Sumatra Documentation

Release 0.4.0

Andrew P. Davison

July 03, 2015

Contents

Sumatra is a tool for managing and tracking projects based on numerical simulation and/or analysis, with the aim of supporting reproducible research. It can be thought of as an automated electronic lab notebook for computational projects.

It consists of:

- a command-line interface, *smt*, for launching simulations/analyses with automatic recording of information about the experiment, annotating these records, linking to data files, etc.
- a web interface with a built-in web-server, *smtweb*, for browsing and annotating simulation/analysis results.
- a Python API, on which smt and smtweb are based, that can be used in your own scripts in place of using smt, or could be integrated into a GUI-based application.

Sumatra is currently beta code, and should be used with caution and frequent backups of your records.

Table of Contents

1.1 Background

1.1.1 Reproducibility, provenance and project management

Reproducibility of experiments is one of the foundation stones of science. A related concept is provenance, being able to track a given scientific result, such as a figure in an article, back through all the analysis steps (verifying the correctness of each) to the original raw data, and the experimental protocol used to obtain it.

In computational, simulation- or numerical analysis-based science, reproduction of previous experiments, and establishing the provenance of results, ought to be easy, given that computers are deterministic, not suffering from the problems of inter-subject and trial-to-trial variability that make reproduction of biological experiments more challenging.

In general, however, it is not easy, perhaps due to the complexity of our code and our computing environments, and the difficulty of capturing every essential piece of information needed to reproduce a computational experiment using existing tools such as spreadsheets, version control systems and paper noteboooks.

1.1.2 What needs to be recorded?

To ensure reproducibility of a computational experiment we need to record:

- the code that was run
- any parameter files and command line options
- the platform on which the code was run

For an individual researcher trying to keep track of a research project with many hundreds or thousands of simulations and/or analyses, it is also useful to record the following:

- the reason for which the simulation/analysis was run
- a summary of the outcome of the simulation/analysis

Recording the code might mean storing a copy of the executable, or the source code (including that of any libraries used), the compiler used (including version) and the compilation procedure (e.g. the Makefile, etc.) For interpreted code, it might mean recording the version of the interpreter (and any options used in compiling it) as well as storing a copy of the main script, and of any external modules or packages that are included or imported into the script file.

For projects using version control, "storing a copy of the code" may be replaced with "recording the URL of the repository and the revision number".

The platform includes the processor architecture(s), the operating system(s), the number of processors (for distributed simulations), etc.

1.1.3 Tools for recording provenance information

The traditional way of recording the information necessary to reproduce an experiment is by noting down all details in a paper notebook, together with copies or print-outs of any results. More modern approaches may replace or augment the paper notebook with a spreadsheet or other hand-rolled database, but still with the feature that all relevant information is entered by hand.

In other areas of science, particularly in applied science laboratories with high-throughput, highly-standardised procedures, electronic lab notebooks and laboratory information management systems (LIMS) are in widespread use, but none of these tools seem to be well suited for tracking simulation experiments.

1.1.4 Challenges for tracking computational experiments

In developing a tool for tracking simulation experiments, something like an electronic lab notebook for computational science, there are a number of challenges:

- different researchers have very different ways of working and different workflows: command line, GUI, batch-jobs (e.g. in supercomputer environments), or any combination of these for different components (simulation, analysis, graphing, etc.) and phases of a project.
- some projects are essentially solo endeavours, others collaborative projects, possibly distributed geographically.
- as much as possible should be recorded automatically. If it is left to the researcher to record critical details there is a risk that some details will be missed or left out, particularly under pressure of deadlines.

The solution we propose is to develop a core library, implemented as a Python package, `sumatra`, and then to develop a series of interfaces that build on top of this: a command-line interface, a web interface, a graphical interface. Each of these interfaces will enable:

- launching simulations/analyses with automated recording of provenance information;
- managing a computational project: browsing, viewing, deleting simulations/analyses.

Alternatively, modellers can use the `sumatra` package directly in their own code, to enable provenance recording, then simply launch experiments in their usual way.

The core `sumatra` package needs to:

- interact with version control systems, such as Subversion, Git, Mercurial, or Bazaar;
- support launching serial, distributed (via MPI) or batch computations;
- link to data generated by the computation, whether stored in files or databases;
- support all and any command-line drivable simulation or analysis programs;
- support both local and networked storage of information;
- be extensible, so that components can easily be added for new version control systems, etc.
- be very easy to use, otherwise it will only be used by the very conscientious.

1.1.5 Further resources

For further background, see the following article:

> Davison A.P. (2012) Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science and Engineering* **14**: 48-56 [preprint]

You may also be interested in exploring this interactive poster about Sumatra: or in watching a talk given at a workshop on "*Reproducible Research-Tools and Strategies for Scientific Computing*" in Vancouver, Canada in July 2011. [video with slides (Silverlight required)] [video only (YouTube)] [slides].

1.2 Installation

To run Sumatra you will need Python installed on your machine. If you are running Linux or OS X, you almost certainly already have it. If you don't have Python, you can install it from python.org, or install one of the "value-added" distributions aimed at scientific users of Python: Enthought or Python(x,y).

The easiest way to install Sumatra is directly from the Python Package Index (PyPI):

```
$ pip install sumatra
```

or:

```
$ easy_install sumatra
```

Alternatively, you can download the Sumatra package from either PyPI or the INCF Software Centre and install it as follows:

```
$ tar xzf Sumatra-0.4.0.tar.gz
$ cd Sumatra-0.4.0
# python setup.py install
```

The last step may need to be run as root, or using sudo.

1.2.1 Installing Django

If you wish to use the web interface, you will also need to install Django. On Linux, you may be able to do this via your package management system: see http://code.djangoproject.com/wiki/Distributions.

Otherwise, it is very easy to install manually: see http://docs.djangoproject.com/en/dev/topics/install/#installing-official-release

You will also need to install the django-tagging and docutils packages, which may be in your package management system, otherwise they can be installed from PyPI:

```
$ pip install django-tagging
$ pip install docutils
```

1.2.2 Installing Python bindings for your version control system

Sumatra currently supports Mercurial, Subversion, Git and Bazaar. If you are using Subversion, you will need to install the pysvn bindings. Since Mercurial and Bazaar are mostly written in Python, just installing the main Mercurial/Bazaar packages is sufficient. For Git, you need to install the GitPython package.

1.3 Getting started

Let us assume that you already have a project based on numerical simulation, which you wish to start managing using Sumatra, and that the code for this project is under version control. Note that the following is equally valid

if your project is based on data analysis rather than, or as well as, simulation: just mentally replace "simulation" with "analysis" in the following.

