Best practices for data management in neurophysiology

Release 0.1

Andrew P. Davison

October 13, 2014

Andrew Davison

Unité de Neurosciences, Information et Complexité (UNIC), Centre National de la Recherche Scientifique, 91198 Gif sur Yvette, France. http://andrewdavison.info

Notes for a tutorial given at the GDR Multielectrodes minischool (http://gdr2904gif.sciencesconf.org/resource/page/id/3). Available to download in PDF (https://media.readthedocs.org/pdf/rrcns/gdr_multielectrode_2014/rrcns.pdf) and Epub (https://media.readthedocs.org/epub/rrcns/gdf_multielectrode_2014/rrcns.epub) formats.

Version 0.1

October 13th 2014

CHAPTER 1

Contents

1.1 Why data management?

The short answer:

- save time;
- do better science.

By taking some time at the beginning of a project to think carefully about how you will handle your data, simulations and analysis results; or, if you are already in the middle of a project, taking some time to reflect on your data handling and, possibly, reorganise your existing data and results, you can save yourself a huge amount of time and energy in the future, as well as improve the reproducibility of your results.

Some of the problems arising from poor data management:

- taking too much time to find relevant files;
- increased risk of errors and mistakes;
- problems understanding what you did six months ago;
- problems working with collaborators;
- difficulty in explaining things to new students;
- general confusion...

These problems are particularly acute in systems and computational neuroscience, especially when using MEAs, due to the complexity of the experimental protocols, the large number of parallel channels and of correlations between them, and the consequent complexity of the data analysis workflows used.

Probably less important at the moment, but perhaps of increased importance in the future, many funding bodies require data management plans, and in future they, and perhaps journals, may increasingly require depositing your data in a public repository on completion of the project/publication of the paper.

1.2 The data life cycle

This tutorial is structured around the idea of the data life cycle.

images/data_life_cycle.jpg

The stages we will discuss are:

- plan - self explanatory
- collect - data acquisition
- assure - quality assurance, cross-checking, pre-processing, data selection
- describe - annotation, metadata, lab notebooks
- preserve - long-term storage
- analyze/integrate/discover - data analysis

1.3 Data acquisition

1.3.1 Where possible, store data in nonproprietary software formats

The software needed to read proprietary file formats can become unavailable - companies change the format, go out of business, your lab no longer has a licence... For these reasons you should store your data, where possible, in a widely used, documented, non-proprietary format for which there are multiple software tools available. Examples include plain text or RTF instead of Microsoft Word, csv instead of Excel, HDF5 for time series data.

In neurophysiology, this is easier said than done, since each recording equipment manufacturer tends to provide their own, proprietary file format [give examples, Plexon, etc.]. Nevertheless, converting these proprietary formats to a more standard one will make subsequent data analysis easier (since you will have a wider choice of tools), facilitate collaboration (since your collaborators may not have access to the same software) and ensure you can still read the data in five or ten years time. For neurophysiology data there are now a number of tools that can perform these conversions, including Neo, sigTool and Neuroshare. *Do not, however, delete the original file after you have converted it!* You should always preserve the original files, so as to be able to prove/check/confirm that no data have been lost/corrupted during the conversion process.

1.3.2 Keep backups on stable media, and in geographically separated locations

- Backup your data as soon as possible after acquiring it.
- If possible, try to keep at least two backups, one in the lab and one several kilometres, at least, from the lab. This is in case of fires, earthquakes, etc.
- At least one backup should be on a stable, long-term storage medium, e.g. CD-R or DVD-R at the time of writing.
- Every 2 years, check whether your backup media are still readable, and consider moving them to a new medium (e.g. most Macbooks no longer have internal CD/DVD drives).
- Consider storing your data in an online repository (e.g. G-Node, INCF Dataspace, Dryad; see Databib (http://databib.org) for a full list).

1.3.3 Have a clear organisation for your data files

There is no one best organisation, but this is something that should ideally be planned in advance, and everyone working on a project needs to agree on. A structure which often works well is PROJECT/YEAR/MONTH/DAY/SUBJECT, but many variations are possible.

- Ensure everything has a time stamp, using standardized time/date formats, i.e. ISO 8601 YYYY-MM-DDThh:mm:ss
- Descriptive filenames are often helpful, but don’t try to store too much information in the filenames, it is better to use a “database” (could be a real database, or just a text file with the same name as the data file but a different extension) of metadata which allows all the contextual information to be linked to the file.
- Have a single canonical location for your files. If you need to make copies (e.g. to work on your laptop) ensure it is clear which is the “master” copy.

1.3.4 Always maintain effective metadata

Metadata are all the pieces of information needed to understand, make sense of and analyze your data - at a minimum, everything which goes in the *Materials and Methods* section of your paper.

There are many ways to store metadata. The classical way is to keep a paper lab notebook. Although this is sometimes a legal requirement, there are many advantages to storing metadata digitally, either instead of, or as well as, a paper lab notebook. (If medical doctors can increasingly use electronic patient records in place of paper, why can't scientists?)

In order of increasing complexity, electronic metadata can take one of the following forms:

- a simple text file, "README" style
- a spreadsheet
- a more structured metadata file. odML is a good format for neurophysiology.
- a home-made relational database. If you're comfortable with SQL, this is a way to have a tool which is custom-made for your particular experiment
- dedicated software, either open-source (Yogo, Helmholtz, ...) or commercial (e.g. Ovation).

Note that it is often useful to store metadata in two places: first, as part of, or next to, the datafile, so that the data never becomes separated from its context; secondly in a database of some kind, for ease of search and meta-analysis. This redundancy carries the risk that the two sources of metadata get out of sync, although this can also be seen as a safety check.

1.4 Pre-processing

In pre-processing, we include everything that is done to the data between acquisition and analysis (although the dividing line between acquisition and analysis is not always clear). This can include assuring the quality of the data through checks and inspections, data cleaning and data selection, e.g. exclusion of cells or periods of time where there was some problem with the recording (50 Hz noise, etc.).

The most important best practice is this:

Important: never change a raw data file (e.g. to remove artifacts, bad recordings), instead copy the file and edit the copy, recording how you got from one to the other. To enforce this, make raw data read-only.

Ideally the transformation should be scripted/automated, i.e. set objective standards for when to exclude data from further analysis, and write a program to implement the application of these standards. If you can't easily automate the process, keep detailed notes of what you did and why.

Consider maintaining separate directory trees for the raw and cleaned-up data files.

1.5 Data analysis

Important: Script your data analysis wherever possible.

- If you use an interactive tool to explore your data, then once you've finished, go back and write a script which repeats what you just did.
- If you cannot script the analysis (e.g. spike sorting often requires manual intervention), document exactly what you did and why; include screenshots if using a graphical interface; write down all the settings/configuration values used by the software.

Some general advice on writing code for data analysis...

1.5.1 Keep code and data separate

- Do not keep code and data in the same directory
- Do not keep raw data and analysis results in the same directory

1.5.2 Use version control

See *Version control*.

1.5.3 Test your code

See *Verification (testing)*.

1.5.4 Split commonly used code off into functions/classes, and put these into libraries

Warning: do not cut-and-paste; ideally the same code should never appear twice.

1.5.5 Prioritize code robustness

By “robustness” here, I mean insensitivity to the precise details of the code and environment: if you try to make one part of the code run faster, does the rest of the code have to be changed as well? If you change to a different Linux distribution, or upgrade your operating system, does the code still run and do you get the same results.

Strategies for more robust code are widely employed in professional software development and have been described in many places [e.g. S. McConnell, Code Complete, 2nd ed., Microsoft Press, 2004.] 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, well-tested and easy-to-install libraries; and
- writing test suites.

In particular, you should design your code to be easily understood by others (where “others” can also include “yourself-in-six-months-time”):

- write comments explaining anything slightly complex, or why a particular choice was made;
- write clear documentation;
- don’t be too clever.

On the latter point, Brian Kernighan said:

> “*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*“

How far should you go in trying to make your code better? Making code more robust has costs in time and manpower, which might not be worth incurring for scientific code with a limited number of users. At the same time, making these time investments up front can save a lot of time later. I don’t have any good guidelines for knowing what the right balance is, other than to step back from the project from time to time, think about how much effort you’ve expended, and decide whether it feels like you’re making too much, or not enough, effort to make your code more reproducible given your goals (e.g., publication).

1.5.6 Maintain a consistent, repeatable computing environment

If you're moving a computation to a new system, it should be simple and straightforward to set up the environment identically (or nearly so) to that of the original machine. This suggests either using a package-management system - for example, the Debian, Red Hat, or MacPorts systems - or a configuration-management tool (such as Puppet, Chef, or Fabric).

The former provide prebuilt, 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 isn't available in a package-management system, whether because it hasn't been packaged (for example, because it isn't widely used or is developed locally for internal use) or the package manager's version is too outdated.

1.5.7 Separate code from configuration

It's good practice to cleanly separate code from the configuration and parameters. There are several reasons for this:

- the configuration and parameters are changed more frequently than the code, so different recording tools are most appropriate for each - for example, using a VCS for the code and a database for the parameters;
- the parameters are directly modified or controlled by the end user, but the code might not be - this means that parameters can be controlled through different interfaces (configuration files or graphical interfaces);
- separating the parameters ensures that changes are made in a single place, rather than spread throughout a code base; and
- the parameters are useful for searching, filtering, and comparing computations made with the same model or analysis method but with different parameters, and storing the parameters separately makes such efforts easier.

1.5.8 Share your code

(more on this in the next update to these notes)

1.6 Version control

1.6.1 Basic ideas

Any time multiple versions of a document exist, whether due to a document changing over time, or because multiple authors are working on it, some kind of version control is needed.

Version control allows:

- accessing any version from the original to the most recent;
- seeing what has changed from one version to the next;
- giving a label of some kind to distinguish a particular version.

1.6.2 Examples of version control systems

The simplest method of version control is probably the most widely used in science: changing the file name.

(http://www.phdcomics.com/comics.php?f=1323)

Other examples include:

(http://www.phdcomics.com/comics.php?f=1531)

Figure 1.1: from *"Piled Higher and Deeper" by Jorge Cham* www.phdcomics.com

- "track changes" in Microsoft Word
- Time Machine in Mac OS X
- versioning in Dropbox, Google Drive
- formal version control systems such as CVS, Subversion, Mercurial, Git

1.6.3 The importance of tracking projects, not individual files

Early version control systems, such as CVS, track each file separately - each file has its own version number. The same is true of Dropbox, Microsoft Word.

This is a problem when you make changes to several files at once, and the changes in one file depend on changes in another.

In modern version control systems, and in backup-based systems such as Time Machine, entire directory trees are tracked as a unit, which means that each version corresponds to the state of an entire project at a point in time.s

1.6.4 Advantages of formal version control systems

- explicit version number for each version
- easy to switch between versions
- easy to see changes between versions
- tools to help merge incompatible changes

In the next sections, we will use Mercurial (http://mercurial.selenic.com), one of the most commonly used, modern version control systems, to introduce the principles of version control. We will use Mercurial's command-line interface because it is easy to use, and widely used. Following this, we will briefly discuss Git (http://git-scm.com/) and Subversion (http://subversion.apache.org/), two other widely-used version control systems, as well as graphical user interfaces for version control.

1.6.5 Installing Mercurial

Mercurial is available for Linux, Mac OS X, Windows, and assorted flavours of UNIX. For Linux, it will certainly be available in your package manager. For Windows and Mac OS X, download from http://mercurial.selenic.com/wiki/Download

Once you've installed it, you should create a file named `.hgrc` in your home directory, as follows:

```
[ui]
username = Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
```

1.6.6 Creating a repository

We start by introducing two concepts:

Working copy the set of files that you are currently working on

Repository a "database" containing the entire history of your project (all versions)

As an example, we will use the Brian code from this paper:

Brette R, Rudolph M, Carnevale T, Hines M, Beeman D, Bower JM, Diesmann M, Morrison A, et al. (2007) Simulation of networks of spiking neurons: A review of tools and strategies. *J Comp Neurosci* **23**:349-98

available from http://senselab.med.yale.edu/modeldb/showmodel.asp?model=83319

```
$ unzip destexhe_benchmarks.zip
$ cd destexhe_benchmarks
$ cp -r Brian ~/my_network_model
$ cd ~/my_network_model
$ ls
COBA.py               COBAHH.py       CUBA.py         README.txt
```

We're going to take this code as the starting point for our own project, and we want to keep track of the changes we make.

