for input data, and a subdirectory “Data” as the place to start looking for output files. (If Bob could not remember the various options to the “smt configure” command, “smt help configure” would tell him).

“smt info” shows the current configuration of Bob’s project. Note that it is using the already-existing Mercurial repository in his working directory:

```
$ smt info
Project name      : ProjectGlass
Default executable : Python (version: 2.6.7) at /usr/bin/python
Default repository : MercurialRepository at /home/bob/Projects/Glass
Default main file  : glass_sem_analysis.py
Default launch mode : serial
Data store (output) : ./Data
.                  (input) : .
Record store      : Django record store at
                   /home/bob/Projects/Glass/.smt/records
Code change policy : error
Append label to    : None
```

Now to run the analysis using Sumatra:

```
$ smt run MV_HFV_012.jpg
1699.875 65.0
```

Since Bob has already specified the executable and script file, all he has to provide is the name of the input data file. The program runs as before and gives the same results, but in addition, Sumatra has captured a great deal of information about the *context* of the computation – exactly which version of the code was used, what the input and output data files were, what operating system and processor architecture were used, etc. Some of this information can be viewed in the console:

```
$ smt list -l
Label      : 20121025-170718
Timestamp  : 2012-10-25 17:07:18
Reason     :
Outcome    :
Duration   : 3.73256802559
Repository : MercurialRepository at /home/bob/Projects/Glass
Main_File  : glass_sem_analysis.py
Version    : 9d24b099b5f3
```

```

Script_Arguments : MV_HFV_012.jpg
Executable       : Python (version: 2.6.5) at /usr/bin/python
Parameters       :
Input_Data       : MV_HFV_012.jpg(5d789282b10a0da7a91560f33f8baf7272f7543d)
Launch_Mode      : serial
Output_Data      : 20121025/MV_HFV_012_170722_phases.png(c9955f84ca3c1912...
                  : 20121025/MV_HFV_012_170722_sand.png(20bd5420d37ee589f3...
                  : 20121025/MV_HFV_012_170722_histogram.png(e7884dc5f3e9c...
Tags             :

```

but in general it is better to use the built-in web browser-based interface, launched with the `smtweb` command – see Figure 1.1.

Two things in particular should be noted from this figure. The first is that the versions of not only the Python interpreter and Bob’s own code, but also the libraries on which Bob’s code depends (NumPy, etc.), are captured. The second is that the path of each input and output data file is accompanied by a long hexadecimal string. This is the SHA1 digest, or hash, of the file contents (as used in cryptographic applications, and also in version control systems such as Git and Mercurial). If the file contents are changed even slightly, the hash will change, which allows us to check for files being corrupted or accidentally over-written.

Now Bob would like to investigate how his image analysis method is affected by changing its parameters. He thinks this will be easier to keep track of if the parameters are separated out into a separate file, so he modifies his script and adds a new file `default_parameters`. The script now expects two arguments, first the parameter file, second the input data, and would normally be run using

```
$ python glass_sem_analysis.py default_parameters MV_HFV_012.jpg
```

but Bob wants to run it with Sumatra:

```
$ smt run default_parameters MV_HFV_012.jpg
Code has changed, please commit your changes.
```

Bob has forgotten to commit his changes to the version control repository. Sumatra detects this, and will then either refuse to run (the default, seen here) or will store the differences since the last commit. Bob commits and tries again.

```
$ hg commit -m 'Separated out parameters into separate file'
$ smt run -r 'test separate parameter file' default_parameters MV_HFV_012.jpg
1699.875 65.0
```

Note that he has also used the “-r” flag to note the reason for running this analysis, in case he forgets in future. Have Bob’s modifications had any effect on his results? The output statistics are the same, and an inspection of the output data hashes in the web interface shows they have not changed either, so no, the results are unchanged.

We have seen already that Bob has less typing to do when running his analyses with Sumatra, as he has already specified the executable and script file as defaults. This is an example of how Sumatra tries to make it easier to use a tool for reproducible research than not to use one. Another example is the ability to specify parameters on the command line, rather than having to edit the parameter file each time:

```
$ smt run -r 'No filtering' default_parameters MV_HFV_012.jpg filter_size=1
$ smt run -r 'Trying a different colourmap' default_parameters
                                     MV_HFV_012.jpg phases_colourmap=hot
$ smt comment 'The default colourmap is nicer'
```

So far, Bob has been using Charlie’s old computer, running Ubuntu Linux 10.04. The next day, he is excited to find that the new computer Alice ordered for him has arrived. He installs Ubuntu 12.04, together with all the latest versions of the Python scientific libraries. He also copies over his glass analysis data, and migrates the Sumatra project. He tries

ProjectGlass
20121025-170718

### General info

**Label:** 20121025-170718  
**Reason:**   
**Outcome:**   
**Timestamp:** 25/10/2012 17:07:18  
**Duration:** 3.73s  
**Executable:** Python version 2.6.5 (/usr/bin/python)  
**Launch mode:** serial  
**Repository:** /home/bob/Projects/Glass  
**Main file:** glass\_sem\_analysis.py  
**Version:** 9d24b099b5f3  
**Arguments:** MV\_HFV\_012.jpg  
**Tags:**

### Input files

/

[MV\\_HFV\\_012.jpg](#) 5d789282b10a0da7a91560f33f8baf7272f7543d 0 bytes

### Output files

./Data

[20121025/MV\\_HFV\\_012\\_170722\\_phases.png](#) c9955f84ca3c19123d24ccc4c87d197514d9e01e image/png 82.2 KB  
[20121025/MV\\_HFV\\_012\\_170722\\_sand.png](#) 20bd5420d37ee589f3c2542a12438cb78663974b image/png 34.3 KB  
[20121025/MV\\_HFV\\_012\\_170722\\_histogram.png](#) e7884dc5f3e9ce2c0d713cb3e9ddffb07bf2010c image/png 27.7 KB

### Parameters

no parameters

### Dependencies

Name	Path	Version
dateutil	/usr/lib/pymodules/python2.6/dateutil	1.4.1
glib	/usr/lib/pymodules/python2.6/gtk-2.0/glib	unknown
gobject	/usr/lib/pymodules/python2.6/gtk-2.0/gobject	unknown
matplotlib	/usr/lib/pymodules/python2.6/matplotlib	0.99.1.1
mpl_toolkits	/usr/lib/pymodules/python2.6/mpl_toolkits	unknown
numpy	/usr/lib/python2.6/dist-packages/numpy	1.3.0
pytz	/usr/lib/python2.6/dist-packages/pytz	2010b
scipy	/usr/lib/python2.6/dist-packages/scipy	0.7.0
wx	/usr/lib/python2.6/dist-packages/wx-2.8-gtk2-unicode/wx	2.8.10.1 (gtk2-unicode)