Change to the working directory for your project, and then create a new Sumatra project in this directory using the `smt init` command:

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

where `MyProject` is the project name. This creates a sub-directory named `.smt`.

Sumatra tracks data files created by your simulation by searching for newly created files within a given directory tree. By default, it assumes that your simulation will create files in a sub-directory `Data` of your working directory. (You can change this by providing the `--datapath` option to `smt init` or `smt configure`.)

Now let's run a simulation. We will assume that your simulation code is written in Python, and that you run the simulation by executing a file called `main.py`, passing it the name of a parameter file on the command line, i.e., you would normally run a simulation using:

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

To run it using Sumatra, you would use:

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

Now we can see a list of the simulations we have run:

```
$ smt list
20121017-114820
```

This shows the label for each simulation we have run. Since we did not specify a label, one was automatically generated from the timestamp. To see more detail, use the `--long` option:

```
$ smt list --long
------------------------------------------------------------------------------
Label             : 20121017-114820
Timestamp         : 2012-10-17 11:48:20.421631
Reason            :
Outcome           :
Duration          : 0.119576931
Repository        : GitRepository at /path/to/myproject
Main_File         : main.py
Version           : 560a4afae1565799a29ca259b6a400aa389e59dd
Script_Arguments  : <parameters>
Executable        : Python (version: 2.7.1) at /usr/local/bin/python
Parameters        : seed = 45245
                  : distr = "uniform"
                  : n = 100
                  : tau_m = 20.0
Input_Data        : []
Launch_Mode       : serial
Output_Data       : [output.dat(dcb86788c2c793804a04e683fae99ad0bac8fb99)]
Tags              :
```

(most options also have a short form, `-l` in this case.)

It is a bit tedious to have to tell Sumatra which simulator and which file to run every time. Presumably, the name of the main file changes infrequently and the simulator almost never. Therefore, these can be set as defaults for a given project:

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

(you could also have given these options to `smt init`. `init` is used to create a project and `configure` to change its configuration later, but they mostly accept the same arguments).

Now you can run a simulation with a much shorter command line:

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

To see the current configuration of your project, use the `info` command:

```
$ smt info
Project name        : MyProject
Default executable  : Python (version: 2.7.1) at /usr/local/bin/python
Default repository  : GitRepository at /path/to/my/project
Default main file   : main.py
Default launch mode : serial
Data store (output) : ./Data
.          (input)  : /
Record store        : Relational database record store using the Django ORM (database file=/path/t
Code change policy  : error
Append label to     : None
```

Sumatra automatically records the identity and versions of the simulation files and the simulator executable, stores links to any files created by the simulation, records any error messages, the date and time at which the simulation was run, and its duration. You may also add your own annotations, in several different ways. On running the simulation, you can specify a unique label, and the reason for which you are running the simulation:

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

After the simulation is complete, you can add a description of the outcome:

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

This adds the comment to the most recent simulation. You may also describe the outcome of an earlier simulation, by specifying its label:

```
$ smt comment 20121017-114820 "Eureka! Nobel prize here we come."
```

You can also tag a simulation record with one or more short keywords:

```
$ smt tag foobar
$ smt tag barfoo
```

and remove tags:

```
$ smt tag --remove barfoo
```

The parameter file may be in any format - it is your script which is responsible for reading it. However, if it is in one of the formats that Sumatra understands then it is possible to modify parameter values on the command line. Suppose `default.param` contains a parameter `tau_m = 20.0`, as well as a number of other parameters, then:

```
$ smt run --reason="test effect of a smaller time constant" default.param tau_m=10.0
```

will generate a new parameter file identical to `default.param` but with `tau_m` equal to 10.0, and then will pass this new parameter file to your script. This can be very convenient when you wish to study the effects of changing one or two parameters, without having to edit your parameter file each time.

One of the main aims of Sumatra is to ensure the reproducibility of simulation results. The `repeat` command re-runs a previous simulation, and checks that the output is identical to that of the original run:

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

Although it is better not to delete simulation records (so as to preserve a full record of the project, false starts and all), it is possible:

```
$ smt delete 20121017-123706
```

It is also possible to delete all simulations with a given tag:

```
$ smt delete --tag foobar
```

Most of the commands described here have further options that we have not described. A full description of the options for each command is given in the `command reference`. The full list of commands is available by running `smt` by itself:

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

Simulation/analysis management tool version 0.4.0.dev

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

and help on a given command is available by running the command with the `--help` option, e.g.:

```
$ smt comment --help
Usage: smt comment [options] [LABEL] [COMMENT]

This command is used to describe the outcome of the simulation/analysis. If
LABEL is omitted, the comment will be added to the most recent experiment.
If the '-f/--file' option is set, COMMENT should be the name of a file
containing the comment, otherwise it should be a string of text.

Options:
  -h, --help     show this help message and exit
  -r, --replace  if this flag is set, any existing comment will be
                 overwritten, otherwise, the new comment will be appended to
                 the end, starting on a new line
  -f, --file     interpret COMMENT as the path to a file containing the
                 comment
```

or `smt help CMD`, where `CMD` is the name of the command.

This tutorial has covered using `smt` for serial simulations/analyses. A further tutorial covers using smt for parallel computations (using MPI).

Also see smtweb, which provides a more graphical interface to viewing lists of records than `smt list`.

1.4 Using the web interface

The web interface is built using the Django web framework, and requires that Django be installed (see Installation).

1.4.1 Starting the web interface

Before using the web interface, you must have created a Sumatra project using `smt init`.

To launch the web interface, in your project directory run:

```
$ smtweb &
```

This will launch a simple web server that listens on port 8000, and will automatically open a new tab in your browser at http://127.0.0.1:8000/. You can specify the `-n` option which will disable automatic opening of the new tab:

```
$ smtweb -n
```

If port 8000 is already in use, you can specify a different port with the `-p` option to `smtweb`, e.g.:

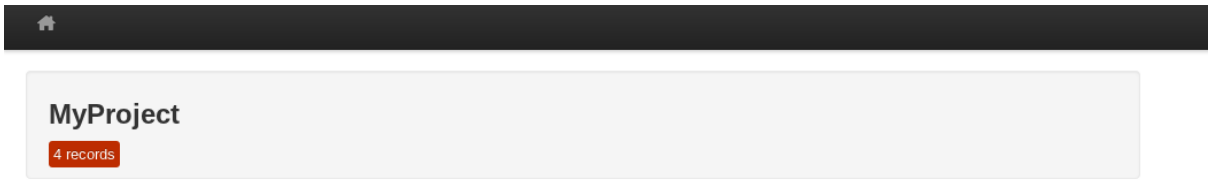
```
$ smtweb -p 8001
```

If you are using a single record store for multiple projects, you can run **`smtweb`** from anywhere and specify the location of the record store on the command line, e.g.:

```
$ smtweb ~/sumatra.db
```

1.4.2 List of projects

When you first start **`smtweb`**, the first page you see is a list of your projects.