The first step is to create a repository, where all the versions will be stored. This is very simple:

```
$ hg init
```

Nothing seems to have happened. In fact, the **hg init** command has created a new subdirectory:

```
$ ls -a
.           ..              .hg             COBA.py         COBAHH.py       CUBA.py         READM
```

You almost never need to care about what is in this directory: this is where Mercurial will store all the information about the repository.

1.6.7 Adding files to the repository

Now we need to tell Mercurial which files are part of our project:

```
$ hg add
ajout de COBA.py
ajout de COBAHH.py
ajout de CUBA.py
ajout de README.txt

$ hg status
A COBA.py
A COBAHH.py
A CUBA.py
A README.txt
```

1.6.8 Committing changes

These files are now *queued* to be added to the repository, but they are not yet there. Nothing is definitive until we make a *commit* (also known as a "*check-in*").

```
$ hg commit
```

This pops me into a text editor where I can enter a message describing the purpose of the commit:

```
HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
HG: branch 'default'
HG: added COBA.py
HG: added COBAHH.py
HG: added CUBA.py
HG: added README.txt
```

In this case, I am using **vi**, but you can use any editor.

```
Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?model=83319 or

HG: Enter commit message.  Lines beginning with 'HG:' are removed.
HG: Leave message empty to abort commit.
HG: --
HG: user: Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
HG: branch 'default'
HG: added COBA.py
HG: added COBAHH.py
HG: added CUBA.py
HG: added README.txt
```

1.6.9 Viewing the history of changes

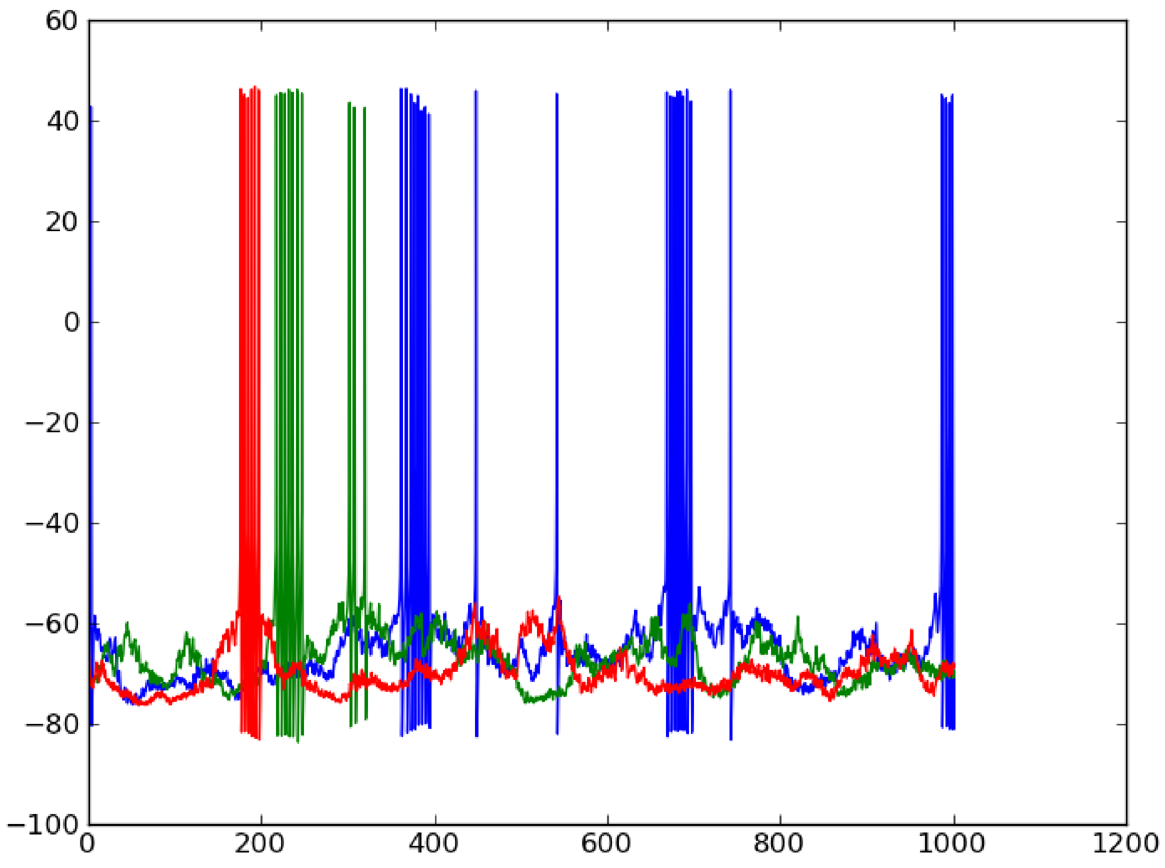
The log command lists all the different versions stored in the repository. For now, of course, we have only one:

```
$ hg log
changeset:   0:ef57b1c87c6a
tag:         tip
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 12:33:21 2012 +0200
summary:     Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?m
```

Now let's run the code:

```
$ python COBAHH.py
Network construction time: 0.814524173737 seconds
Simulation running...
Simulation time: 45.7264661789 seconds
126014 excitatory spikes
29462 inhibitory spikes
```

This pops up a window with the following figure:



We'd prefer to save the figure to a file for further use, rather than work with the model interactively, so let's change the last lines of the script from:

```
plot(trace.times/ms,trace[1]/mV)
plot(trace.times/ms,trace[10]/mV)
plot(trace.times/ms,trace[100]/mV)
show()
```

to

```
plot(trace.times/ms,trace[1]/mV)
plot(trace.times/ms,trace[10]/mV)
plot(trace.times/ms,trace[100]/mV)
savefig("COBAHH_output.png")
```

1.6.10 Seeing what's changed

Now if we run **hg status** we see:

```
$ hg status
M COBAHH.py
```

The "M" indicates that the file has been modified. To see the changes:

```
$ hg diff
diff -r ef57b1c87c6a COBAHH.py
--- a/COBAHH.py     Wed Jul 11 12:33:21 2012 +0200
+++ b/COBAHH.py     Wed Jul 11 15:56:05 2012 +0200
@@ -93,4 +93,4 @@
 plot(trace.times/ms,trace[1]/mV)
 plot(trace.times/ms,trace[10]/mV)
 plot(trace.times/ms,trace[100]/mV)
-show()
+savefig("COBAHH_output")
```

Now let's commit the changes, and look at the log again:

```
$ hg commit -m 'Save figure to file'
$ hg log
changeset:   1:e323d363742a
tag:         tip
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 15:59:02 2012 +0200
summary:     Save figure to file

changeset:   0:ef57b1c87c6a
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 12:33:21 2012 +0200
summary:     Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?m
```

1.6.11 Switching between versions

To switch between versions (you should not do this if you have modified any of the files - commit your changes first), use **hg update**:

```
$ hg update 0
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

This will change the files in your working copy to reflect the state they had when you committed that particular version.

Using **hg summary** we can see which version we are currently using:

```
$ hg summary
parent: 0:ef57b1c87c6a
 Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?model=83319
branch: default
commit: 1 unknown (clean)
update: 1 new changesets (update)
```

When specifying the version number to switch to, you can use either the short form (a decimal integer, like `0` or `1`) or the hexadecimal form (like `ef57b1c87c6a`). The difference between these two forms is discussed below, in Collaborating with others.

With no version number, **hg update** switches to the most recent version:

```
$ hg up
1 files updated, 0 files merged, 0 files removed, 0 files unresolved
$ hg sum
parent: 1:b0275b66ad2b tip
 Save figure to file
branch: default
commit: 1 unknown (clean)
update: (current)
```

Also note that all Mercurial commands can be abbreviated, provided the abbreviation is unambiguous.

1.6.12 Giving informative names to versions

Remembering the version number for a particular version of interest (for example, the version used to generate a particular figure in your manuscript) can be difficult. For this reason, the **hg tag** command can be used to give descriptive and memorable names to significant versions:

```
$ hg tag "Figure 1"
```

Note that this automatically makes a new commit:

```
$ hg log
changeset:   2:416ac8894202
tag:         tip
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 14:28:19 2012 +0200
summary:     Added tag Figure 1 for changeset b0275b66ad2b

changeset:   1:b0275b66ad2b
tag:         Figure 1
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 16:01:32 2012 +0200
summary:     Save figure to file

changeset:   0:ef57b1c87c6a
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 12:33:21 2012 +0200
summary:     Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?
```

You can now switch to a tagged version using the tag name:

```
$ hg update "Figure 1"
```

1.6.13 Recap #1

So far, we have learned how to:

- Create a repository
- Add files to a repository
- Commit changes
- Move your code-base backwards and forwards in time

These operations are so easy and so useful that there is no reason not to use them for almost any work you do as a scientist. Any time I start a new project, whether writing code or writing a paper with LaTeX, I now run **hg init** as soon as I've created a new directory for the project.

1.6.14 Making backups

As well as helping to keep track of different versions of a project, version control systems are hugely useful for keeping backups of your code with minimal hassle.

Making a copy of your repository is as simple as moving to the location where the backup will be, and then using the **hg clone** command.

```
$ cd /Volumes/USB_DRIVE
$ hg clone ~/my_network_model

$ cd ~/Dropbox
$ hg clone ~/my_network_model

$ ssh cluster.example.edu
(cluster)$ hg clone ssh://my_laptop.example.edu/my_network_model
```

You can then keep the backup in-sync with the main repository by either using **hg pull** in the backup location, or using **hg push** in your working directory:

```
$ cd ~/my_network_model
$ hg push /Volumes/USB_DRIVE/my_network_model
pushing to /Volumes/USB_DRIVE/my_network_model
searching for changes
aucun changement trouvé
```

1.6.15 Working on multiple computers

As an extension of the idea of backups, version control systems are excellent for keeping code in sync between multiple computers. Suppose you have a copy of your repository on your laptop, and you were working on the code in the airport.