### Platform information

Name	IP address	Processor	Architecture	System type	Release	Version
Ubuntu01	127.0.1.1	i686	32bit ELF	Linux	2.6.32-24-generic	#39-Ubuntu SMP Wed Jul 28 06:07:29 UTC 2010

### Stdout & Stderr

```

/usr/lib/python2.6/dist-packages/numpy/lib/function_base.py:272: DeprecationWarning:
    The histogram semantics being used is now deprecated and
    will disappear in NumPy 1.4. Please update your code to
    use the default semantics.

""", DeprecationWarning)
1699.875 65.0
  
```

**FIGURE 1.1**

Record of a computation captured with Sumatra, displayed in the web browser interface.

to run the analysis script, but gets an error: in the latest version of NumPy, the return format of the `histogram()` function has changed. This is straightforward to fix (see <https://bitbucket.org/apdavison/ircr2013/changeset/924a39a>), so now Bob can commit and try again:

```
$ smt run -r 'Fixed to work with new histogram() function'
                                default_parameters MV_HFV_012.jpg
```

Has the upgrade affected Bob's results?

```
$ smt diff 20121025-172833 20121026-174545
Record 1           : 20121025-172833
Record 2           : 20121026-174545
Executable differs : no
Code differs       : yes
  Repository differs : no
  Main file differs  : no
  Version differs    : yes
  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
```

OK, Bob knew he had changed the code because of the new `histogram()` function, and he knew the dependencies had changed, because of the operating system upgrade, but it was a bit disappointing to see the output data are different. Using the web browser, we can look at the results from the two simulations (one from Ubuntu 10.04, one from Ubuntu 12.04) side by side (Figure 1.2) – visually there is no difference, just a tiny change in the margins, probably due to the upgraded `matplotlib` package.

Alice puts her head round the door to ask how Bob is getting on with Sumatra. So far, Bob is happy. His productive workflow has hardly changed – in fact he has a little bit less to type, since Sumatra stores the names of the default executable and default script for him, and he can modify parameters quickly on the command line rather than having to open up the parameter file in his editor. The web browser interface lets him quickly browse and search through his results (Figure 1.3), and compare different runs side-by-side. And he feels much more confident that he will be able to replicate his results in the future.

Alice tries Sumatra out for herself the following week. Alice wants to use one of Bob's figures in a grant application, but Bob is on vacation, and she wants to make a few small changes to the figure. She copies Bob's Sumatra record store (which by default was created as the file `.smt/records` in a subdirectory of Bob's working directory) to the lab network file server, so that she can access Bob's records and Bob in turn will be able to see her results when he returns, and sets up a new project on her MacBook:

```
$ smt init -s /Volumes/shared/glass/smt_records ProjectGlass
$ smt configure -e python -m glass_sem_analysis.py -i . -d Data
```

Before starting her own modifications, she re-runs Bob's last analysis:

```
$ smt repeat 20121026-174545
The new record does not match the original. It differs as follows.
Record 1           : 20121026-174545
Record 2           : 20121026-174545_repeat
Executable differs : no
Code differs       : yes
  Repository differs : no
  Main file differs  : no
  Version differs    : no
  Non checked-in code : no
```

## Comparison of records

Click the records you would like to compare:

20121026-174545 20121025-172833

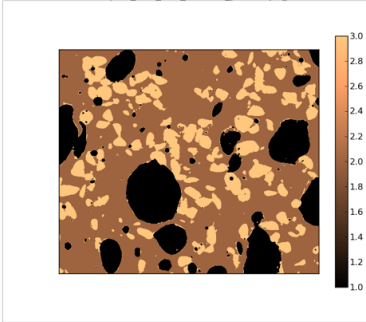
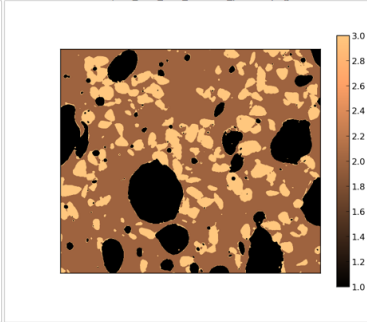
	20121025-172833	20121026-174545																																																																														
Reason:	Parameters are now in a separate file	Fixed to work with the new numpy.histogram() function																																																																														
Outcome:																																																																																
Timestamp:	25/10/2012 17:28:33	26/10/2012 17:45:45																																																																														
Duration:	3.85s	3.85s																																																																														
Executable:	Python version 2.6.5 (/usr/bin/python)	Python version 2.7.3 (/usr/bin/python)																																																																														
Launch mode:	serial	serial																																																																														
Repository:	/home/bob/Projects/Glass	/home/bob/Projects/Glass																																																																														
Main file:	glass_sem_analysis.py	glass_sem_analysis.py																																																																														
Version:	432ff7ef3f45	924a39a0d24c																																																																														
	File name: 20121025/MV_HFV_012_172836_phases.png	File name: 20121026/MV_HFV_012_174557_phases.png																																																																														
																																																																																