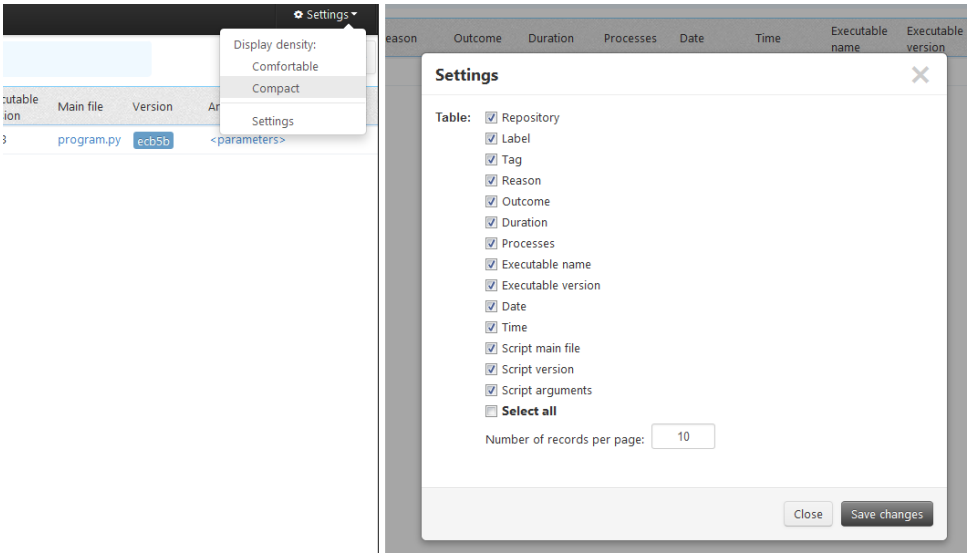
Click on the project name to see the records of your simulations/analyses in that project.

1.4.3 List of records

The list of records page contains a table with the following columns:

- version control repository
- label
- tags
- reason
- outcome
- duration
- number of processes
- date
- time
- executable name
- executable version
- main file
- version
- command line arguments

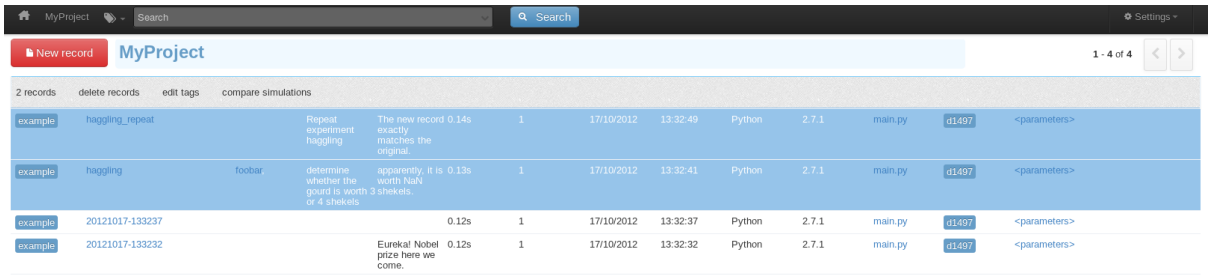
You can change which columns to display by clicking on **Settings**.



Selecting records

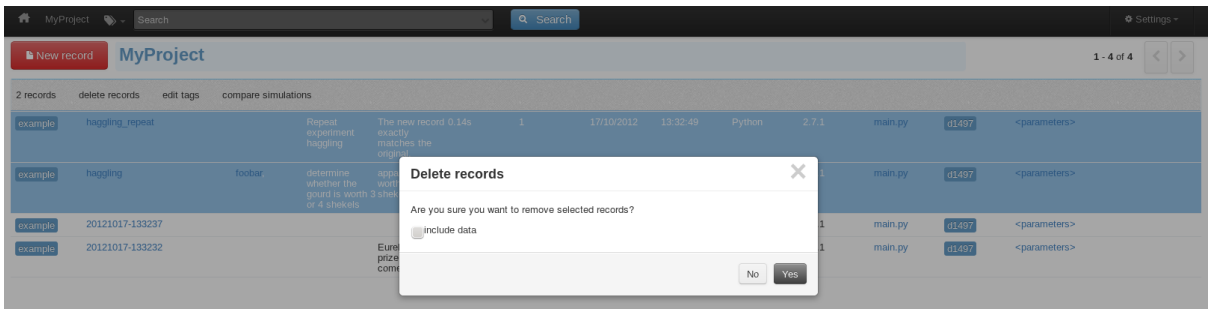
Each record is represented as one row in the table. The rows can be selected by dragging the mouse over them. As soon as you start doing that, the header of the table will be changed to contain actions you can perform with selected records. Actions you can perform are:

- set tags for the selected records
- delete records
- compare records



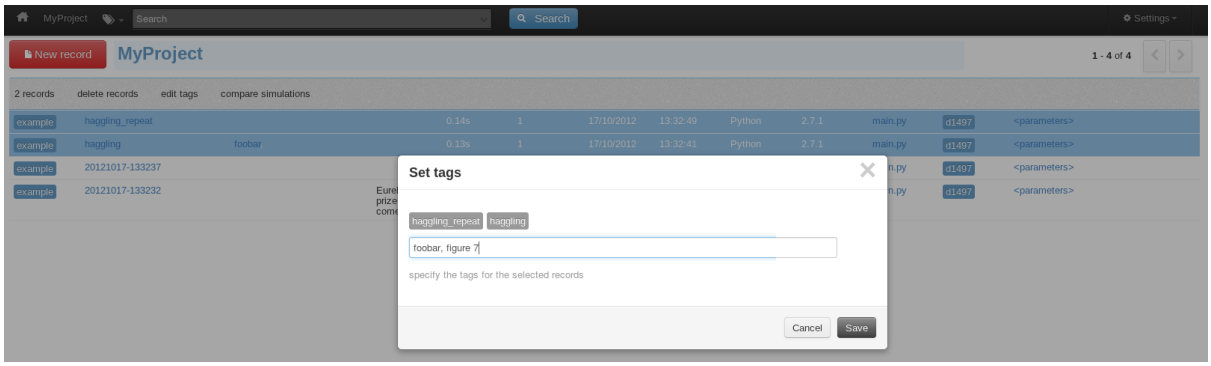
Deleting records

When deleting records, you have the option of also deleting any data generated by that simulation or analysis.