```
(laptop)$ hg diff
diff -r 416ac8894202 -r 0467691f7881 CUBA.py
--- a/CUBA.py       Thu Jul 12 14:28:19 2012 +0200
+++ b/CUBA.py       Thu Jul 12 15:18:09 2012 +0200
@@ -72,4 +72,4 @@
 print Me.nspikes,"excitatory spikes"
 print Mi.nspikes,"inhibitory spikes"
 plot(M.times/ms,M.smooth_rate(2*ms,'gaussian'))
-show()
+savefig("CUBA_output.png")
(laptop)$ hg commit -m 'CUBA script now saves figure to file'
```

The log on your laptop now looks like this:

```
(laptop)$ hg log
changeset:   3:0467691f7881
tag:         tip
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 15:18:09 2012 +0200
summary:     CUBA script now saves figure to file

changeset:   2:416ac8894202
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 14:28:19 2012 +0200
summary:     Added tag Figure 1 for changeset b0275b66ad2b

changeset:   1:b0275b66ad2b
tag:         Figure 1
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 16:01:32 2012 +0200
summary:     Save figure to file

changeset:   0:ef57b1c87c6a
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 12:33:21 2012 +0200
summary:     Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?m
```

Meanwhile, you've started running some simulations on a local cluster, and you're investigating the effect of changing some parameters:

```
(cluster)$ hg diff
diff -r 416ac8894202 CUBA.py
--- a/CUBA.py       Thu Jul 12 14:28:19 2012 +0200
+++ b/CUBA.py       Thu Jul 12 15:19:49 2012 +0200
@@ -25,9 +25,9 @@
 import time

 start_time=time.time()
-taum=20*ms
-taue=5*ms
```

```
-taui=10*ms
+taum=15*ms
+taue=3*ms
+taui=5*ms
 Vt=-50*mV
 Vr=-60*mV
 El=-49*mV
(cluster)$ hg commit -m 'Changed time constants in CUBA model'
(cluster)$ hg log
changeset:   3:243c20657dc4
tag:         tip
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 15:20:17 2012 +0200
summary:     Changed time constants in CUBA model

changeset:   2:416ac8894202
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 14:28:19 2012 +0200
summary:     Added tag Figure 1 for changeset b0275b66ad2b

changeset:   1:b0275b66ad2b
tag:         Figure 1
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 16:01:32 2012 +0200
summary:     Save figure to file

changeset:   0:ef57b1c87c6a
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Wed Jul 11 12:33:21 2012 +0200
summary:     Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmodel.asp?m
```

Now the repositories on the two machines are out of sync. The first three commits are the same on both, but the fourth is different on the two machines. Note that versions 0, 1, and 2 have the same hexadecimal version number on both machines, but that version 3 has a different hex number:

Laptop	Cluster
0:ef57b1c87c6a	0:ef57b1c87c6a
1:b0275b66ad2b	1:b0275b66ad2b
2:416ac8894202	2:416ac8894202
3:0467691f7881	3:243c20657dc4

This is the reason for having both the short, integer number and the hex version: the short integer is local to each machine, while the hex number is global.

So, how do we get the two machines in sync? This can be done from either machine. Here, we'll do it from the laptop.

```
(laptop)$ hg pull -u ssh://cluster.example.edu/my_network_model
pulling from ssh://cluster.example.edu/my_network_model
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files (+1 heads)
not updating: crosses branches (merge branches or update --check to force update)
```

Note that **hg pull -u** is equivalent to running **hg pull** followed by **hg update**. *"Pull"* pulls changes into the local *repository*, but does not change the *working copy*, i.e. it does not change your files. *"Update"* is the part that changes your files.

Here, the pull succeeded, but the update failed, because we made two different commits on different machines.

```
(laptop)$ hg merge
merging CUBA.py
```

```
0 files updated, 1 files merged, 0 files removed, 0 files unresolved
(branch merge, don't forget to commit)
```

Because Mercurial is clever enough to realize that we'd edited different parts of the file `CUBA.py`, it can automatically merge the two changes. If there was a conflict (if we'd edited the same lines on both machines), the merge would fail and we'd have to manually merge the files (see below).

Mercurial does not automatically commit after the merge, so we have the chance to check we are happy with how Mercurial has merged the files before committing.

```
(laptop)$ hg commit -m 'merge'
```

Now we can see the full history, with all changes:

```
(laptop)$ hg log -r5:2
changeset:   5:12fddba7aaa7
tag:         tip
parent:      3:16e621976c95
parent:      4:243c20657dc4
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 15:54:29 2012 +0200
summary:     merge

changeset:   4:243c20657dc4
parent:      2:416ac8894202
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 15:20:17 2012 +0200
summary:     Changed time constants in CUBA model

changeset:   3:16e621976c95
user:        apdavison
date:        Thu Jul 12 15:40:04 2012 +0200
summary:     CUBA script now saves figure to file

changeset:   2:416ac8894202
user:        Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
date:        Thu Jul 12 14:28:19 2012 +0200
summary:     Added tag Figure 1 for changeset b0275b66ad2b
```

(Note that we've truncated the output by asking for only a subset of the commits).

To complete the sync, we now push the merged repository back to the cluster:

```
(laptop)$ hg push ssh://cluster.example.edu/my_network_model
pushing to ../my_network_model
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
```

1.6.16 Collaborating with others

Using version control systems to collaborate with others is essentially no different to working solo on multiple machines, except that you perhaps have less knowledge of exactly what changes have been made by others.

Suppose my colleague Barbara has also been working on the same code: she cloned my repository at version 0, and since then has been working independently. I'm a little wary of pulling in her changes, so first I can take a look at what she's changed:

```
$ hg incoming /Users/barbara/our_network_model
comparaison avec /Users/barbara/our_network_model
searching for changes
```

```
changeset:   1:40f575c2c5a4
user:        Barbara Bara <barbara@example.com>
date:        Thu Jul 12 16:16:58 2012 +0200
summary:     Changed some parameters in CUBA.py, and saved figure to postscript

changeset:   2:2024998fd5ec
tag:         tip
user:        Barbara Bara <barbara@example.com>
date:        Thu Jul 12 16:17:58 2012 +0200
summary:     Save COBAHH figure to postscript
```

Looks like there may be some problems, since I've also changed parameters in that file, and I'm saving figures to PNG format. Oh, well, deep breath, let's plunge in:

```
$ hg pull -u /Users/barbara/our_network_model
pulling from /Users/barbara/our_network_model
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 2 files (+1 heads)
not updating: crosses branches (merge branches or update --check to force update)

$ hg merge
merging COBAHH.py
warning: conflicts during merge.
merging COBAHH.py failed!
merging CUBA.py
warning: conflicts during merge.
merging CUBA.py failed!
0 files updated, 0 files merged, 0 files removed, 2 files unresolved
use 'hg resolve' to retry unresolved file merges or 'hg update -C .' to abandon
```

Unlike last time, when our changes were in different parts of the file, and so could be merged automatically, here Barbara has changed some of the same lines as me, and Mercurial can't choose which changes to keep.

If we now look at `CUBA.py`, we can see the conflicts marked with <<<<<<< and >>>>>>>:

```
...
from brian import *
import time

start_time=time.time()
<<<<<<< local
taum=15*ms
taue=3*ms
taui=5*ms
=======
taum=25*ms
taue=5*ms
taui=10*ms
>>>>>>> other
Vt=-50*mV
Vr=-65*mV
El=-49*mV

...

<<<<<<< local
savefig("CUBA_output.png")
=======
savefig("firing_rate_CUBA.eps")
>>>>>>> other
```

Well, it makes sense for both me and Barbara to explore different parameters, and it makes sense to allow different file formats, so let's move the parameters into a separate file, and parameterize the file format. The file now looks like this:

```
...
from brian import *
import time
from parameters import TAU_M, TAU_E, TAU_E, FILE_FORMAT

start_time=time.time()
taum = TAU_M*ms
taue = TAU_E*ms
taui = TAU_I*ms
Vt=-50*mV
Vr=-65*mV
El=-49*mV

...

assert FILE_FORMAT in ('eps', 'png', 'jpg')
savefig("firing_rate_CUBA.%s" % FILE_FORMAT)
```

After manually editing `COBAHH.py` as well, I need to tell Mercurial that all the conflicts have been resolved, before I do a commit:

```
$ hg resolve -m
$ hg add parameters.py
$ hg commit -m "Merged Barbara's changes; moved parameters to separate file"
```

I've decided to add the new `parameters.py` to the repository. This means Barbara and I will still have conflicts in future if we're using different parameters, but at least the conflicts will be localized to this one file. It might have been better not to have `parameters.py` under version control, since it changes so often, but then we need another mechanism, in addition to version control, to keep track of our parameters. For more on this issue, see the section on *Provenance tracking*.

I send Barbara an e-mail to tell her what I've done. Now all she has to do is run **hg pull -u**.

```
(barbara)$ cd ~/our_network_model
(barbara)$ hg pull -u /Users/andrew/my_network_model
pulling from /Users/andrew/my_network_model
searching for changes
adding changesets
adding manifests
adding file changes
added 6 changesets with 8 changes to 4 files
4 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Now she has the new file, `parameters.py`, as well as the modified versions of `CUBA.py` and `COBAHH.py`.

1.6.17 Recap #2

You should now be able to use Mercurial for:

- quick and easy backups of your code
- keeping your work in sync between multiple computers
- collaborating with colleagues

1.6.18 A comparison of Git and Mercurial

Git (http://git-scm.com/) is another popular version control system, which shares many concepts and even command names with Mercurial. For simple use there is little to choose between them. The main difference is that

Git has the additional concept of a staging area for arranging exactly what gets committed. With Mercurial, **hg commit** will commit all modified files, while with Git, modified files have to be added to the staging area using **git add**, otherwise they will not be committed.

The following table shows the approximate equivalence between the most common Mercurial and Git commands.

hg clone <url>	git clone <url>
hg diff	git diff HEAD
hg status	git status
hg commit	git commit -a
hg help <command>	git help <command>
hg paths	git remote -v
hg add	git add
hg rm	git rm
hg push	git push
hg pull	git fetch
hg pull -u	git pull –rebase
hg revert -a	git reset –hard
hg revert <some_file>	git checkout <some_file>
hg outgoing	git fetch ; git log FETCH_HEAD..master
hg incoming	git fetch ; git log master..FETCH_HEAD
hg update <version>	git checkout <version>
.hg/hgrc	.git/config
.hgignore	.gitignore

Read up on both, pick one, although if you collaborate a lot with others you will probably end up using both anyway.

1.6.19 A comparison of Subversion and Mercurial

Subversion is a centralized, not distributed, version control system, in that the repository sits on a central server and each user has only a working copy (in contrast to Mercurial and Git, where each user has both repository and working copy). This means that a network connection is required for operations such as **log** and **commit**. It is apparently not as good at merging as Git, Mercurial.

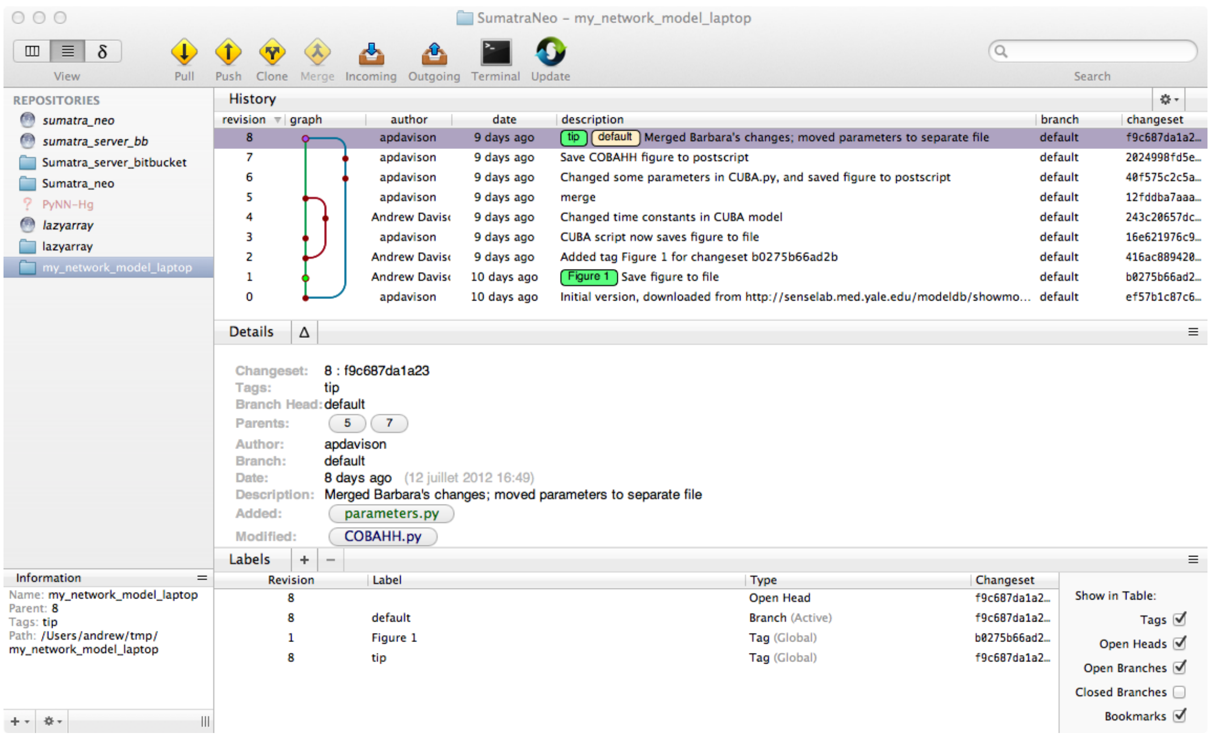
A few years ago, Subversion was by far the most popular open-source version control system, but it is now losing ground to distributed tools such as Git, Mercurial and Bazaar.