	Digest: c9955f84ca3c19123d24ccc4c87d197514d9e01e	Digest: 7f8ed0c6ef97b8317af8e2d9ab9f856a193c2687																																																																														
Dependencies:	<table><tr><th>Name</th><th>Path</th><th>Version</th></tr><tr><td>dateutil</td><td>/usr/lib/pymodules/python2.6/dateutil</td><td>1.4.1</td></tr><tr><td>glib</td><td>/usr/lib/pymodules/python2.6/gtk-2.0/glib</td><td>unknown</td></tr><tr><td>gobject</td><td>/usr/lib/pymodules/python2.6/gtk-2.0/gobject</td><td>unknown</td></tr><tr><td>matplotlib</td><td>/usr/lib/pymodules/python2.6/matplotlib</td><td>0.99.1.1</td></tr><tr><td>mpl_toolkits</td><td>/usr/lib/pymodules/python2.6/mpl_toolkits</td><td>unknown</td></tr><tr><td>numpy</td><td>/usr/lib/python2.6/dist-packages/numpy</td><td>1.3.0</td></tr><tr><td>pytz</td><td>/usr/lib/python2.6/dist-packages/pytz</td><td>2010b</td></tr><tr><td>scipy</td><td>/usr/lib/python2.6/dist-packages/scipy</td><td>0.7.0</td></tr><tr><td>wx</td><td>/usr/lib/python2.6/dist-packages/wx-2.8-gtk2-unicode/wx</td><td>2.8.10.1 (gtk2-unicode)</td></tr></table>	Name	Path	Version	dateutil	/usr/lib/pymodules/python2.6/dateutil	1.4.1	glib	/usr/lib/pymodules/python2.6/gtk-2.0/glib	unknown	gobject	/usr/lib/pymodules/python2.6/gtk-2.0/gobject	unknown	matplotlib	/usr/lib/pymodules/python2.6/matplotlib	0.99.1.1	mpl_toolkits	/usr/lib/pymodules/python2.6/mpl_toolkits	unknown	numpy	/usr/lib/python2.6/dist-packages/numpy	1.3.0	pytz	/usr/lib/python2.6/dist-packages/pytz	2010b	scipy	/usr/lib/python2.6/dist-packages/scipy	0.7.0	wx	/usr/lib/python2.6/dist-packages/wx-2.8-gtk2-unicode/wx	2.8.10.1 (gtk2-unicode)	<table><tr><th>Name</th><th>Path</th><th>Version</th></tr><tr><td>PIL</td><td>/usr/lib/python2.7/dist-packages/PIL</td><td>unknown</td></tr><tr><td>PyQt4</td><td>/usr/lib/python2.7/dist-packages/PyQt4</td><td>unknown</td></tr><tr><td>apport</td><td>/usr/lib/python2.7/dist-packages/apport</td><td>unknown</td></tr><tr><td>apt</td><td>/usr/lib/python2.7/dist-packages/apt</td><td>unknown</td></tr><tr><td>dateutil</td><td>/usr/lib/python2.7/dist-packages/dateutil</td><td>1.5</td></tr><tr><td>glib</td><td>/usr/lib/python2.7/dist-packages/glib</td><td>unknown</td></tr><tr><td>gobject</td><td>/usr/lib/python2.7/dist-packages/gobject</td><td>unknown</td></tr><tr><td>matplotlib</td><td>/usr/lib/pymodules/python2.7/matplotlib</td><td>1.1.1rc</td></tr><tr><td>mpl_toolkits</td><td>/usr/lib/pymodules/python2.7/mpl_toolkits</td><td>unknown</td></tr><tr><td>nose</td><td>/usr/lib/python2.7/dist-packages/nose</td><td>1.1.2</td></tr><tr><td>numpy</td><td>/usr/lib/python2.7/dist-packages/numpy</td><td>1.6.1</td></tr><tr><td>pytz</td><td>/usr/lib/python2.7/dist-packages/pytz</td><td>2011k</td></tr><tr><td>scipy</td><td>/usr/lib/python2.7/dist-packages/scipy</td><td>0.9.0</td></tr><tr><td>setuptools</td><td>/usr/lib/python2.7/dist-packages/setuptools</td><td>0.6</td></tr><tr><td>wx</td><td>/usr/lib/python2.7/dist-packages/wx-2.8-gtk2-unicode/wx</td><td>2.8.12.1 (gtk2-unicode)</td></tr></table>	Name	Path	Version	PIL	/usr/lib/python2.7/dist-packages/PIL	unknown	PyQt4	/usr/lib/python2.7/dist-packages/PyQt4	unknown	apport	/usr/lib/python2.7/dist-packages/apport	unknown	apt	/usr/lib/python2.7/dist-packages/apt	unknown	dateutil	/usr/lib/python2.7/dist-packages/dateutil	1.5	glib	/usr/lib/python2.7/dist-packages/glib	unknown	gobject	/usr/lib/python2.7/dist-packages/gobject	unknown	matplotlib	/usr/lib/pymodules/python2.7/matplotlib	1.1.1rc	mpl_toolkits	/usr/lib/pymodules/python2.7/mpl_toolkits	unknown	nose	/usr/lib/python2.7/dist-packages/nose	1.1.2	numpy	/usr/lib/python2.7/dist-packages/numpy	1.6.1	pytz	/usr/lib/python2.7/dist-packages/pytz	2011k	scipy	/usr/lib/python2.7/dist-packages/scipy	0.9.0	setuptools	/usr/lib/python2.7/dist-packages/setuptools	0.6	wx	/usr/lib/python2.7/dist-packages/wx-2.8-gtk2-unicode/wx	2.8.12.1 (gtk2-unicode)
Name	Path	Version																																																																														
dateutil	/usr/lib/pymodules/python2.6/dateutil	1.4.1																																																																														
glib	/usr/lib/pymodules/python2.6/gtk-2.0/glib	unknown																																																																														
gobject	/usr/lib/pymodules/python2.6/gtk-2.0/gobject	unknown																																																																														
matplotlib	/usr/lib/pymodules/python2.6/matplotlib	0.99.1.1																																																																														
mpl_toolkits	/usr/lib/pymodules/python2.6/mpl_toolkits	unknown																																																																														
numpy	/usr/lib/python2.6/dist-packages/numpy	1.3.0																																																																														
pytz	/usr/lib/python2.6/dist-packages/pytz	2010b																																																																														
scipy	/usr/lib/python2.6/dist-packages/scipy	0.7.0																																																																														
wx	/usr/lib/python2.6/dist-packages/wx-2.8-gtk2-unicode/wx	2.8.10.1 (gtk2-unicode)																																																																														
Name	Path	Version																																																																														
PIL	/usr/lib/python2.7/dist-packages/PIL	unknown																																																																														
PyQt4	/usr/lib/python2.7/dist-packages/PyQt4	unknown																																																																														
apport	/usr/lib/python2.7/dist-packages/apport	unknown																																																																														
apt	/usr/lib/python2.7/dist-packages/apt	unknown																																																																														
dateutil	/usr/lib/python2.7/dist-packages/dateutil	1.5																																																																														
glib	/usr/lib/python2.7/dist-packages/glib	unknown																																																																														
gobject	/usr/lib/python2.7/dist-packages/gobject	unknown																																																																														
matplotlib	/usr/lib/pymodules/python2.7/matplotlib	1.1.1rc																																																																														
mpl_toolkits	/usr/lib/pymodules/python2.7/mpl_toolkits	unknown																																																																														
nose	/usr/lib/python2.7/dist-packages/nose	1.1.2																																																																														
numpy	/usr/lib/python2.7/dist-packages/numpy	1.6.1																																																																														
pytz	/usr/lib/python2.7/dist-packages/pytz	2011k																																																																														
scipy	/usr/lib/python2.7/dist-packages/scipy	0.9.0																																																																														
setuptools	/usr/lib/python2.7/dist-packages/setuptools	0.6																																																																														
wx	/usr/lib/python2.7/dist-packages/wx-2.8-gtk2-unicode/wx	2.8.12.1 (gtk2-unicode)																																																																														
Platform information:	<ol style="list-style-type: none"><li>Name: Ubuntu01</li><li>IP address: 127.0.1.1</li><li>Processor: i686</li><li>Architecture: 32bit</li><li>System type: Linux</li><li>Release: 2.6.32-24-generic</li></ol>	<ol style="list-style-type: none"><li>Name: Ubuntu01</li><li>IP address: 127.0.0.1</li><li>Processor: i686 i686</li><li>Architecture: 32bit</li><li>System type: Linux</li><li>Release: 3.2.0-32-generic</li></ol>																																																																														