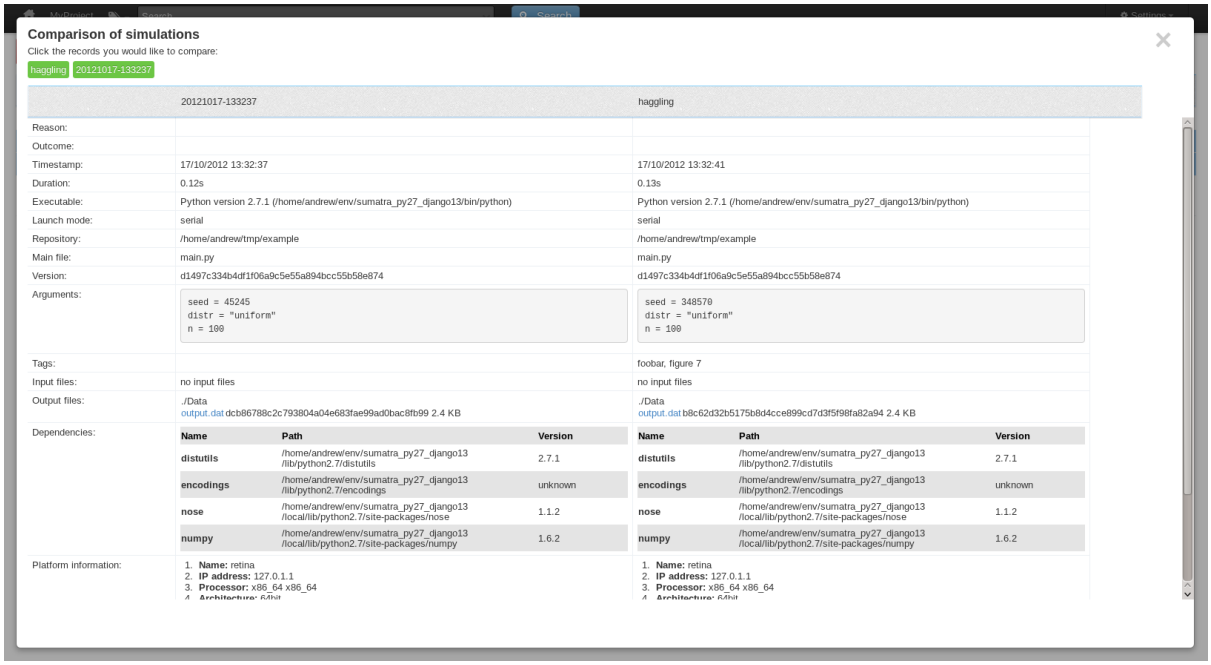
Editing tags

For the selected records you can specify additional tags, edit or remove them.



Comparing records

From the set of selected records you can choose any two to compare them.



Reviewing your code

You can see the contents of your main script file by clicking the corresponding link in the table. It will be shown in the modal window which can be dragged around.

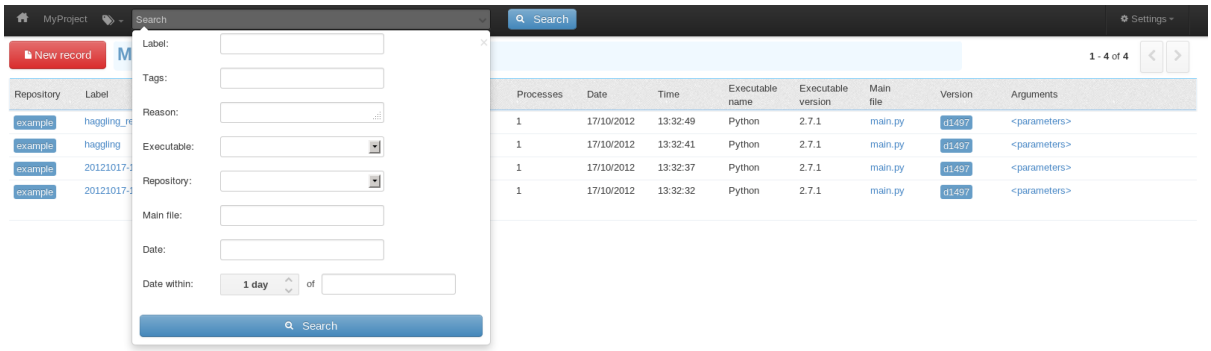
Important: for now this works only for Git and Mercurial repositories.

Filtering the records

You can filter the records by clicking on the 'tag' button or by using the search form. This form contains the following items:

- label

- tags
- reason
- executable
- repository
- main file
- date
- interval of dates



Search by **variable name** allows filtering the records using the name of parameter. If for each new simulation you have different parameter set, this feature can be useful for narrowing the set of possible records.