The following table shows the approximate equivalence between the most common Subversion and Mercurial commands.

svn checkout <url>	hg clone <url>
svn update	hg pull -u
svn commit	hg commit; hg push
svn log	hg log
svn status	hg status
svn info	hg summary
svn rm	hg rm

1.6.20 Graphical tools

As well as the command-line interface, graphical tools are available for all major version control systems. The following screenshot shows MacHg, a tool for working with Mercurial on Mac OS X: note the graph of branching, showing where the laptop repository, cluster repository and Barbara's repository branched off and then were merged back together.



revision	graph	author	date	description	branch	changeset
8		apdavison	9 days ago	tip default Merged Barbara's changes; moved parameters to separate file	default	f9c687da1a2…
7		apdavison	9 days ago	Save COBAHH figure to postscript	default	2024998fd5e…
6		apdavison	9 days ago	Changed some parameters in CUBA.py, and saved figure to postscript	default	40f575c2c5a…
5		apdavison	9 days ago	merge	default	12fddba7aaa…
4		Andrew Davis	9 days ago	Changed time constants in CUBA model	default	243c20657dc…
3		apdavison	9 days ago	CUBA script now saves figure to file	default	16e621976c9…
2		Andrew Davis	9 days ago	Added tag Figure 1 for changeset b0275b66ad2b	default	416ac889420…
1		Andrew Davis	10 days ago	Figure 1 Save figure to file	default	b0275b66ad2…
0		apdavison	10 days ago	Initial version, downloaded from http://senselab.med.yale.edu/modeldb/showmo…	default	ef57b1c87c6…

Changeset:	8 : f9c687da1a23
Tags:	tip
Branch Head:	default
Parents:	5 7
Author:	apdavison
Branch:	default
Date:	8 days ago (12 juillet 2012 16:49)
Description:	Merged Barbara's changes; moved parameters to separate file
Added:	parameters.py
Modified:	COBAHH.py

Name: my_network_model_laptop
Parent: 8
Tags: tip
Path: /Users/andrew/tmp/my_network_model_laptop

Revision	Label	Type	Changeset
8		Open Head	f9c687da1a2…
8	default	Branch (Active)	f9c687da1a2…
1	Figure 1	Tag (Global)	b0275b66ad2…
8	tip	Tag (Global)	f9c687da1a2…

Show in Table:
Tags ✓
Open Heads ✓
Open Branches ✓
Closed Branches ☐
Bookmarks ✓

The following Wikipedia entries may provide good starting points for investigating graphical version control clients:

- http://en.wikipedia.org/wiki/Comparison_of_Subversion_clients
- http://en.wikipedia.org/wiki/Mercurial
- http://en.wikipedia.org/wiki/Category:GIT_Tools

1.6.21 Web-based tools

There are many web-based services for hosting version control repositories, for example Google Code (http://code.google.com/), Sourceforge (http://sourceforge.net/), GitHub (https://github.com/) and BitBucket (https://bitbucket.org/). The following table shows which version control systems are supported by these four services, and whether they provide free private repositories (all support free public repositories):

Service	Subversion	Mercurial	Git	Bazaar	Free private repositories
Sourceforge	x	x	x	x	
Google Code	x	x	x		
BitBucket		x	x		x
GitHub			x		

1.7 Verification (testing)

1.7.1 How confident are you that your code is doing what you think it's doing?

When I began learning computational neuroscience and writing code for models and simulations, my programming experience amounted to no more than one course and one programming assignment using FORTRAN 77, as part of my undergraduate Physics degree and a short course on image processing and a little *ad hoc* data processing using Matlab during my MSc in Medical Physics.

When I wrote code (using Hoc and NMODL, the languages of the NEURON (http://www.neuron.yale.edu/) simulator) I tested it as I wrote it by running it and comparing the output to what I expected to see, to the results of previous simulations (one of my first projects was porting a model from GENESIS to NEURON, so I could quantitatively compare the output of the two simulators), and, later on, to experimental data.

If I might generalize from my own experience, from talking to colleagues, and from supervising student projects, this kind of informal, *the-results-look-about-right* testing is very widespread, especially for the physics- and biology-trained among us without any formal computer-science education.

1.7.2 Automated testing

Perhaps the biggest flaw of my informal testing regime was that none of the tests were automated. So, for example, I would test that the height of the excitatory-post-synaptic-potential was what I calculated it should be by plotting the membrane potential and reading the value off the graph. Each time I changed any code that could have affected this, I had to repeat this manual procedure, which of course discouraged any thought of making large-scale reorganisations of the code.

The advantages of creating automated tests are therefore:

- it gives you confidence that your code is doing what you think it is doing;
- it frees you to make wide-ranging changes to the code (for the purposes of optimization or making the code more robust, for example) without worrying that you will break something: if you do break something, your tests will tell you immediately and you can undo the change.

Of course, writing tests requires an initial time investment, but if you already perform manual, informal testing then this time will be paid back the first time you run the automated suite of tests. Even if you did no testing at all previously, the loss of fear of changing code will lead to more rapid progress.

There is one gotcha to be aware of with automated tests, a risk of false confidence which can lead to a lack of critical thinking about your results (*"if the tests pass, it must be alright"*). It is unlikely that your test suite will test every possible path through your code with all possible inputs, so you should always be aware of the fallibility of your test procedures, and should expect to add more tests as the project develops.

1.7.3 Terminology

Professional software engineering, where automated testing has been in wide use for a long time, has developed a rich vocabulary surrounding testing. For scientists, it is useful to understand at least the following three ideas:

unit test a test of a single element of a program (e.g. a single function) in isolation.

system test a test of an entire program, or an entire sub-system.

regression test a test for a specific bug that was found in a previous version of the program; the test is to ensure that the bug does not reappear once fixed. Regression tests may be unit tests or system tests.

Generally, you should write unit tests for every function and class in your program. Think of these like simple controls in an experiment. There should in general be multiple unit tests per function, to test the full range of expected inputs. For each argument, you should test at least:

- one or more typical values;
- an invalid value, to check that the function raises an exception or returns an error code;
- an empty value (where the argument is a list, array, vector or other container datatype).

It is not always easy to isolate an individual function or class. One option is to create "mock" or "fake" objects or functions for the function under test to interact with. For example, if testing a function that uses numbers from a random number generator, you can create a fake RNG that always produces a known sequence of values, and pass that as the argument instead of the real RNG.

Even though all unit tests pass, it may be that the units do not work properly together, and therefore you should write a number of system tests to exercise the entire program, or an entire sub-system.

On finding a bug in your program, don't leap immediately to try to fix it. Rather:

- find a simple example which demonstrates the bug;
- turn that example into a regression test (unit or system, as appropriate);
- check that the test fails with the current version of the code;
- now fix the bug;
- check that the regression test passes;
- check that all the other tests still pass.

1.7.4 Test frameworks

For a typical computational neuroscience project, you will probably end up with several hundred tests. You should run these, and check they all pass, before every commit to your version control system. This means that running all the tests should be a one-line command.

If you are familiar with the **make** utility, you could write a Makefile, so that:

```
$ make test
```

runs all your tests, and tells you at the end which ones have failed.

Most programming languages provide frameworks to make writing and running tests easier, for example:

Python unittest (http://docs.python.org/library/unittest.html), nose (http://nose.readthedocs.org/en/latest/), doctest (http://docs.python.org/library/doctest.html)

Matlab xUnit (http://www.mathworks.com/matlabcentral/fileexchange/22846-matlab-xunit-test-framework), mlUnit (http://sourceforge.net/projects/mlunit/), MUnit (http://www.mathworks.com/matlabcentral/fileexchange/11306-munit-a-unit-testing-framework-in-matlab), doctest (http://www.mathworks.com/matlabcentral/fileexchange/28862-doctest-embed-testable-examples-in-your-functions-help-comments)

C++ CppUnit (http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page), and many more

Java Junit (http://www.junit.org/), and many more

Also see http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

1.7.5 Examples

Here is an example of some unit tests for PyNN, a Python API for neuronal network simulation. PyNN provides a module `random` which provides wrappers for a variety of random number generators, so as to give them all the same interface so that they can be used more-or-less interchangeably.

```python
import pyNN.random as random
import numpy
import unittest

class SimpleTests(unittest.TestCase):
    """Simple tests on a single RNG function."""

    def setUp(self):
        random.mpi_rank=0; random.num_processes=1
        self.rnglist = [random.NumpyRNG(seed=987)]
        if random.have_gsl:
            self.rnglist.append(random.GSLRNG(seed=654))

    def testNextOne(self):
        """Calling next() with no arguments or with n=1 should return a float."""
        for rng in self.rnglist:
            assert isinstance(rng.next(), float)
            assert isinstance(rng.next(1), float)
            assert isinstance(rng.next(n=1), float)

    def testNextTwoPlus(self):
        """Calling next(n=m) where m > 1 should return an array."""
        for rng in self.rnglist:
            self.assertEqual(len(rng.next(5)), 5)
            self.assertEqual(len(rng.next(n=5)), 5)

    def testNonPositiveN(self):
        """Calling next(m) where m < 0 should raise a ValueError."""
        for rng in self.rnglist:
            self.assertRaises(ValueError, rng.next, -1)

    def testNZero(self):
        """Calling next(0) should return an empty array."""
        for rng in self.rnglist:
            self.assertEqual(len(rng.next(0)), 0)
```

We define a subclass of `TestCase` which contains several methods, each of which tests the `next()` method of a random number generator object. The `setUp()` method is called before *each* test method - it provides a place to put code that is common to all tests. Note that each test contains one or more assertions about the expected behaviour of `next()`.

Now, here is an example of a regression test (since it tests a particular bug that was found and fixed, to ensure the bug doesn't reappear later) that is also a system test (as it tests many interacting parts of the code, not a single code unit).

```
from nose.tools import assert_equal, assert_almost_equal
import pyNN.neuron

def test_ticket168():
    """
    Error setting firing rate of `SpikeSourcePoisson` after `reset()` in NEURON
    http://neuralensemble.org/trac/PyNN/ticket/168
    """
    pynn = pyNN.neuron
    pynn.setup()
    cell = pynn.Population(1, cellclass=pynn.SpikeSourcePoisson, label="cell")
    cell[0].rate = 12
    pynn.run(10.)
    pynn.reset()
    cell[0].rate = 12
    pynn.run(10.)
    assert_almost_equal(pynn.get_current_time(), 10.0, places=11)
    assert_equal(cell[0]._cell.interval, 1000.0/12.0)
```

For this tests we used the nose (http://nose.readthedocs.org/en/latest/) framework rather than the unittest (http://docs.python.org/library/unittest.html) framework used in the previous example. This test runs a short simulation, and then, as with unittest (http://docs.python.org/library/unittest.html), we make assertions about what values we expect certain variables to have.

1.7.6 Test coverage measurement

How do you know when you've written enough tests? Tools are available for many languages that will track which lines of code get used when running the test suite (list of tools at http://en.wikipedia.org/wiki/Code_coverage).

For example, the following command runs the test suite for the PyNN package and produces a report in HTML, highlighting which lines of the code have not been covered by the test:

```
$ nosetests --with-coverage --cover-erase --cover-package=pyNN --cover-html
```

1.7.7 Test-driven development

As the name suggests, test-driven development (TDD) involves writing tests *before* writing the code to be tested.