FIGURE 1.2

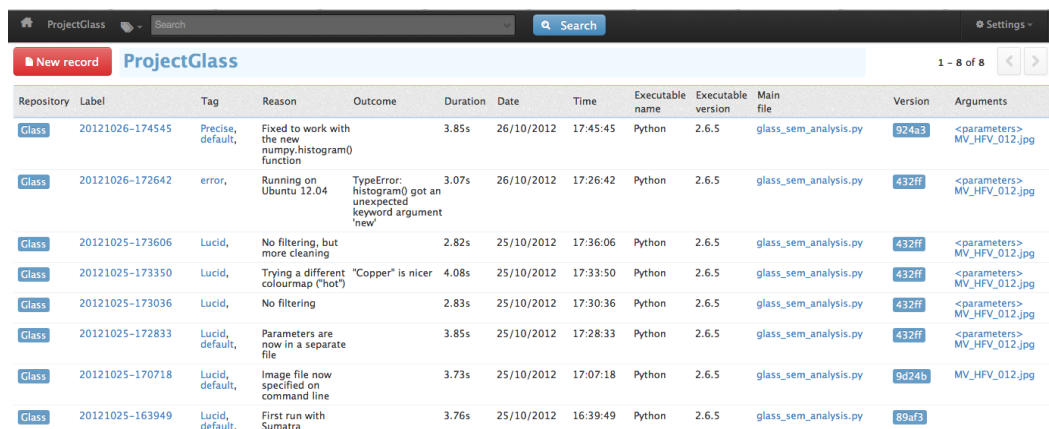
Excerpts from a side-by-side comparison of two computation records, one run on Ubuntu 10.04, the other on Ubuntu 12.04.

```

Dependencies differ      : yes
Launch mode differs     : no
Input data differ       : no
Script arguments differ : no
Parameters differ       : no
Data differ             : no

```

She has slightly different versions of the dependencies on her MacBook, but the results are unchanged. Alice can now proceed to reformat the figures, confident that her computing environment is consistent with that of her graduate student. Since the grant application is being written in  $\text{\LaTeX}$ , Alice can also use the `sumatra`  $\text{\LaTeX}$  package to automatically pull images from the Sumatra record store into her document, with automatic cross-checking of SHA1 hashes to ensure the image is indeed the correct one and has not been accidentally over-written.



The screenshot shows the ProjectGlass web interface. At the top, there is a search bar and a 'New record' button. Below the header, a table lists computation records. The table has columns for Repository, Label, Tag, Reason, Outcome, Duration, Date, Time, Executable name, Executable version, Main file, Version, and Arguments. The records are sorted by date and time, showing various execution attempts with different parameters and outcomes.

Repository	Label	Tag	Reason	Outcome	Duration	Date	Time	Executable name	Executable version	Main file	Version	Arguments
Class	20121026-174545	Precise, default,	Fixed to work with the new numpy.histogram() function		3.85s	26/10/2012	17:45:45	Python	2.6.5	glass_sem_analysis.py	924a3	<parameters> MV_HFV_012.jpg
Class	20121026-172642	error,	Running on Ubuntu 12.04	TypeError: histogram() got an unexpected keyword argument 'new'	3.07s	26/10/2012	17:26:42	Python	2.6.5	glass_sem_analysis.py	432ff	<parameters> MV_HFV_012.jpg
Class	20121025-173606	Lucid,	No filtering, but more cleaning		2.82s	25/10/2012	17:36:06	Python	2.6.5	glass_sem_analysis.py	432ff	<parameters> MV_HFV_012.jpg
Class	20121025-173350	Lucid,	Trying a different colourmap ("hot")	"Copper" is nicer	4.08s	25/10/2012	17:33:50	Python	2.6.5	glass_sem_analysis.py	432ff	<parameters> MV_HFV_012.jpg
Class	20121025-173036	Lucid,	No filtering		2.83s	25/10/2012	17:30:36	Python	2.6.5	glass_sem_analysis.py	432ff	<parameters> MV_HFV_012.jpg
Class	20121025-172833	Lucid, default,	Parameters are now in a separate file		3.85s	25/10/2012	17:28:33	Python	2.6.5	glass_sem_analysis.py	432ff	<parameters> MV_HFV_012.jpg
Class	20121025-170718	Lucid, default,	Image file now specified on command line		3.73s	25/10/2012	17:07:18	Python	2.6.5	glass_sem_analysis.py	9d24b	MV_HFV_012.jpg
Class	20121025-163949	Lucid, default,	First run with Sumatra		3.76s	25/10/2012	16:39:49	Python	2.6.5	glass_sem_analysis.py	89af3	

**FIGURE 1.3**

List of computation records in the Sumatra web browser interface.

In conclusion, we hope to have demonstrated that by using Sumatra, Alice and Bob have improved the reproducibility of their computational experiments, enhanced communication within their lab, and increased the manageability of their projects, with minimal effort and minimal change to their existing workflow.

### 1.3 Design criteria

In introducing the architecture of Sumatra so that others can build upon and extend it, we begin by describing the constraints we wish Sumatra to satisfy, before describing, in the following section, its current architecture.