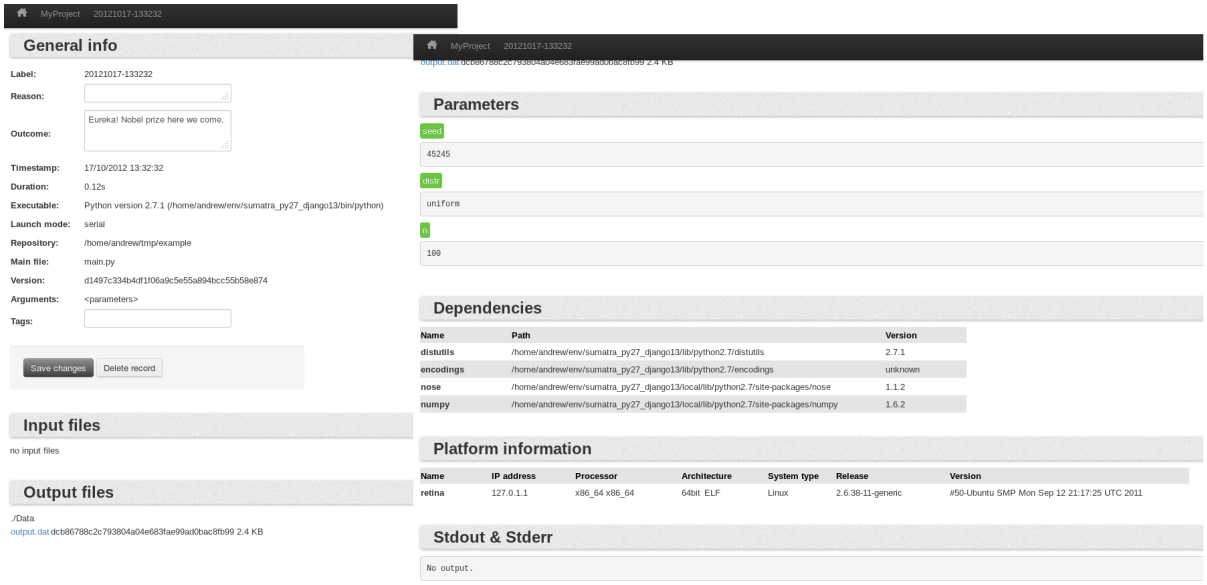
1.4.4 Accessing record details

You can access the record detail by clicking the corresponding label name in the main table. The record detail page contains the following sections:

- general info
- input files
- output files
- parameters
- dependencies
- platform information
- stdout & stderr

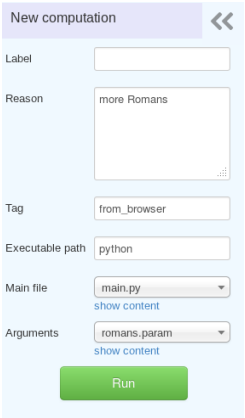
1.4.5 Finishing up

Don't forget to kill the webserver process (e.g. with `fg`, `Ctrl-C`) when you are finished with it.



1.4.6 Launching computations from the web interface

It is possible to run simulations/analyses from within the web interface. Clicking on the "New record" button will bring up the following dialog:

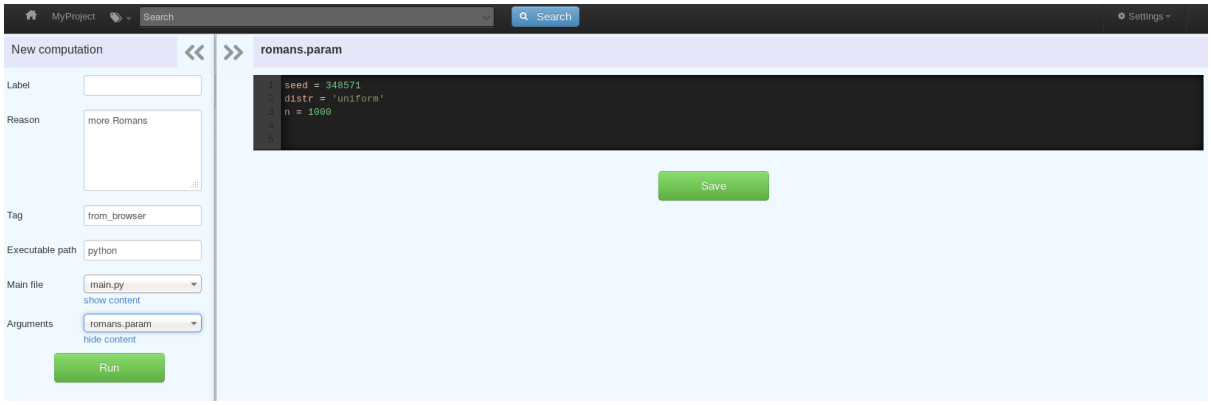


You can specify **label**, **reason**, **tag**, **main file**, **arguments**, and **executable** there. **Main file** and **arguments** are drop-down lists with the names of the files from the folder of the Sumatra project. As soon as the file is picked from the list, you can see its content. Moreover, the **argument** file is editable and any changes you made can be saved from this web page. You can hide and open the content of the files by clicking the corresponding links below the drop-down list.

On clicking run, the computation is launched and a progress bar appears. On succesful completion, a new record is added to the **list of records** page.

Customizing the web interface

You can customize the web interface on a per-project basis by placing your own Django templates in a "templates" subdirectory of the Sumatra ".smt" directory. The templates you can customize are called "base.html", "record_list.html", "show_file.html", "project_detail.html", "show_csv.html", "show_image.html", "record_detail.html", "show_diff.html", "tag_list.html". The best way to proceed is to copy the default template from "/path/to/sumatra/web/templates" to "/path/to/myproject/.smt/templates" and then modify it.



1.5 Graphical user interfaces

Sumatra has a command line interface and a web interface, but does not currently have a graphical desktop client.

We will probably write one at some point, but we hope that other people will also write their own, building on top of the tools and functionality provided in the Sumatra package.

The reason we hope for multiple desktop clients is that everyone has their own preferred workflow, and it seems unlikely that one graphical interface will work equally well for everyone.

1.6 Parallel simulations

As well as launching computations on your local machine, Sumatra can launch distributed, MPI-based computations on a cluster, at least for simple use-cases. We assume you already have your hosts files, etc. set up. Then, to run your simulation on 17 nodes, run:

```
$ smt run -n 17 default.param
```

(assuming you have already configured your default executable and main script file). This will call `mpiexec` for you with the appropriate arguments.

If this is insufficiently configurable for you, please take a look at the `DistributedLaunchMode` class in `launch.py` within the source distribution, and get in touch with the Sumatra developers, for example by creating a ticket or asking a question on the mailing list.

1.7 Using the Sumatra API within your own scripts

Using the `smt run` command is quick and convenient, but it does require you to change the way you launch your simulations/analyses.

One way to avoid this, if you use Python, is to use the `sumatra` package within your own scripts to perform the record-keeping tasks performed by `smt run`.

You may also wish to write your own custom script for creating a Sumatra project, instead of using `smt init`, but we do not cover this scenario here.

We will start with a simple example script, a dummy simulation, that reads a parameter file, generates some random numbers, and writes some data to file:

```
import numpy
import sys

def main(parameters):
    numpy.random.seed(parameters["seed"])
```

```
    distr = getattr(numpy.random, parameters["distr"])
    data = distr(size=parameters["n"])
    output_file = "example.dat"
    numpy.savetxt(output_file, data)

parameter_file = sys.argv[1]
parameters = {}
execfile(parameter_file, parameters) # this way of reading parameters
                                     # is not necessarily recommended
main(parameters)
```

Let's suppose this script is in a file named `myscript.py`, and that we have a parameter file named `defaults.param`, which contains:

```
seed = 65784
distr = "uniform"
n = 100
```

Without Sumatra, we would normally run this script using something like:

```
$ python myscript.py defaults.param
```

To run the script using the `smt` command line tool, we would use:

```
$ smt run --reason="reason for running this simulation" defaults.param
```

(This assumes we have previously used `smt init` or `smt configure` to specify that our executable is `python` and our main file is `myscript.py`.)