It involves an iterative style of development, repeatedly following this sequence:

- write a test for a feature you plan to add
- the test should *fail*, since you haven't implemented the feature yet
- now implement just that feature, and no more
- the test should now pass
- now run the entire test suite and check *all* the tests still pass
- clean up code as necessary (use the test suite to check you don't break anything)
- repeat for the next feature

The advantages of TDD are:

- makes you think about requirements before writing code
- makes your code easier to test (written to be testable)
- ensures that tests for every feature will be written
- reduces the temptation to over-generalize ("as soon as the test passes, stop coding")

Of course, writing more tests takes time, but there is some evidence that the total development time is reduced by TDD due to its encouragement of better design and much less time spent debugging.

1.8 Provenance tracking

1.8.1 What is provenance?

The term comes originally from the art world, where it refers to the chronology of the ownership and/or location of a work of art.

Having detailed evidence of provenance can help to establish that a painting has not been altered or stolen, and is not a forgery. Wikipedia has a nice entry on the provenance of the Arnolfini Portrait by van Eyck.

The provenance (abridged) of the painting is as follows:

1434 painting dated by van Eyck;

before 1516 in possession of Don Diego de Guevara, a Spanish career courtier of the Habsburgs;

1516 portrait given to Margaret of Austria, Habsburg Regent of the Netherlands;

1530 inherited by Margaret's niece Mary of Hungary;

1558 inherited by Philip II of Spain;

1599 on display in the Alcazar Palace in Madrid;

1794 now in the Palacio Nuevo in Madrid;

1816 in London, probably plundered by a certain Colonel James Hay after the Battle of Vitoria (1813), from a coach loaded with easily portable artworks by King Joseph Bonaparte;

1841 the painting was included in a public exhibition;

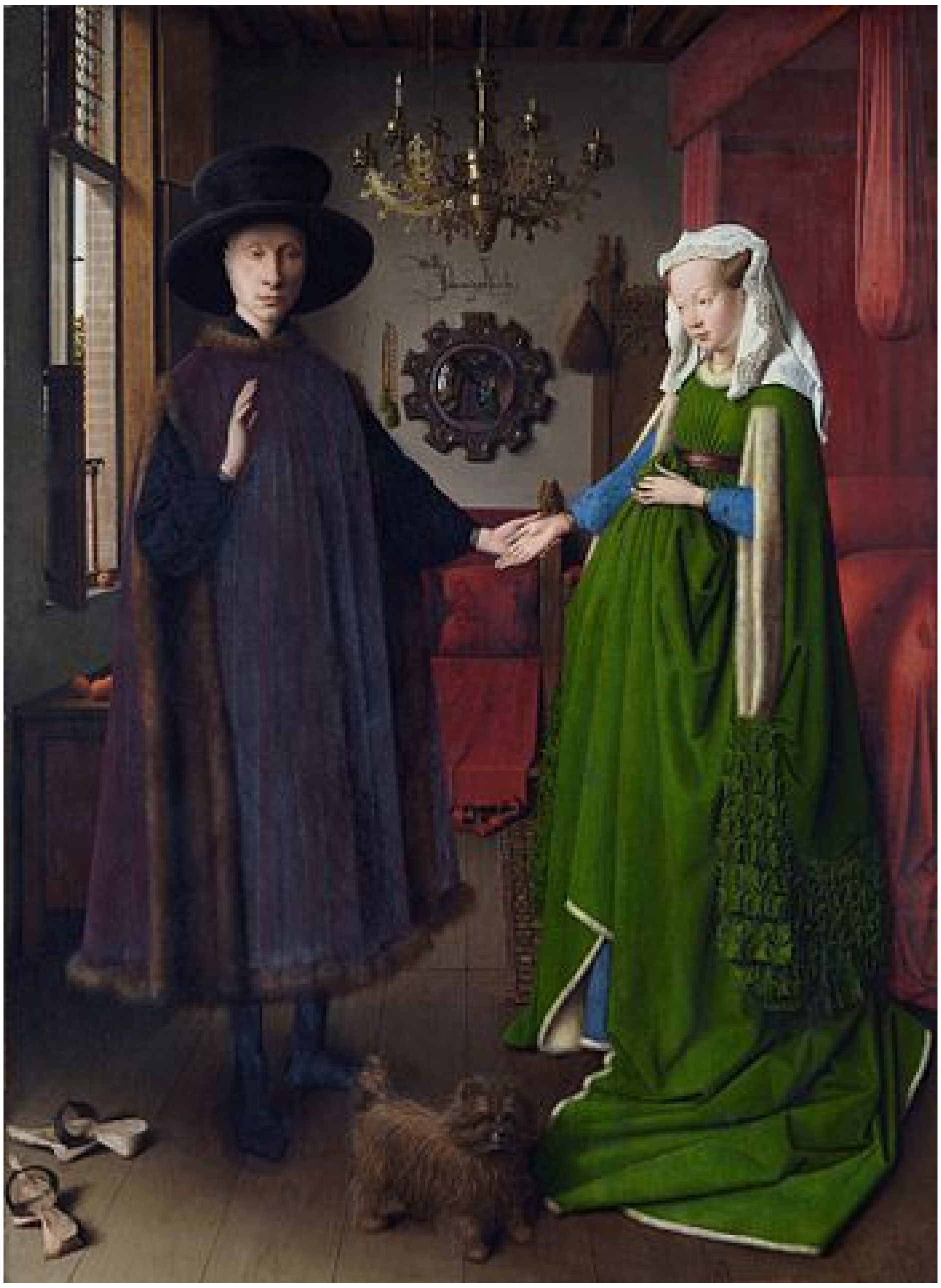


Figure 1.2: The Arnolfini Portrait, by Jan van Eyck

1842 bought by the National Gallery, London for £600, where it remains.

More recently, the term has been applied to other fields, including archaeology, palaeontology, and science more generally, where it refers to having knowledge of all the steps involved in producing a scientific result, such as a figure, from experiment design through acquisition of raw data, and all the subsequent steps of data selection, analysis and visualization. Such information is necessary for reproduction of a given result, and can serve to establish precedence (in case of patents, Nobel prizes, etc.)

1.8.2 The lab notebook

(http://www.flickr.com/photos/proteinbiochemist/3167660996/)

Figure 1.3: *"Lab bench" by proteinbiochemist on Flickr, CC BY-NC licence.*

The traditional tool for tracking provenance information in science is the laboratory notebook.

The problem with computational science is that the number of details that must be recorded is so large that writing them down by hand is incredibly tedious and error-prone.

What should be recorded for a single run of a computational neuroscience simulation?

- the code that was run:
  - the version of Matlab, Python, NEURON, NEST, etc.;
  - the compilation options;
  - a copy of the simulation script(s) (or the version number and repository URL, if using version control)
  - copies (or URLs + version numbers) of any external modules/packages/toolboxes that are imported/included
- how it was run:

    - parameters;
    - input data (filename plus cryptographic identifier to be sure the data hasn't been changed, later);
    - command-line options;
- the platform on which it was run:
    - operating system;
    - processor architecture;
    - network distribution, if using parallelization;
- output data produced (again, filename plus cryptographic identifier)
    - including log files, warnings, etc.

Even if you are *very* conscientious, this is a *lot* of information to record, and when you have a deadline (for a conference, or for resubmitting an article) there will be a strong temptation to cut corners - and those are the results where it is *most* important to have full provenance information.

Of course, some of these data are more important than others, but the less you record, the more likely you are to have problems replicating your results later on.

The obvious solution for computation-based science is to automate the process of provenance tracking, so that it requires minimal manual input, effectively to create an automated lab notebook.

In the general case, this is not a trivial task, due to the huge diversity in scientific workflows (command-line or GUI, interactive or batch-jobs, solo or collaborative) and in computing environments (from laptops to supercomputers).

We can make a list of requirements such a system should satisfy:

- automate as much as possible, prompt the user for the rest;
- support version control, ideally by interacting with existing systems (Git, Mercurial, Subversion, Bazaar ...);
- support serial, distributed, batch simulations/analyses;
- link to, and uniquely identify, data generated by the simulation/analysis;
- support all and any (command-line driven) simulation/analysis programs;
- support both local and networked storage of simulation/analysis records.

1.8.3 Software tools for provenance tracking

A wide variety of software tools has been developed to support reproducible research and provenance tracking in computational research. Each of these tools takes, in general, one of three approaches - literate programming, workflow management systems, or environment capture.

Literate programming

Literate programming, introduced by Donald Knuth in the 1980s, interweaves text in a natural language, typically a description of the program logic, with source code. The file can be processed in two ways: either to produce human-readable documentation (using LateX, HTML, etc.) or to extract, possibly reorder, and then either compile or iterpret the source code.

Closely related to literate programming is the "interactive notebook" approach used by Mathematica (http://www.wolfram.com/mathematica/), Sage (http://www.sagemath.org/), IPython (http://ipython.org/), etc., in which the output from each snippet of source code (be it text, a figure, a table, or whatever) is included at that point in the human-readable document.

This is obviously useful for scientific provenance tracking, since the code and the results are inextricably bound together. With most systems it is also possible for the system to automatically include information about software versions, hardware configuration and input data in the final document.

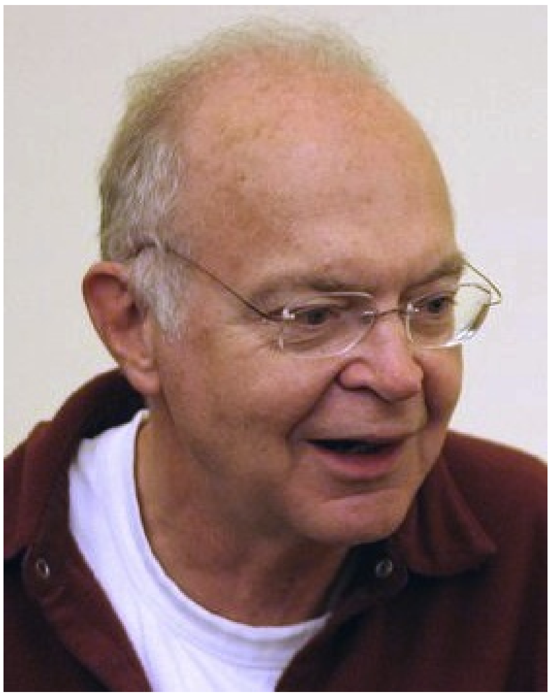


Figure 1.4: Donald Knuth

The literate programming/interactive notebook approach is less likely to be appropriate in the following scenarios:

- each individual step takes hours or days to compute;
- computations must be run on parallel and/or remote, time-sharing hardware;
- code is split among many modules or packages (i.e. the front-end, "user-facing" code that is included in the final documentation is only a small fraction of the total code).

This is not to say that the literate programming approach will not prove to be a good one in these scenarios (IPython includes good support for parallelism, for example, and many tools provide support for caching of results so only the code that has changed needs to be re-run), but the current generation of tools are generally more difficult to use in these scenarios.

The following are some literate programming/interactive notebook tools that are suitable for scientific use:

Sweave Sweave (http://www.statistik.lmu.de/ leisch/Sweave/) *"is an open-source tool that allows to embed the R code for complete data analyses in LaTeX documents. The purpose is to create dynamic reports, which can be updated automatically if data or analysis change."* Sweave is specific to the R-language, and is included in every R installation.

Emacs org mode Org-mode (http://orgmode.org/) is a mode for the open-source Emacs editor which enables tranforming plain text documents into multiple output formats: HTML, PDF, LaTeX, etc. The Babel extension to org-mode enables mixing of text and code (33 programming languages supported as of version 7.7). Delescluse et al. (2011) [1] present some examples of using both org-mode and Sweave for reproducible research.

TeXmacs TeXmacs (http://www.texmacs.org/) is an open-source *"wysiwyw (what you see is what you want) editing platform with special features for scientists"*, which allows mixing text, mathematics, graphics and interactive content. TeXmacs allows embedding and executing code in many languages, including Scheme, Matlab and Python.