The design of Sumatra is driven by two principles:

1. there is a huge diversity in computational science workflows;
2. software to assist reproducibility must be very easy to use, or only the very conscientious will use it.

To elaborate on the first issue, of workflow diversity, different scientists may launch computations from the command-line, in interactive notebooks, in graphical interfaces, in web-based tools. Computations may be launched serially, as batch jobs, as distributed computations, for immediate execution or queued for deferred execution, on local machines, small clusters, supercomputers, grids, or in the cloud. Projects may be solo or collaborative efforts. Different workflows may be used for different components of a project or during different phases of a project (e.g. exploration vs preparation of final published figures).

Given this diversity, it is unlikely there is a single software tool to support reproducible research which will be optimal for all possible workflows. At the same time, there is a considerable amount of functionality that is required whatever the workflow, e.g. unambiguous identification of exactly which code has been run. Sumatra is therefore designed as a core library of loosely-coupled components for common functionality, easily extensible and customizable, so people can adapt Sumatra to their own use cases, and so other people can build other tools on top of Sumatra.



Such a library is potentially useful to tool developers, but will not on its own promote reproducibility: it must be integrated into scientists' existing workflows, so that the barrier to adoption is as low as possible. Sumatra also, therefore, provides tools, built on top of the core library, that wrap around or work alongside widely-used types of workflow. Three such tools are available at the time of writing: `smt`, which supports workflows built around running individual computations on the command line; `smtweb`, which provides a browser-based tool for browsing and querying the results of previous computations; and a LaTeX package which allows the automated inclusion of figures generated by a Sumatra-tracked computation in documents, with hyperlinks to the provenance information. The use of these tools was demonstrated in the previous section. In the future, further tools may be developed to support more interactive workflows.

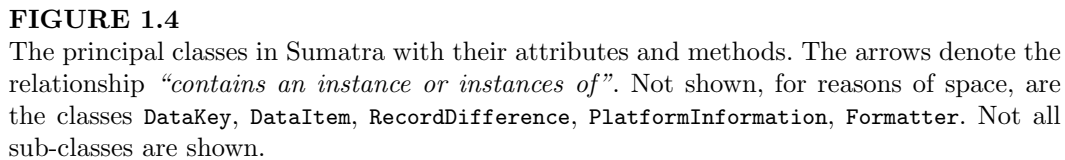
Given the above constraints, Sumatra must enable a scientist to easily respond to the following questions:

- what code was run?
  - which executable?
    - \* name, location, version, compilation options
  - which script?
    - \* name, location, version
    - \* options, parameters
    - \* dependencies (name, location, version)
- what were the input data?
  - name, location, content
- what were the outputs?
  - data, logs, stdout/stderr
- who launched the computation?
- when was it launched/when did it run? (queueing systems)
- where did it run?
  - machine name(s), other identifiers (e.g. IP addresses)
  - processor architecture
  - available memory
  - operating system
- why was it run?
- what was the outcome? (interpreted in terms of the ongoing project)
- which project was it part of?

---

## 1.4 Architecture

This section gives an overview of Sumatra's architecture, intended for readers who may be interested in extending or building upon Sumatra, or applying some of its methods in their own approaches to replicability. More fine-grained detail is available in the online documentation at <http://neuralensemble.org/sumatra>. Sumatra has a modular design, with the coupling between modules made as loose as possible. Within modules, a common motif to provide flexibility and configurability is to use abstract base classes to define a common interface, which are then subclassed to provide different implementations of a given type of functionality (e.g. version control, data storage). The principal classes in the core Sumatra library, and their composition, are shown in Figure 1.4. More detail about the individual modules, classes and their interactions is given in the following sections.



To ensure replication, we need to capture identifying information about all of the code that was run. Where code is modular, this means capturing the local file system path of each library/module/package that is included/imported by the “main” file (its “dependencies”), together with, if possible, the version of the module, so that (i) the environment could be recreated in future, (ii) if failing to replicate with more up-to-date versions of libraries in future, we can investigate what has changed. This must be done recursively, of course, if a dependency itself has dependencies.

Version information may be provided in many ways, some of which are dependent on the programming language used, others independent. As an example of the former, Python modules often define a variable called `__version__`, `VERSION` or `version`, or a function called

`get_version()`. Two examples of the latter are: obtaining the version from a VCS; obtaining the version from a package management system (such as apt, on Debian). Sumatra's strategy, therefore, is that each `dependency_finder` module provide a list of functions, each implementing one heuristic for finding versions, e.g. `find_versions_by_attribute()`, or `find_versions_by_version_control()`. Each of these is tried in turn, and the first version found is the one used (the order is important: generally a version obtained from a VCS is more reliable/precise than a version obtained from a variable defined within the code).

It may happen that some of the code under version control has been modified since the last commit. In this scenario, it is usually best to abort the computation and to commit the changes before proceeding. However, there may be good reasons for not wanting to commit, and so Sumatra also provides the option of storing the “diff” between the VCS working copy and the last commit.

Given the variety of VCSs in use, Sumatra's strategy is to wrap each VCS so as to provide a uniform interface. For each VCS supported by Sumatra, the `versioncontrol` module contains a sub-module containing two classes - a subclass of `versioncontrol.base.Repository` and a subclass of `versioncontrol.base.WorkingCopy`. Sumatra does not require all the functionality of VCSs, and is not intended to replace the normal methods of interacting with a VCS for code development. The `Repository` subclass has two roles: storing the repository URL, and obtaining a fresh checkout/clone of the code base from a remote server (even the latter is not strictly necessary). The functionality required of the `WorkingCopy` subclass is more extensive: determine the current version; determine whether any of the code has been modified; determine the diff between the working code and the last commit; determine whether a given file is under version control; change the working copy to an older or newer version (for replicating previous computations and then returning to the most recent state of the code base).

In general, the difference between distributed and centralized version control systems is not important for Sumatra. The only difference is that, for distributed VCSs, the repository used is always a local one, and it is therefore often useful, for the purposes of future replication and open science, to store the URL of the “upstream” repository, often a public repository on a remote server.