To benefit from the functionality of Sumatra without having to use `smt run`, we have to integrate the steps performed by `smt run` into our script.

First, we have to load the Sumatra project:

```
from sumatra.projects import load_project
project = load_project()
```

We're going to want to record the simulation duration, so we import the standard Python `time` module and record the start time:

```
import time
start_time = time.time()
```

We need to slightly modify the procedure for reading parameters. Sumatra stores the parameters for later use in searching and comparison, so they need to be transformed into a form Sumatra can use. This is very simple, we just replace the `execfile()` call with a `build_parameters()` call:

```
from sumatra.parameters import build_parameters
parameters = build_parameters(parameter_file)
```

Now we create a new `Record` object, telling it that the script is the current file; this automatically registers information about the simulation environment:

```
record = project.new_record(parameters=parameters,
                            main_file=__file__,
                            reason="reason for running this simulation")
```

Now comes the main body of the simulation, which is unchanged except that we take the opportunity to give the output data file a more informative name by adding the record label to the parameter file:

```
output_file = "%s.dat" % parameters["sumatra_label"]
```

At the end of the simulation, we calculate the simulation duration and search for newly created files:

```
record.duration = time.time() - start_time
record.output_data = record.datastore.find_new_data(record.timestamp)
```

Now we add this simulation record to the project, and save the project:

```
project.add_record(record)
project.save()
```

Putting this all together:

```
import numpy
import sys
import time
from sumatra.projects import load_project
from sumatra.parameters import build_parameters

def main(parameters):
    numpy.random.seed(parameters["seed"])
    distr = getattr(numpy.random, parameters["distr"])
    data = distr(size=parameters["n"])
    output_file = "%s.dat" % parameters["sumatra_label"]
    numpy.savetxt(output_file, data)

parameter_file = sys.argv[1]
parameters = build_parameters(parameter_file)

project = load_project()
record = project.new_record(parameters=parameters,
                            main_file=__file__,
                            reason="reason for running this simulation")
parameters.update({"sumatra_label": record.label})
start_time = time.time()

main(parameters)

record.duration = time.time() - start_time
record.output_data = record.datastore.find_new_data(record.timestamp)
project.add_record(record)

project.save()
```

Now you can run the simulation in the original way:

```
python myscript.py defaults.param
```

and still have the simulation recorded in your Sumatra project. For such a simple script and simple run environment there is no advantage to doing it this way: `smt run` is much simpler. However, if you already have a fairly complex run environment, this provides a straightforward way to integrate Sumatra's functionality into your existing system.

You will have noticed that much of the Sumatra code you have to add is effectively boilerplate, which will be the same for all your scripts. To save time, and typing therefore, Sumatra provides a `@capture` decorator for your `main()` function:

```
import numpy
import sys
from sumatra.parameters import build_parameters
from sumatra.decorators import capture

@capture
def main(parameters):
    numpy.random.seed(parameters["seed"])
    distr = getattr(numpy.random, parameters["distr"])
    data = distr(size=parameters["n"])
```

```
    numpy.savetxt("%s.dat" % parameters["sumatra_label"], data)

parameter_file = sys.argv[1]
parameters = build_parameters(parameter_file)
main(parameters)
```

This is now hardly any longer than the original script.

1.8 Parameter files

There is no requirement to put parameters in a separate file to use Sumatra, nor is it required to use a particular parameter file format. However, if you do use one of the formats Sumatra supports then you will gain extra functionality: currently, the ability to modify/add parameters on the command line and to have Sumatra automatically add the record label to the parameter file; in future versions, the ability to search and filter your records based on parameters.

1.8.1 Supported formats

Simple

Each parameter on a separate line, in "name = value" format. Values may be numbers, strings or lists (denoted with square brackets, comma-separated). Comments are indicated by a leading "#". Example:

```
# Example parameter file
nx = 10    # } grid
ny = 12    # } size
inputs = [1e-3, 2e-3, 5e-3, 1e-2]
label = "default"
```

Config/ini-style

Traditional config file format, as parsed by the standard Python ConfigParser module. Note that this format does not distinguish numbers from string representations of those numbers, so all parameter values are treated as strings. This format allows one level of nesting: you can create sections within which you can define parameters. Comments are indicated by a leading "#". Example:

```
[sectionA]
    a: 2
    b: 3

[sectionB]
    c: hello
    d: world
```

See the ConfigParser docs for more details.

JSON

See http://www.json.org/.

NeuroTools format

NeuroTools parameter format is essentially the same as JSON, but with the addition of a `url()` function which allows sub-parameter sets to be included from other files. Example:

```
{
  "network": {
    "excitatory_cells": url("https://neuralensemble.org/svn/NeuroTools/trunk/doc/example.param")
    "inhibitory_cells": {
      "tau_m": 15.0,
      "cm": 0.75,
    },
  },
  "sim": {
    "tstop": 1000.0,
    "dt": 0.11,
  }
}
```

1.8.2 Adding new formats

If your parameter file format is not supported by Sumatra, there is no problem: Sumatra will treat your parameter file as any other input data file and pass it directly through to your simulation/analysis script.

However, it is fairly straightforward to add support for a new format. You will need to write a Python class according to the following skeleton:

```
class MyParameterSet(object):

    def __init__(self, initialiser):
        # initialiser could be either a filesystem path or a string containing
        # the contents of your parameter file, and should raise a SyntaxError
        # if it cannot make sense of the contents.

    def __getitem__(self, name):
        # return the parameter or sub-parameter set with the given name

    def __eq__(self, other):
        # must be implemented

    def __ne__(self, other):
        # must be implemented

    def as_dict(self):
        # return the parameter set as a Python dict containing only numerical
        # types, lists, or other dicts.

    def save(self, filename):
        # self-explanatory

    def pop(self, k, d=None):
        # same behaviour as Python dict

    def update(self, E, **F):
        # same behaviour as Python dict

    def pretty(self, expand_urls=False):
        # Return a string representation of the parameter set, suitable for
        # creating a new, identical parameter set.
        # expand_urls is present for compatibility with NTParameterSet, and need
        # not be used.
```

For this version of Sumatra, you will have to include this class within the file `parameters.py` of your Sumatra installation, or send it to the developers to include in the Sumatra repository (see Developers' guide), as well as editing the `build_parameters()` function within `parameters.py` so that it tries to use your class. In the next version of Sumatra, we plan to include a plugin system which will greatly simplify adding your own customizations.