IPython notebook IPython (http://ipython.org/) is an open-source environment for interactive and exploratory computing using the Python language. It has two main components: an enhanced interactive Python shell, and an architecture for interactive parallel computing. The IPython

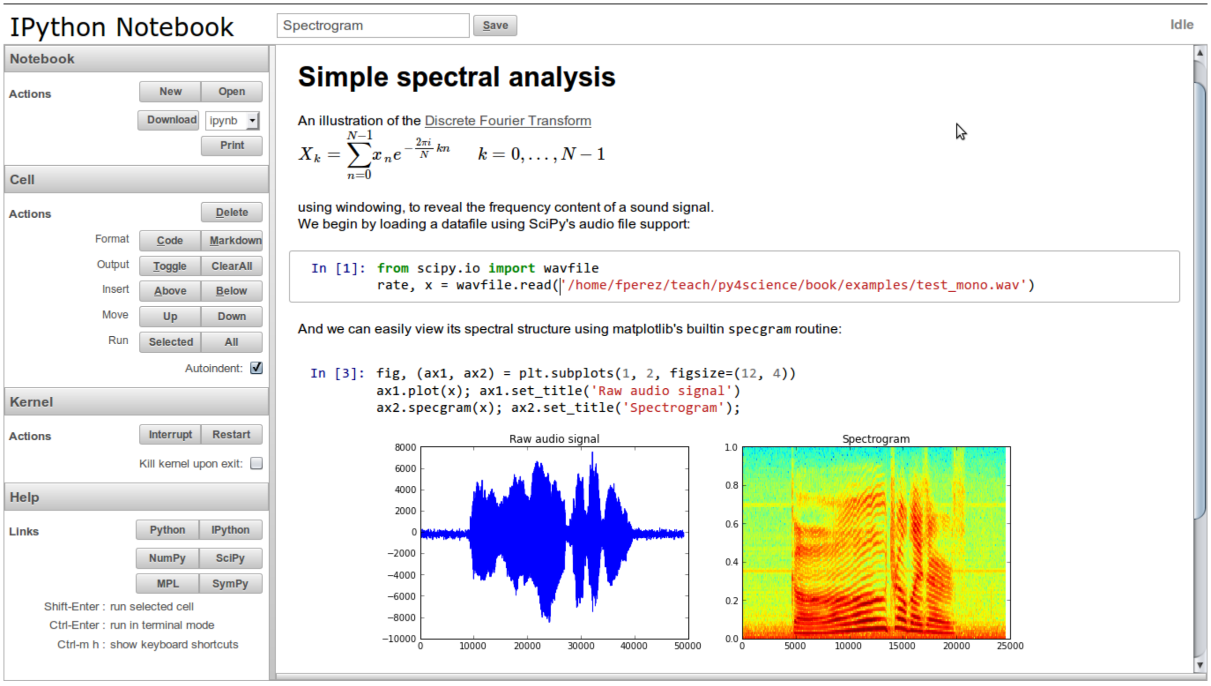
[1] Delescluse M., Franconville, R., Joucla, S., Lieurya, T. and Pouzat, C. (2011) Making neurophysiological data analysis reproducible. Why and how?, *J Physiol Paris* **106**:159-70.

notebook provides a web-browser-based user interface that allows mixing formatted text with Python code. The Python code can be executed from the browser and the results (images, HTML, LaTeX, movies, etc.) displayed inline in the browser.

Mathematica notebook Mathematica (http://www.wolfram.com/mathematica/) is proprietary software for symbolic and numerical mathematics, data analysis and visualization, developed by Wolfram Research. It has a notebook interface in which code and the results of executing the code are displayed together. It was part of the inspiration for the IPython notebook and the Sage mathematics system also mentioned here.

Sage notebook Sage (http://www.sagemath.org/) is a mathematics software system with a Python-based interface. It aims to create *"a viable free open source alternative to Magma, Maple, Mathematica and Matlab."* Like IPython, Sage notebooks allow creation of interactive notebooks that mix text, code and the outputs from running code.

Dexy Dexy (http://www.dexy.it/) is *"a free-form literate documentation tool for writing any kind of technical document incorporating code. Dexy helps you write correct documents, and to easily maintain them over time as your code changes."* Dexy supports multiple programming and text-markup languages. Dexy is slightly different to the other tools described here in that code lives in its own files (as in a normal, non-literate-programming approach) and parts (or all) of the code can then be selectively included in the document.



Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

using windowing, to reveal the frequency content of a sound signal.
We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('/home/fperez/teach/py4science/book/examples/test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

```
In [3]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

Figure 1.5: *An example of using the IPython notebook*

Workflow management systems

Where literate programming focuses on expressing computations through code, workflow management systems express computations in terms of higher-level components, each performing a small, well-defined task with well defined inputs and outputs, joined together in a pipeline.

Obviously, there is code underlying each component, but for the most part this is expected to be code written by someone else, the scientist uses the system mainly by connecting together pre-existing components, or by wrapping existing tools (command-line tools, webservices, etc.) so as to turn them into components.

Workflow management systems are popular in scientific domains where there is a certain level of standardization of data formats and analysis methods - for example in bioinformatics and any field that makes extensive use of image processing.

The advantages of workflow management systems are:

- reduces or eliminates the need for coding - pipelines can be built up using purely visual, graphical tools;
- helps with data format conversion;
- decoupling of the specification of a computation from its execution: this allows automated optimisation of pipeline execution, for example by computing different components in different environments, distributing components over multiple computing resources, caching previously-calculated results, etc.

The main disadvantage is that where there are no pre-existing components, nor easily-wrapped tools (command-line tools or webservices), for a given need, writing the code for a new component can be rather involved and require detailed knowledge of the workflow system architecture.

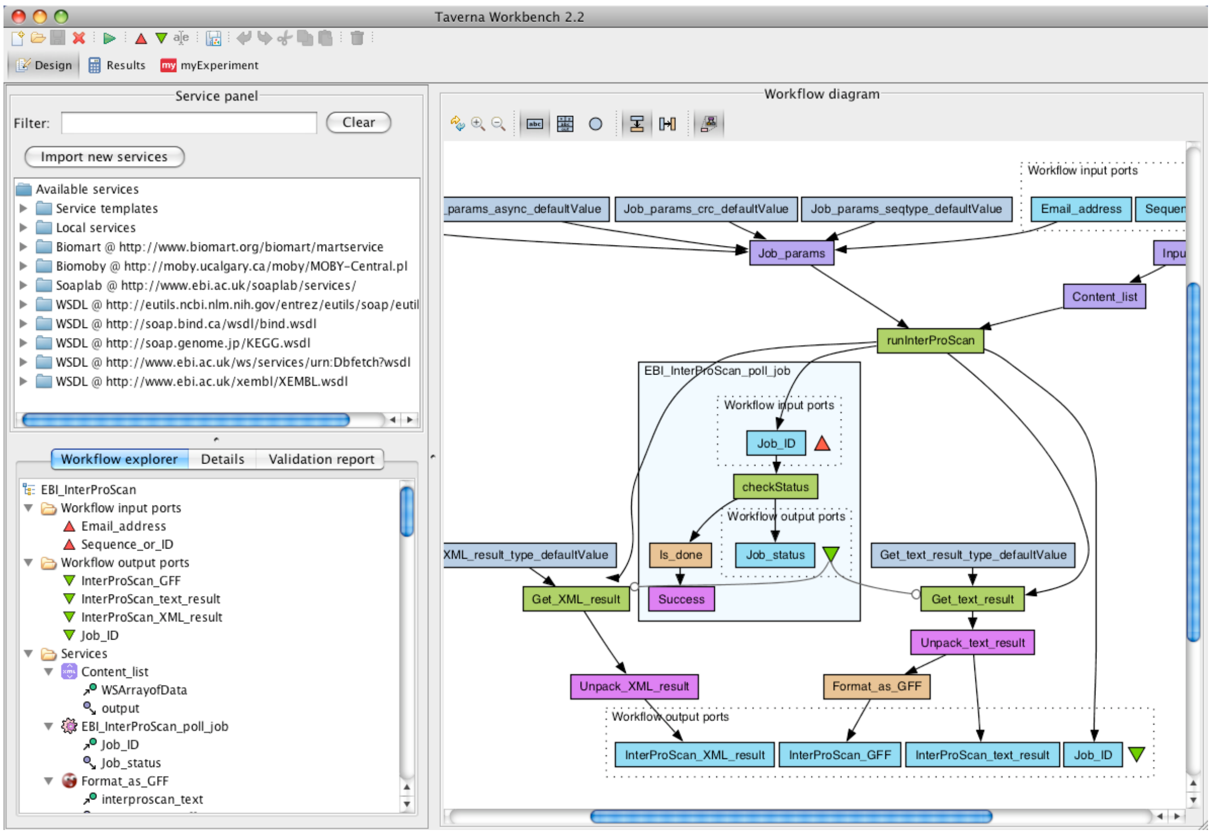


Figure 1.6: *The Taverna Workbench*

The following are some workflow management systems in wide use:

Taverna Taverna (http://www.taverna.org.uk/) is an open-source project mainly developed at the University of Manchester, UK, and related to the myGrid project. It is written in Java, and uses the OPM (Open Provenance Model) standard to provide provenance information: *"which services were executed, when, which inputs were used and what outputs were produced."* Taverna seems to be particularly widely used in the bioinformatics, cheminformatics and astronomy communities.

Kepler Kepler (https://kepler-project.org/) is an open-source project developed by a team centred on the University of California (UC Davis, UC Santa Barbara, UC San Diego). It is written in Java, and has support for components written in R and Matlab. Kepler seems to be particularly widely used in the environmental and earth sciences communities. Kepler has an optional provenance tracking module.

Galaxy Galaxy (http://galaxy.psu.edu/) *"is an open, web-based platform for data intensive biomedical research"*, developed mainly by groups at Penn State and Emory University. It appears to be focused specifically on bioinformatics. Galaxy seems to support provenance tracking through its "History" system.

VisTrails VisTrails (http://www.vistrails.org/) is an open-source project developed at the University of Utah, and written in Python. The distinctiveness of VisTrails is that it focuses on exploratory workflows: the entire history of a workflow is saved, so that it is easy to go back and forth between earlier and later versions of a workflow. Provenance information can be stored in a relational database or in XML files. VisTrails also supports a literate programming approach, providing a LaTeX package that allows links to workflows to be embedded in the document source. When the document is compiled, the workflow is executed in VisTrails and the output inserted into the document.

LONI Pipeline LONI Pipeline (http://pipeline.loni.ucla.edu/) is developed by the Laboratory of Neuro Imaging at UCLA. It is a general purpose workflow management system, but with a strong focus on neuroimaging applications. Recent versions of LONI Pipeline include a provenance manager for tracking data, workflow and execution history.

CARMEN The CARMEN (http://www.carmen.org.uk/portal) portal, developed by a consortium of UK universities, is a *"virtual research environment to support e-Neuroscience"*. It is *"designed to provide services for data and processing of that data in an easy to use environment within a web browser"*. Components can be written in Matlab, R, Python, Perl, Java and other languages. Provenance tracking does not appear to be automatic, but there is a system that *"allows the generation of metadata from services to provide provenance information"*.

Environment capture

The third approach to software tools for provenance tracking is in some ways the most lightweight, has the fewest restrictions on programming languages supported, and requires the least modification to computational scientists' existing workflows.

Environment capture means capturing all the details of the scientists' code, data and computing environment, in order to be able to reproduce a given computation at a later time.

The simplest approach is to capture the entire operating system in the form of a virtual machine (VM) image. When other scientists wish to replicate your results, you send them the VM image together with some instructions, and they can then load the image on their own computer, or run it in the cloud.

The VM image approach has the advantage of extreme simplicity, both in capturing the environment (all virtual machine software has a "snapshot" function which captures the system state exactly at one point in time) and in replaying the computation later.

The main disadvantages are:

- VM images are often very large files, of several GB in size;
- there is a risk that the results will be highly sensitive to the particular configuration of the VM, and will not be easily reproducible on different hardware or with different versions of libraries, i.e. the results may be highly replicable but not reproducible;
- it is not possible to index, search or analyse the provenance information;
- it requires the original computations to use virtualisation technologies, which inevitable have a performance penalty, even if small;
- the approach is challenging in a context of distributed computations spread over multiple machines.

An interesting tool which supports a more lightweight approach (in terms of filesize) than capturing an entire VM image, and which furthermore does not require use of virtualization technology, is CDE (http://www.pgbovine.net/cde.html). CDE stands for "Code, Data, Environment" and is developed by Philip Guo. CDE works only on Linux, but will work with any command-line launched programme. After installing CDE, prepend your usual command-line invocation with the **cde** command, e.g.:

```
$ cde nrngui init.hoc
```

CDE will run the programs as usual, but will also automatically detect all files (executables, libraries, data files, etc.) accessed during program execution and package them up in a compressed directory. This package can then be unpacked on any modern x86 Linux machine and the same commands run, using the versions of libraries and other files contained in the package, not those on the new system.

The advantages of using CDE are:

- more lightweight than the full VM approach, generates much smaller files
- doesn't have performance penalty of using VM
- minimal changes to existing workflow (use on your current computers)

The disadvantages are in general shared with the VM approach:

- the risk of results being highly sensitive to the particular configuration of your computer
- the difficulty in indexing, searching or analyzing the provenance information

In addition, CDE works only on modern versions of Linux.

An alternative to capturing the entire experiment context (code, data, environment) as a binary snapshot is to *capture all the information needed to recreate the context*. This approach is taken by Sumatra (http://neuralensemble.org/sumatra).

Note: Sumatra is developed by the author of this tutorial. I aim to be objective, but cannot guarantee this!

The advantages of this approach are:

- it is possible to index, search, analyse the provenance information;
- it allows testing whether changing the hardware/software configuration affects the results;
- it works fine for distributed, parallel computations;
- it requires minimal changes to existing workflows.

The main disadvantages, compared to the VM approach, are:

- a risk of not capturing all the context;
- doesn't offer "plug-and-play" replicability like VMs, CDE - the context must be reconstructed based on the captured metadata if you want to replicate the computation on another computer.

An interesting approach would be to combine Sumatra and CDE, so as to have both information about the code, data, libraries, etc. *and* copies of all the libraries in binary format.

1.8.4 An introduction to Sumatra

I will now give a more in-depth introduction to Sumatra. Sumatra is a Python package to enable systematic capture of the context of numerical simulations/analyses. It can be used directly in your own Python code or as the basis for interfaces that work with non-Python code.

Currently there is a command line interface **smt**, which is mainly for configuring your project and launching computations, and a web interface which is mainly for browsing and inspecting both the experiment outputs and the captured provenance information.

The intention is that in the future Sumatra could be integrated into existing GUI-based tools or new desktop/web-based GUIs written from scratch.

Installation

Installing Sumatra is easy if you already have Python and its associated tools:

Setting-up project tracking

Suppose you already have a simulation project, and are using Mercurial for version control. In your working directory (which contains your Mercurial working copy), use the **smt init** command to start tracking your project with Sumatra.

```
$ cd myproject
$ smt init MyProject
```

This creates a sub-directory, `.smt` which contains configuration information for Sumatra and a database which will contain simulation records (it is possible to specify a different location for the database, or to use an already existing one, but this is the default).

Capturing experimental context

Suppose that you usually run your simulation as follows:

```
$ python main.py default.param
```

To run it with automatic capture of the experiment context (code, data, environment):

```
$ smt run --executable=python --main=main.py default.param
```

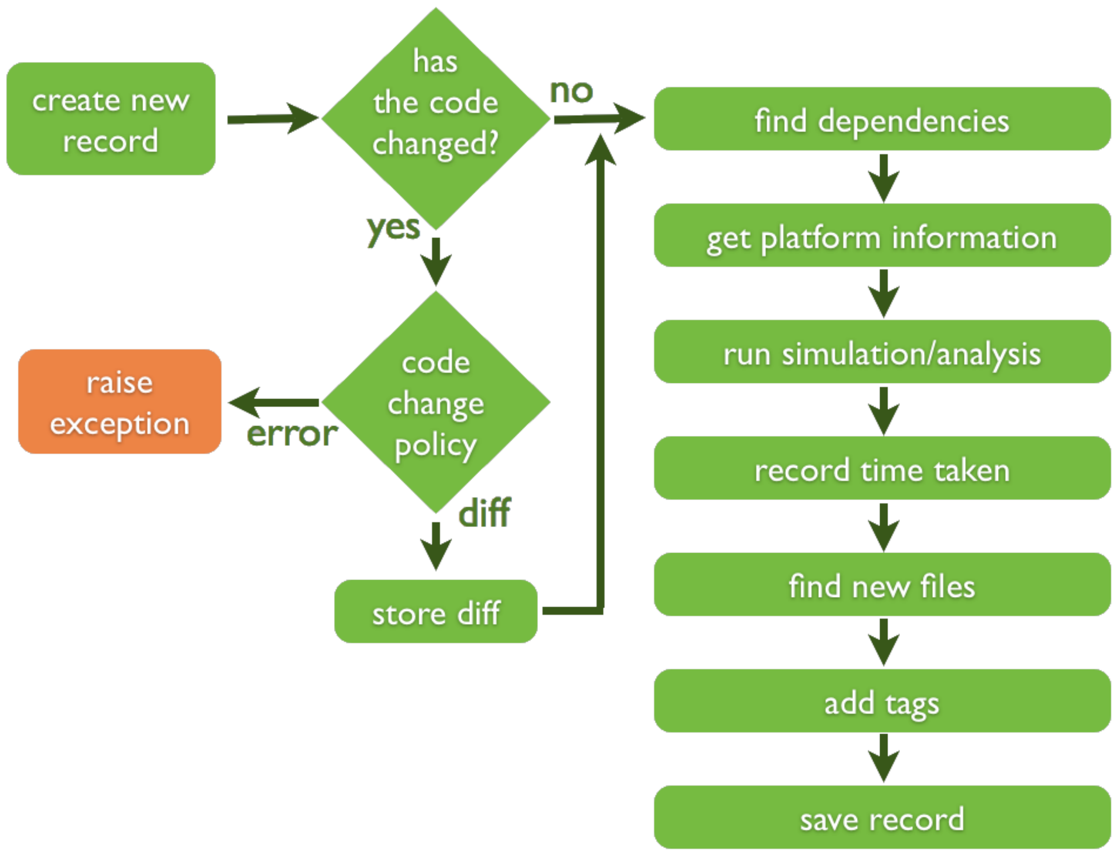
or, you can set defaults:

```
$ smt configure --executable=python --main=main.py
```

and then run simply:

```
$ smt run default.param
```

What happens when you do this is illustrated in the following figure:



Browsing the list of simulations

To see a list of simulations you have run, use the **smt list** command:

```
$ smt list
20130711-191915
20130711-191901
```

Each line of output is the *label* of a simulation record. Labels should be unique within a project. By default, Sumatra generates a label for you based on the timestamp, but it is also possible to specify your own label, as well as to add other information that might be useful later, such as the reason for performing this particular simulation:

```
$ smt run --label=haggling --reason="determine whether the gourd is worth 3 or 4 shekels" romans.
```

(see Monty Python's Life of Brian to understand the gourd reference).

After the simulation has finished, you can add further information, such as a qualitative assessment of the outcome, or a tag for later searching:

```
$ smt comment "apparently, it is worth NaN shekels."
$ smt tag "Figure 6"
```

This adds the comment and tag to the most recent simulation record. If you wish to annotate an older record, specify its label (this is why labels should be unique):

```
$ smt comment 20130711-191901 "Eureka! Nobel prize here we come."
```

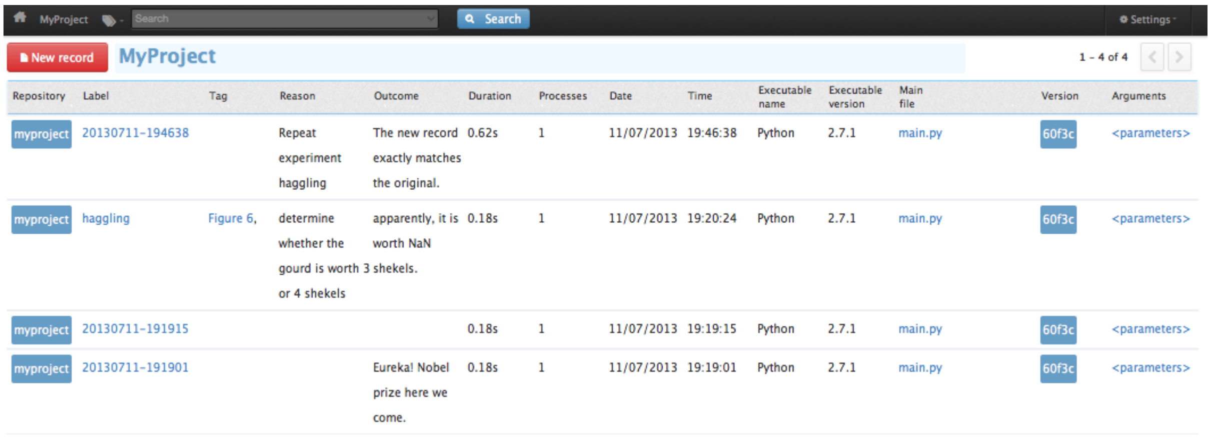
To get more information about your simulations, the **smt list** command takes various options:

```
$ smt list -l
--------------------------------------------------------------------------------
Label            : 20130711-191901
Timestamp        : 2013-07-11 19:19:01.678901
Reason           :
Outcome          : Eureka! Nobel prize here we come.
Duration         : 0.183968782425
Repository       : MercurialRepository at /path/to/myproject
Main_File        : main.py
Version          : 60f3cdd50431
Script_Arguments : <parameters>
Executable       : Python (version: 2.7.1) at /usr/bin/python
Parameters       : n = 100 # number of values to draw
                 : seed = 65785 # seed for random number generator
                 : distr = "uniform" # statistical distribution to draw values from
Input_Data       : []
Launch_Mode      : serial
Output_Data      : [example2.dat(43a47cb379df2a7008fdeb38c6172278d000fdc4)]
User             : Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
Tags             :
.
.
.
```

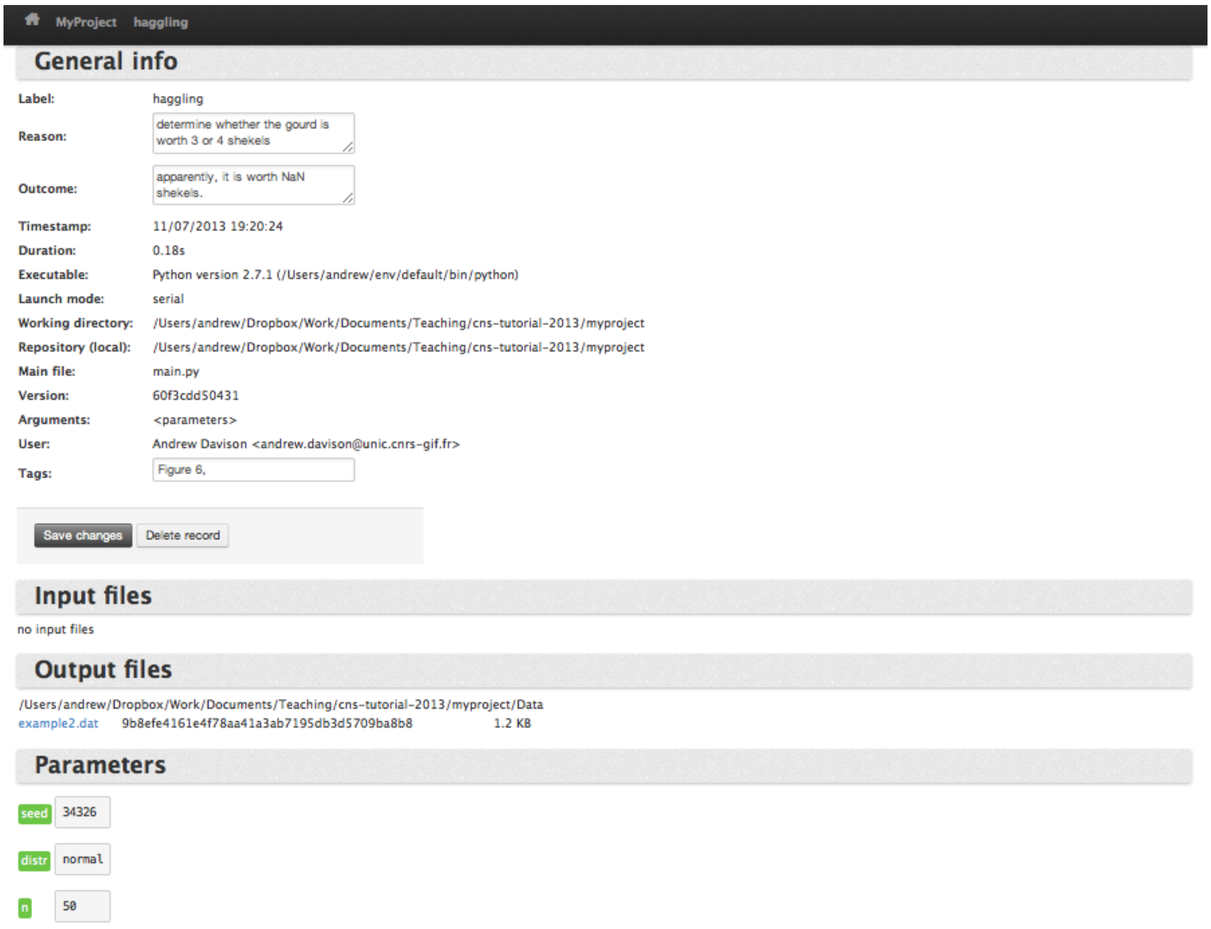
But in general, it is better to use the web interface to inspect this information. The web interface is launched with:

```
$ smtweb
```

This will run a small, local webserver on your machine and open a new browser tab letting you see the records in the Sumatra database:



Repository	Label	Tag	Reason	Outcome	Duration	Processes	Date	Time	Executable name	Executable version	Main file	Version	Arguments
myproject	20130711-194638		Repeat experiment haggling	The new record exactly matches the original.	0.62s	1	11/07/2013	19:46:38	Python	2.7.1	main.py	60f3c	<parameters>
myproject	haggling	Figure 6,	determine whether the gourd is worth 3 or 4 shekels	apparently, it is worth NaN shekels.	0.18s	1	11/07/2013	19:20:24	Python	2.7.1	main.py	60f3c	<parameters>
myproject	20130711-191915				0.18s	1	11/07/2013	19:19:15	Python	2.7.1	main.py	60f3c	<parameters>
myproject	20130711-191901			Eureka! Nobel prize here we come.	0.18s	1	11/07/2013	19:19:01	Python	2.7.1	main.py	60f3c	<parameters>



General info

Label:	haggling
Reason:	determine whether the gourd is worth 3 or 4 shekels
Outcome:	apparently, it is worth NaN shekels.
Timestamp:	11/07/2013 19:20:24
Duration:	0.18s
Executable:	Python version 2.7.1 (/Users/andrew/env/default/bin/python)
Launch mode:	serial
Working directory:	/Users/andrew/Dropbox/Work/Documents/Teaching/cns-tutorial-2013/myproject
Repository (local):	/Users/andrew/Dropbox/Work/Documents/Teaching/cns-tutorial-2013/myproject
Main file:	main.py
Version:	60f3cdd50431
Arguments:	<parameters>
User:	Andrew Davison <andrew.davison@unic.cnrs-gif.fr>
Tags:	Figure 6,

Input files

no input files

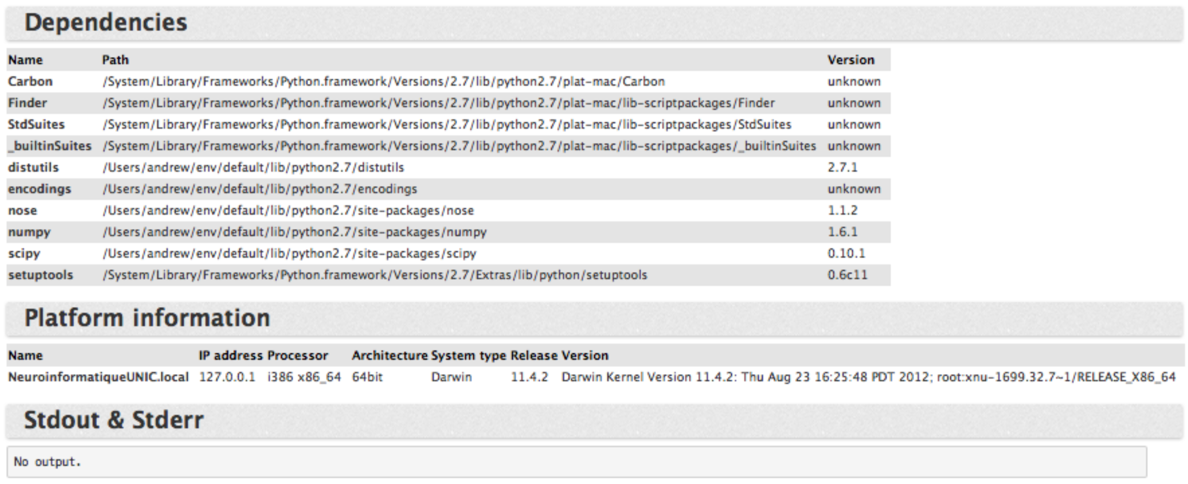
Output files

/Users/andrew/Dropbox/Work/Documents/Teaching/cns-tutorial-2013/myproject/Data

```
example2.dat    9b8efe4161e4f78aa41a3ab7195db3d5709ba8b8        1.2 KB
```

Parameters

seed	34326
distr	normal
n	50



Dependencies

Name	Path	Version
Carbon	/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/Carbon	unknown
Finder	/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages/Finder	unknown
StdSuites	/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages/StdSuites	unknown
_builtinSuites	/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages/_builtinSuites	unknown
distutils	/Users/andrew/env/default/lib/python2.7/distutils	2.7.1
encodings	/Users/andrew/env/default/lib/python2.7/encodings	unknown
nose	/Users/andrew/env/default/lib/python2.7/site-packages/nose	1.1.2
numpy	/Users/andrew/env/default/lib/python2.7/site-packages/numpy	1.6.1
scipy	/Users/andrew/env/default/lib/python2.7/site-packages/scipy	0.10.1
setuptools	/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/setuptools	0.6c11

Platform information

Name	IP address	Processor	Architecture	System type	Release	Version
NeuroinformatiqueUNIC.local	127.0.0.1	i386 x86_64	64bit	Darwin	11.4.2	Darwin Kernel Version 11.4.2: Thu Aug 23 16:25:48 PDT 2012; root:xnu-1699.32.7~1/RELEASE_X86_64

Stdout & Stderr

```
No output.
```

Replicating previous simulations

To re-run a previous simulation, use the **smt repeat** command:

```
$ smt repeat haggling
The new record exactly matches the original.
```

This command will use the version control system to checkout the version of the code that was used for the original simulation, run the simulation, and then compare the simulation outputs to the outputs from the original.

It does not attempt to match the rest of the environment (versions of libraries, processor architecture, etc.) and so this is a useful tool for checking the robustness of your results: if you have upgraded some library, do you still get the same results?

If the output data do not match, Sumatra will provide a detailed report of what is different between the two simulation runs.

Getting help

Sumatra has full documentation at http://packages.python.org/Sumatra/. The **smt** command also has its own built-in documentation. With no arguments it produces a list of available subcommands, while the **smt help** command can be used to get detailed help on each of the subcommands.

```
$ smt
Usage: smt <subcommand> [options] [args]

Simulation/analysis management tool version 0.6.0

Available subcommands:
  init
  configure
  info
  run
  list
  delete
  comment
  tag
  repeat
  diff
  help
  export
  upgrade
```

```
sync
migrate
```

Finding dependencies

Most of the metadata captured by **smt** is independent of the programming language used. Capturing the code dependencies (versions of software libaries, etc.) does however requires a per-language implementation.

Sumatra currently supports: Python, Matlab, Hoc (the language for the NEURON simulator) and the script language for the GENESIS 2 simulator. Support for Octave, R, C and C++ is planned.

Version finding is based on various heuristics:

- some language specific (e.g. in Python check for a variable called `__version__` or a function called `get_version()`, etc.)
- some generic (e.g. where dependency code is under version control or managed by a package manager)

Linking to input and output data

Part of the information Sumatra stores is the paths to input and output data. Currently, only data on the local filesystem is supported. In future, we plan to support data from relational databases, web-based databases, etc.) Sumatra stores the cryptographic signature of each data file to ensure file contents at a later date are the same as immediately after the simulation (this will catch overwriting of the file, etc.).

Record stores

Sumatra has multiple ways to store experiment records, to support both solo/local and collaborative/distributed projects:

- for local or network filesystems, there is a record store based on SQLite or PostgreSQL
- for collaboration over the internet, there is a server application providing a remote recordstore communicating over HTTP

Version control support

Sumatra requires your code to be under version control. Currently supported systems are Subversion, Git, Mercurial and Bazaar.

Summary

In summary, Sumatra (http://neuralensemble.org/sumatra) is a toolbox for automated context capture for computational experiments. Basic metadata is captured for any language, logging dependencies requires a language-specific plugin.

Using the **smt** command:

- requires no changes to existing code
- requires minimal changes to your existing workflow

thus, we hope, meeting our original criterion of:

> *"Be very easy to use, or only the very conscientious will use it"*

1.8.5 References

1.9 Conclusions and outlook

Perhaps you already follow all of the best practices mentioned in this document. If so, congratulations, and I'm sorry for wasting your time! Maybe you disagree with me about some of the suggestions. If so, please feel free to get in touch and we can discuss it (see below for contact details). Maybe you think some of the suggestions will be too much work, or take too much time. This is a fair point, and it is not easy to find the right balance between getting work done now and investing time in improving future productivity. In general, though, I think you will find that following good practices for data management and reproducible data analysis will pay off on a fairly short timescale: in a few weeks, at most a few months, you will reap the benefits of the time invested and be doing faster and better science. (Unfortunately there are no peer-reviewed, double-blind studies of the efficacy of most of these best practices; all I can say is that they work for me and for many others).

The sources for these notes are at https://bitbucket.org/apdavison/reproducible_research_cns/

Feel free to share, to modify and to reuse these notes, provided you give attribution to the author, and include a link to this web page (Creative Commons Attribution Licence (http://creativecommons.org/licenses/by/3.0/)).

Feedback would be very much appreciated. If you find an error or omission in these notes, or wish to suggest an improvement, you can create a ticket (https://bitbucket.org/apdavison/reproducible_research_cns/issues?status=new&status=open) or you can contact the author directly:

- by e-mail (davison@unic.cnrs-gif.fr (davison@unic.cnrs-gif.fr))
- via my website (http://andrewdavison.info)
- or as @apdavison on Twitter (https://twitter.com/apdavison).

Licence

CHAPTER 3

Sources

https://bitbucket.org/apdavison/reproducible_research_cns - feel free to fork the repository!