### 1.4.2 Data handling

Replicability of a computational result requires knowing what the input data (if any) were, and it requires storing the output data so that future replication attempts can be checked against the original results. Inputs to a program can be subdivided into data, and configuration/parameters. These can generally be distinguished in that data could be processed by a different program, while parameters are tightly tied to the code. Sumatra attempts to distinguish parameter/configuration files from input data files by the structure of the data; as a fall-back, parameters will be treated as input data. Parameter file handling is described below.

Data may be stored in many ways: in individual files on a local or remote file system, in a relational database, in a remote resource accessed over the internet by some API. However it is stored, the most important thing to know about data is its content. However, it would be redundant for Sumatra to store a separate copy of each input and output data item, especially given the potentially enormous size of data items in many scientific disciplines. Sumatra therefore stores an identifier for each data item, which enables retrieval of the item from whichever data store – the file system, a relational database, etc. – is used. In the case of the file system, for example, the identifier consists of the file system path relative to a user-defined root directory together with the SHA1 hash of the file contents. The latter is needed to catch overwriting or corruption of files.

To handle different ways of storing data, Sumatra defines an abstract `DataStore` class, which is then subclassed: for example, the `FileSystemDataStore` which is used to work with data stored on a local file system. The minimal functionality required of a `DataStore` subclass is: find new content, given a time stamp (used to link output data to a given computation); return a data item object, given the item's identifier ("key"); return the contents of a data item; delete a data item. `DataItem` objects support obtaining the data contents, and may also contain additional metadata, such as the mimetype.

It is straightforward to add extra functionality to a `DataStore` subclass. For example, the `ArchivingFileSystemDataStore` works the same as the plain `FileSystemDataStore` but in addition copies all the output data files to an archive format. The `MirroredFileSystemDataStore` allows specifying a URL from which the data file can be retrieved (in addition to the local version). This supports, for example, using Dropbox (<https://www.dropbox.com>) with a public folder, or FTP, or FigShare (<http://figshare.com>) to make your data available online.

### 1.4.3 Storing provenance information

Once Sumatra has captured the context of your computational experiment, it needs to store all this information somewhere. For individual projects, a local database is probably the best way to do this. For collaborative projects, or if you often work while travelling, it may be necessary for this information to be stored in a remote database accessible over the internet. To provide this flexibility, Sumatra defines an abstract `RecordStore` class, which is then subclassed.

Sumatra currently provides three `RecordStore` subclasses: `ShelveRecordStore`, which provides only basic functionality, but has the advantage of requiring no external libraries to be installed; `DjangoRecordStore`, which uses the Django web framework to store the provenance information in a relational database (SQLite by default, but MySQL, PostgreSQL and others are also supported) and adds the ability to browse the record store using a web browser; and `HttpRecordStore`, which is a client for storing provenance information in a remote database accessed over HTTP using JSON as the transport format. The server for the `HttpRecordStore` is not distributed with Sumatra, but such a server is straightforward to implement. Two implementations currently exist – a Django-based implementation at [https://bitbucket.org/apdavison/sumatra\\_server](https://bitbucket.org/apdavison/sumatra_server) and a MongoDB-based version at <https://github.com/btel/Sumatra-MongoDB>.

The functionality required of a `RecordStore` subclass is: support multiple Sumatra projects; list all projects contained in the store; save a Sumatra `Record` object under a given project; list all the records in a project; retrieve a `Record` given its identifier (project+label); delete a `Record` given its identifier; delete all `Records` which have a given tag; return the most recent record; export a record in JSON format; import a record in the same format; synchronize with another record store so that they both contain the same records for a given project.

### 1.4.4 Parameter handling

It is a common practice in scientific computing to run a simulation or analysis with different parameters and to compare the results. Given this important use case, Sumatra allows parameters to be handled differently from other input data. If Sumatra is able to recognize a particular parameter file format then (i) the parameters are available for future searching/querying/comparison; (ii) Sumatra can add extra parameters. An important use case of the latter is that Sumatra can add the label/identifier for the current record, for use by the user's code in constructing file names, etc. Sumatra currently supports four parameter

file formats, including simple “key=value” files, JSON, and config/ini-style formats. Implementing support for a new parameter file format is straightforward: define a `MyParameterSet` class whose constructor accepts either a filename or a text string containing the file contents. The class should also implement method `as_dict()` which returns parameter names and values in a (possibly nested) Python dict, `update()`, which functions like `dict.update()`, and `save()`, which writes the parameter set to file in the given format.

### 1.4.5 Launching computations

If your code is written in Python, then you can use Sumatra directly within your scripts, and run your computation with Python as usual. If you are using other tools (or if using Python and you do not want to modify your code) then Sumatra needs to launch your computation in order to be able to capture the context. The challenge here is that there are so many different workflows, so many different ways of launching a computation: from the command line on the local machine, from the command line on a remote machine (using ssh, for example), on a cluster, computing grid or supercomputer using a job manager, as a parallel computation using MPI, or by clicking a button in a graphical interface

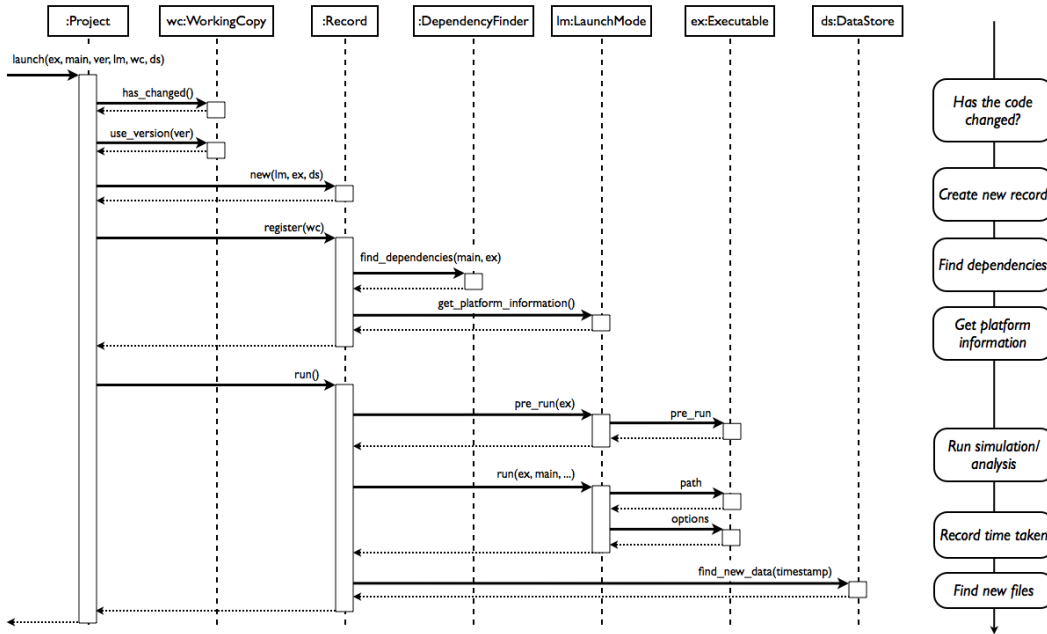
To handle this variety, Sumatra follows the usual pattern of defining an abstract base class, `LaunchMode`, which is then subclassed to support different methods of launching computations. A `LaunchMode` subclass needs to define a method `generate_command()` which should return a string which will be executed on the command line. The `LaunchMode` is also responsible for capturing information about the platform – the operating system, the processor architecture, etc. For computations run on the local machine, the base class takes care of this. For computations run on a remote machine or machines, the `LaunchModel` subclass must override the `get_platform_information()` method. Sumatra currently provides `SerialLaunchMode` and `DistributedLaunchMode` subclasses.