1.9 Upgrading your projects

Since new versions of Sumatra extend its capabilities, and may change the way records are stored, when you install a new version of Sumatra you will need to upgrade your existing Sumatra projects to work with the new version.

In future, this will probably be done automatically, but while Sumatra is still rapidly evolving we are keeping it as a simple manual process to minimize the risk of data loss.

1.9.1 Export using the old version

Before installing the new version of Sumatra, you must export your project to a file.

For Sumatra 0.1-0.3

First, download export.py to your project directory, then run:

```
$ python export.py
```

This will export your project in JSON format to two files in the `.smt` directory: `records_export.json` and `project_export.json`.

You can now delete `export.py`

For Sumatra 0.4 and later

Run:

```
$ smt export
```

This will export your project in JSON format to two files in the `.smt` directory: `records_export.json` and `project_export.json`.

1.9.2 Install the new version and upgrade

Now you can install the new version, e.g. with:

```
$ pip install --upgrade sumatra
```

or:

```
$ easy_install -U sumatra
```

(or you can install from source, as explained in doc:installation‘).

Then run:

```
$ smt upgrade
```

The original `.smt` directory will be copied to a time-stamped directory, e.g. `.smt_backup_20110209132422`

1.10 Migrating records between record stores

Sumatra supports multiple back-ends for storing records, `ShelveRecordStore`, based on the `shelve` module from the Python standard library, `DjangoRecordStore`, based on a relational database accessed via the Django

ORM, and `HttpRecordStore`, which stores records on a remote database with communication based on JSON over HTTP.

Suppose you created a Sumatra project using the `ShelveRecordStore` since you didn't want to install Django, then decided you'd like to change to `DjangoRecordStore`. This is what the project looks like at the beginning:

```
$ smt info
Sumatra project
---------------
Name                : MyProject
Default executable  : Python (version: 2.5.2) at /usr/local/bin/python
Default repository  : MercurialRepository at /path/to/working/directory
Default main file   : main.py
Default launch mode : serial
Data store          : /path/to/data
Record store        : Record store using the shelve package (database file=.smt/records)
Code change policy  : store-diff
Append label to     : cmdline
$ smt list
20110309-141853
20110309-141849
```

First, rename the ".smt" directory, and then create a new project:

```
$ mv .smt .smt_orig
$ smt init MyProject --addlabel cmdline --executable=python --on-changed=store-diff --main=main.py
```

Now we synchronize the old and new databases:

```
$ smt sync .smt_orig/records
```

and just to check it worked:

```
$ smt info
Sumatra project
---------------
Name                : MyProject
Default executable  : Python (version: 2.5.2) at /usr/local/bin/python
Default repository  : MercurialRepository at /path/to/working/directory
Default main file   : main.py
Default launch mode : serial
Data store          : /path/to/data
Record store        : Relational database record store using the Django ORM (database file=.smt/re
Code change policy  : store-diff
Append label to     : cmdline
$ smt list
20110309-141853
20110309-141849
```

1.11 Developers' guide

These instructions are for developing on a Unix-like platform, e.g. Linux or Mac OS X, with the bash shell.

1.11.1 Requirements

- Python 2.6 and/or 2.7
- Django >= 1.2
- django-tagging >= 0.3
- nose >= 0.11.4

- if using Python 2.6, unittest2 >= 0.5.1
- Distribute >= 0.6.14
- (optional) mpi4py >= 1.2.2
- (optional) tox >= 0.9 (makes it easier to test with multiple Python versions)
- (optional) twill >= 0.9 (needed for testing web interface)
- (optional) coverage >= 3.3.1 (for measuring test coverage)

1.11.2 Getting the source code

We use the Mercurial version control system. To get a copy of the code:

```
$ cd /some/directory
$ hg clone https://neuralensemble.org/hg/sumatra sumatra_src
```

Now you need to make sure that the `sumatra` package is on your PYTHONPATH and that the `smt` and `smtweb` scripts are on your PATH. You can do this either by installing Sumatra:

```
$ cd sumatra_src
$ python setup.py install
```

(if you do this, you will have to re-run `setup.py install` any time you make changes to the code) *or* by creating symbolic links from somewhere on your PATH and PYTHONPATH, for example:

```
$ cd /some/directory
$ ln -s sumatra_src/src sumatra
$ export PYTHONPATH=/some/directory:${PYTHONPATH}
$ export PATH=/some/directory/sumatra_src/bin:${PATH}
```

To update to the latest version from the repository:

```
$ hg pull -u
```

1.11.3 Running the test suite

Before you make any changes, run the test suite to make sure all the tests pass on your system:

```
$ cd sumatra_src/test/unittests
$ nosetests
```

You will see some error messages, but don't worry - these are just tests of Sumatra's error handling. At the end, if you see "OK", then all the tests passed, otherwise it will report how many tests failed or produced errors.

```
$ cd ..
$ python smt_test.py
$ python smtweb_test.py
```

1.11.4 Writing tests

You should try to write automated tests for any new code that you add. If you have found a bug and want to fix it, first write a test that isolates the bug (and that therefore fails with the existing codebase). Then apply your fix and check that the test now passes.

To see how well the tests cover the code base, run:

```
$ nosetests --coverage --cover-package=sumatra --cover-erase
```

1.11.5 Committing your changes

Once you are happy with your changes, you can commit them to your local copy of the repository:

```
$ hg commit -m 'informative commit message'
```

If you have a NeuralEnsemble account, you can now push your changes to the central repository:

```
$ hg push https://neuralensemble.org/hg/sumatra
```

Otherwise, you can create a patch:

```
$ hg export tip > descriptive_name.patch
```

and attach it to a ticket in the issue tracker. If you have made more than one commit, determine the revision number of when you cloned or last updated from the central repository (using `hg log`), and then give a range of revisions to include in the patch:

```
$ hg export start-revision:tip > descriptive_name.patch
```

Before either pushing or creating a patch, run the test suite again to check that you have not introduced any new bugs.

1.11.6 Coding standards and style

All code should conform as much as possible to PEP 8, and should run with Python 2.6 and 2.7.

1.12 Frequently asked questions

1.12.1 Where does the name "Sumatra" come from?

It is based on the initial letters of "Simulation Management Tool". (Sumatra was originally conceived for tracking simulations, it was only later that I realized it could equally well be used for any command-line driven computation). Despite a certain geographical proximity, it has nothing to do with Java :-)

1.12.2 When I run more than one simulation at once, Sumatra mixes up the output files. How can I make it associate the right files with the right simulation?