To generate the launch command, Sumatra may need extra information about the particular executable being used – particular arguments or flags that are needed in different circumstances. Similarly, there may be a build step or other preliminary that is needed before launching the computation. If this is the case, a user may define an `Executable` subclass which may define any of the attributes `pre_run`, `mpi_options`, `requires_script`, and may optionally redefine the method `_get_version()`. The user then calls the `programs.register_executable()` method to register the new subclass with Sumatra.

### 1.4.6 Putting it all together

Tying all of the foregoing together are the `Record` class and the `Project` class. The `Record` class has two main roles: gathering provenance information when running a computation, and acting as a container for provenance information. When launching a new computation, as diagrammed in Figure 1.5, a new `Record` object stores the identifiers of any input data, interacts with a `WorkingCopy` object to check that the code is at the requested version, uses the `dependency_finder` module to find the list of dependencies (and their versions), and then obtains platform information from the appropriate `LaunchMode`. It then runs any precursor tasks, such as building the executable, writes a modified parameter file, if necessary, and then passes control to the `LaunchMode`, which spawns a new process in which it runs the requested computation while capturing the standard output and standard error streams. Once this completes, the `Record` object calculates the time taken, stores stdout and stderr, asks the `DataStore` object to find any new data generated by the computation, and stores the identifiers of this output data.

The `Project` class has one main role: to simplify use of the Sumatra API by storing

**FIGURE 1.5**

The flow of control between different Sumatra objects during a computation. Time flows from top to bottom. Each dashed vertical line represents the life time of an object, labelled at the top with the class and, in some cases, an instance name. Solid horizontal arrows represent method calls or attribute access.

default values and providing shortcut functions for frequently-performed tasks. Thus, for example, while creating a new `Record` object requires passing up to 16 arguments, the `Project.new_record()` method will often be called with just two – the parameter set and the list of input data items – since most of the others take default values stored by the `Project`. The `smt` command accesses Sumatra’s functionality almost entirely through an instance of the `Project` class.

The precise division of responsibilities between the `Record` and `Project` class is not critical, and could evolve in future versions of Sumatra to enhance usability of the API.

#### 1.4.7 Search/query/reuse

So far we have talked about the API from the perspective of capturing provenance information. We now consider the use cases of accessing, querying and using the stored provenance information.

As described above, this information is stored in a “record store”, represented by a subclass of `RecordStore`, and whose backend may be a flat file, relational database, or web service. The common record store interface allows querying based on record identifiers (project + label) and on tags. Individual record store implementations may allow more sophisticated queries: for example, the `DjangoRecordStore` allows queries based on Django’s object-relational-mapper, or even using plain SQL.

The main use cases for accessing records of previous computations: (i) are comparing the results of similar runs (e.g. examining the effects of parameter changes); (ii) repeating

a previous computation to check that the results are reproducible; (iii) further processing of results, e.g. further analyses, visualization, inclusion in manuscripts.

The first two of these use cases are supported by the `Project.compare()` method, which calls `Record.difference()`, which returns an instance of the `RecordDifference` class. This class has assorted methods which allow a precise dissection of the differences between two computations.

---

## 1.5 Discussion

In this chapter we have presented Sumatra for two (albeit overlapping) audiences: the working computational scientist, and the software developer or scientist-developer who may wish to extend or build-upon Sumatra. In this book as a whole, a number of different tools to support reproducible research have been presented. For a scientist interested in ensuring their research is easily reproducible, when should you use Sumatra and when another tool?

Software for reproducible research can be divided into three general categories: tools for literate programming, workflow management systems, and tools for environment capture.

Literate programming<sup>1</sup> and the closely related “interactive notebook” approach<sup>2</sup> inextricably bind together code and the results generated by that code, which is clearly hugely beneficial for reproducible research. With some such systems, information about software versions, input data and the computing environment can also be included in the final document. If your literate programming environment or interactive notebook supports Python, you could also use Sumatra via its API to provide this functionality. Scenarios that are generally more difficult to handle with the current generation of literate programming tools and interactive notebooks are: (i) where computations take considerable time (hours or days) to run; (ii) where computations are distributed on parallel hardware or are queued for later execution; (iii) where code is split among many modules, so that the code included in the literate document or notebook is only a small part of the whole.

Visual workflow management or pipeline tools, such as Kepler [8], Taverna [10] and VisTrails [3, 4] are aimed at scientists with limited coding experience, or who prefer visual programming environments. They are particularly straightforward to use in domains where there some standardization of data formats and analysis methods – for example in bioinformatics and in fields that make extensive use of image processing. The main disadvantage is that where there are no pre-existing components for a given need, creating a new component can require considerable effort and a detailed knowledge of the workflow system architecture. Most widely-used systems include provenance tracking either as an integral part or as an optional module.

Environment capture systems, such as Sumatra, are generally the easiest to adopt for an existing workflow. The simplest approach is to capture the entire operating system as a virtual machine (VM) image – see the chapter by Howe [7] in the current volume. A more lightweight alternative to this is CDE[5, 6], which archives only those executables and libraries actually used by the computation. The main disadvantages with such approaches are: (i) your results risk being highly sensitive to the particular configuration of your computer; (ii) it is difficult or impossible to index, search or analyse the provenance information. Sumatra aims to overcome both of these disadvantages by capturing the information needed

---

<sup>1</sup>see for example ref [2], which explains the use of Sweave (<http://www.statistik.lmu.de/~leisch/Sweave/>) and Org-mode (<http://orgmode.org>) for reproducible research, and ref[11] in the current volume.