When you run a simulation/analysis, Sumatra looks for any new files within your datastore root directory, and associates them with your computation. This means that if you launch a second computation before the first one has finished Sumatra can't distinguish which files were produced by which computation. The solution is to save the results for a given computation in a subdirectory whose name is a unique id, and for Sumatra to look only in this subdirectory for output files.

The easiest way to do this is to use the record label. First run:

```
$ smt configure --addlabel=cmdline
```

or:

```
$ smt configure --addlabel=parameters
```

Then Sumatra will add the record label (which is generated from the timestamp unless you use the '`--label`' option to `smt run`) to either the command line or the parameter file for your script. It is then up to your script to read this value and use it to name your output files accordingly. Here is an example for a Python script, using the `cmdline` option and "./Data" set as the datastore root:

```
import sys
import os.path
options = sys.argv[1:]
label = options[-1] # label is added to the end of the command line

# computations happen here, results stored in `output_data`

output_dir = os.path.join("Data", label)
with open(os.path.join(output_dir, "mydata.txt"), 'w') as fp:
    fp.write(output_data)
```

1.13 Getting support

If you have a question about, or problems with using Sumatra, please post a message on either the sumatra-users or neuralensemble Google Groups.

1.14 Release notes

1.14.1 Sumatra 0.4.0 release notes

October 18th 2012

Welcome to Sumatra 0.4.0!

Overview

The biggest change in Sumatra 0.4 is the redesign of the browser-based interface, launched with **smtweb**. Thanks to the Google Summer of Code program, Dmitry Samarkanov was able to spend his summer working on improving Sumatra, with the results being a much improved web interface, better support for running Sumatra on Windows, and better support for running Matlab scripts with Sumatra. Many thanks to Google and to the INCF as mentoring organisation. In addition to Dmitry's improvements, handling of input and output data files is much improved, and Sumatra now captures and stores standard output (stdout) and standard error (stderr) streams. More details on all of these, plus a bunch of minor improvements and bug fixes, is given below. Finally, Sumatra no longer supports Python 2.5 - the minimum requirement is Python 2.6.

Web interface

The Sumatra browser-based interface runs a local webserver on your computer, and allows you to browse the information that Sumatra captures about your analyses, simulations or other computations, including code versions, input and output data files, parameter/configuration files, the operating system and processor architecture.

The interface has been completely redesigned for Sumatra 0.4, and includes dozens of large and small improvements, including:

- a more modern, attractive design
- the ability to select which columns to display in the record list view
- the ability to search all of your records based on date, tags or full-text
- side-by-side comparison of records
- sorting of records based on any column
- selection of multiple records by clicking or dragging for deletion, comparison and tagging

Furthermore, it is now possible to launch computations from the browser interface.

For more information, see Using the web interface and this blog post from Dmitry Samarkanov.

Data file handling

In earlier versions of Sumatra, the filename (or rather, the file path relative to a user-defined root) was used as the identifier for input and output data files. The problem with this, of course, is that it is possible to overwrite a given file with new data. For this reason, Sumatra 0.4 now calculates and stores the SHA1 hash of the file contents. If the file contents change, the hash will also change, so that Sumatra can alert you if a file is accidentally overwritten, for example.

Sumatra 0.4 also adds a new data store which automatically archives a copy of the output data from your computations in a user-selected location. This data store is accessible through the API as the `ArchivingFileSystemDataStore` class, or through the **`smt`** command-line interface with the "–archive" option to the "init" and "configure" commands.

Finally, Sumatra now allows the user the choice of whether to use an absolute or relative path for the data store root directory. Using a relative path makes projects easier to move and easier to access from other locations (e.g. with symbolic links or NFS).

Matlab support

Sumatra can capture certain information for *any* command-line tool: input and output data, version of the main codebase, operating system and processor architecture, etc. For dependency information, however (i.e. which libraries, modules or packages are imported/included by your main script), a separate plugin is required for each language. Sumatra already has a dependency tracking plugin for Python and for two computational neuroscience simulation environments, NEURON and GENESIS. Sumatra 0.4 adds dependency tracking for Matlab scripts.

Recording of stdout and stderr

Sumatra 0.4 now supports recording and storage of the standard output and standard error streams from your scripts.

Other new features

- added support for JSON-format parameter files;
- added **`smt export`** command, which allows the contents of a Sumatra record store to be exported in JSON format;
- more information is now printed by **`smt list --long`**;
- the Python dependency finder now supports scripts run with Python 3 (although Sumatra itself still needs Python 2);
- can now specify `HttpRecordStore` username and password as part of the URL passed to `smt init`;
- added support for markup using reStructuredText in the project description
- it is no longer required to have a script file, which makes it possible to use Sumatra with your own compiled executables. Further support for compiled languages is planned for the next release.

Bug fixes

A whole bunch of bugs were fixed in Sumatra 0.4.

1.15 Authors and contributors

The following people have contributed code to Sumatra. The institutional affiliations are those at the time of the contribution, and may not be the current affiliation of a contributor.

- Andrew Davison [1]
- Dmitry Samarkanov [2]
- Bartosz Telenczuk [1, 3]
- Michele Mattioni [4]
- Eilif Muller [5]
- Konrad Hinsen [6]
- Stephan Gabler [7]
- Takafumi Arakaki [8]

1. Unité de Neuroscience, Information et Complexité, CNRS UPR 3293, Gif-sur-Yvette, France
2. Ecole Centrale de Lille, Lille, France
3. Institute for Theoretical Biology, Humboldt University zu Berlin, Berlin, Germany
4. European Bioinformatics Institute, Hinxton, UK
5. Blue Brain Project, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
6. Centre de biophysique moléculaire, CNRS UPR 4301, Orléans, France
7. Max Planck Institute for Human Development, Berlin, Germany
8. Laboratoire de Neurophysique et Physiologie, CNRS UMR 8119, Université Paris Descartes, Paris, France

If I've somehow missed you off the list I'm very sorry - please let us know.

Many thanks also go to everyone who has reported bugs on the issue tracker.

1.15.1 Licence

Sumatra is freely available under the CeCILL v2 license, which is equivalent to, and compatible with, the GNU GPL license, but conforms to French law (and is also perfectly suited to international projects) - see http://www.cecill.info/index.en.html for more information.

If you are interested in using Sumatra, but the choice of licence is a problem for you, please contact us - we are open to persuasion.