<sup>2</sup>for example Mathematica (<http://www.wolfram.com/mathematica/>), Sage (<http://www.sagemath.org>) and IPython (<http://ipython.org>)

to recreate the experimental context, rather than the context itself in binary form. Some combination of Sumatra and CDE would perhaps give the best of both worlds. Integration of CDE is planned in a future version of Sumatra.

In summary, at the time of writing, Sumatra is most suitable for scientists who prefer to write their own code and run it from the command line, especially when factors such as computation time, parallelism, or remote execution make it difficult to work interactively, or where code is highly modular so that literate programming tools capture only the tip of the code iceberg. In any case, Sumatra is fast to set up, easy to use and requires no changes to existing code, so there is little to be lost in trying it out.

We have seen that Sumatra makes it much easier to replicate computational research, in capturing the details of the software and hardware environment that was used. In particular, Sumatra makes it much easier to identify, in the case of failure to reproduce a result, what are the differences between the original and current environments. However, Sumatra cannot *guarantee* reproducibility, for two reasons. First, there are some details that are not captured. For example, in Figure 1.1 you can see that for some of the dependencies the version is unknown, either because the version information is genuinely not present or because Sumatra does not yet have a heuristic for finding it. Similarly, the compilation procedure and software library versions used to compile third-party programs, such as the Python interpreter, are not currently captured, and it may sometimes be impossible to capture this information. Second, with the passage of time, even if you know the particular versions of the libraries used, these versions may no longer be available, or the particular hardware architecture needed may not even be available. This problem is not restricted to Sumatra, of course. The use of virtual machines and careful archiving of old hardware is one partial solution, while for code that continues to be useful, a program of maintenance and ongoing updates can avoid obsolescence.

In the future, we plan to add support for using Sumatra with interactive notebooks (i.e. supporting a more granular unit of computation than an entire script), automated re-creation of software environments using the captured information, support for pipelines (where the output in one Sumatra record is the input in another), better support for compiled languages and software build systems, and interoperability with other provenance tracking tools, probably using the Open Provenance Model [9].

Sumatra is open source software, and is developed as an open community – if you have ideas or wish to contribute in any way, please join us at <http://neuralensemble.org/sumatra>.

---

## Acknowledgments

We would like to thank Eilif Muller, Konrad Hinsén, Stephan Gabler, Takafumi Arakaki, Yoav Ram, Tristan Webb and Maximilian Albert for their contributions to Sumatra, as well as everyone who has reported bugs on the issue tracker. The code examples of SEM image analysis were based on the SciPy tutorial at <http://scipy-lectures.github.com/>.



---

## Short biographies

Andrew P. Davison is a senior research scientist in the Unité de Neurosciences, Information and Complexité of the Centre National de la Recherche Scientifique, where he leads the Neuroinformatics group. His research interests focus on biologically detailed modeling, simulating neuronal networks (particularly the mammalian visual system), and developing tools to improve the reliability, reproducibility, and efficiency of biologically realistic simulation. Davison has a PhD in computational neuroscience from the University of Cambridge.

During his joint PhD at the University of Cambridge and the European Bioinformatics Institute, Michele Mattioni worked on multiscale modelling, in particular focusing on the synaptic plasticity of the Medium Spiny Neuron of the Striatum, where he developed a new event-driven algorithm which permits the integration of simulators working in different timescales. He is also the main developer of Neuronvisio, a package to visualize, simulate and explore neuronal models.

Dmitry Samarkanov is a PhD student at the Ecole Centrale de Lille, Laboratory of Electrical Engineering and Power Electronics. In 2008 he received an MS degree in Electrical Engineering from the Moscow Power Engineering Institute. His research interests focus on methods of multi-objective optimization, reproducibility in electrical engineering and development of tools for managing simulations.

Bartosz Teleńczuk is a post-doctoral fellow in the Unité de Neurosciences, Information and Complexité of the Centre National de la Recherche Scientifique. He obtained his PhD in biophysics from Humboldt University in Berlin. His work is focused on combining recordings of neuronal activity measured at various scales to understand the processing of sensory activity in mammalian brains. He is also a committed Python advocate and an instructor in courses on best practices in scientific programming.



---

## Bibliography

---

- [1] Andrew P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science and Engineering*, 14:48–56, 2012.
- [2] Matthieu Delescluse, Romain Franconville, Sébastien Joucla, Tiffany Lieury, and Christophe Pouzat. Making neurophysiological data analysis reproducible: why and how? *Journal of Physiology Paris*, 106:159–70, 2012.
- [3] Juliana Freire. Making computations and publications reproducible with VisTrails. *Computing in Science and Engineering*, 14(4):18–25, 2012.
- [4] Juliana Freire, David Koop, Fernando Seabra Chirigati, and Cláudio T. Silva. Reproducibility using VisTrails. In Victoria Stodden, Friedrich Leisch, and Roger Peng, editors, *Implementing Reproducible Computational Research*. CRC Press/Taylor and Francis, Boca Raton, Florida, 2013.
- [5] Philip J. Guo. CDE: A tool for creating portable experimental software packages. *Computing in Science and Engineering*, 14:32–35, 2012.
- [6] Philip J. Guo. CDE: Automatically package and reproduce computational experiments. In Victoria Stodden, Friedrich Leisch, and Roger Peng, editors, *Implementing Reproducible Computational Research*. CRC Press/Taylor and Francis, Boca Raton, Florida, 2013.
- [7] Bill Howe. Reproducibility, virtual appliances and cloud computing. In Victoria Stodden, Friedrich Leisch, and Roger Peng, editors, *Implementing Reproducible Computational Research*. CRC Press/Taylor and Francis, Boca Raton, Florida, 2013.
- [8] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.
- [9] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, July 2010.
- [10] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: Lessons in creating a workflow environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.
- [11] Yihui Xie. knitr: a comprehensive tool for reproducible research in R. In Victoria Stodden, Friedrich Leisch, and Roger Peng, editors, *Implementing Reproducible Computational Research*. CRC Press/Taylor and Francis, Boca Raton, Florida, 2013.