
Sumatra Documentation

Release 0.8dev

The Sumatra Project

Sep 11, 2017

Contents

1	Table of Contents	3
	Python Module Index	67

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 LaTeX package and Sphinx extension for including Sumatra-tracked figures and links to provenance information in papers and other documents.
- 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.

Table of Contents

Background

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 notebooks.

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.

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.

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.

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](#)]

For more detail on how Sumatra is implemented, see:

Davison A.P., Mattioni M., Samarkanov D. and Teleńczuk B. (2014) Sumatra: A Toolkit for Reproducible Research. In: *Implementing Reproducible Research*, edited by V. Stodden, F. Leisch and R.D. Peng, Chapman & Hall/CRC: Boca Raton, Florida., pp. 57-79. [PDF]

You might also be interested 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].

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](#), [Python\(x,y\)](#) or [Anaconda](#).

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

```
$ pip 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-0.7.0.tar.gz
$ cd Sumatra-0.7.0
# python setup.py install
```

The last step may need to be run as root, or using sudo, although in general we recommend installing in an isolated environment created using virtualenv or conda.

Installing Django

If you wish to use the web interface, you will also need to install [Django](#) version 1.6 or later. On Linux, you may be able to do this via your package management system: see <https://code.djangoproject.com/wiki/Distributions>.

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

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

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

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 Bazaar is mostly written in Python, just installing the main Bazaar package is sufficient. For Git, you need to install the [GitPython](#) package. For Mercurial, you need to install the [hgapi](#) package.

Command completion for bash

Sumatra comes with a limited bash completion facility. You can install it to you system by sourcing the file *smt-completion.sh* in your *.bashrc* or *.profile*. By default, Sumatra installs this script to your */usr/bin* directory by default but moving it elsewhere (e.g. to *~/bash_completion.d/*) is recommended.

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
20140418-154800
```

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           : 20140418-154800
Timestamp       : 2014-04-18 15:48:00.439809
Reason          :
Outcome         :
Duration        : 0.216287851334
Repository      : GitRepository at /path/to/myproject
Main_File       : main.py
Version         : a75cf131a69ba831e915bc6a09987e832e65e7bc
Script_Arguments : <parameters>
Executable      : Python (version: 2.6.8) at /usr/local/bin/python
Parameters      : seed = 65785 # seed for random number generator
                  : distr = "uniform" # statistical distribution to draw values from
                  : n = 100 # number of values to draw
Input_Data      : []
Launch_Mode     : serial
Output_Data     : [example2.dat (43a47cb379df2a7008fdeb38c6172278d000fdc4)]
User            : Arthur Dent <dent@example.com>
Tags            :
Repeats         : None
```

(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: 2.6.8) at /usr/local/bin/python
Default repository : GitRepository at /path/to/myproject
Default main file  : main.py
Default launch mode : serial
Data store (output) : /path/to/myproject/Data
.                 (input) : /
Record store       : Django (/path/to/myproject/.smt/records)
Code change policy : error
Append label to    : None
Label generator    : timestamp
Timestamp format   : %Y%m%d-%H%M%S
Sumatra version    : 0.7.0
```

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.param
```

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 20140418-154800 "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 20140418-154800
```

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 0.7.0

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

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. By default, comments
will be appended to any existing comments. To overwrite existing comments, use
the '-r/--replace flag.

positional arguments:
  LABEL                the record to which the comment will be added
  comment              a string of text, or the name of a file containing the
                      comment.

optional arguments:
```

```
-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`.

Managing a research project with Sumatra

Before reading this, we recommend reading *Getting started* for a quick introduction to Sumatra. This document expands on the information presented there, giving a fuller picture of the available options.

Setting up your project

To create a Sumatra project, run:

```
$ smt init ProjectName
```

in your working directory. **`smt init`** has many options (see *smt command reference* for a full list), but in general, **`smt configure`** has the same options, so any of the options can be changed later. To see the current configuration of your project, run **`smt info`**.

Telling Sumatra about your code

Sumatra expects to find a version control repository in your working directory or one of its parent directories. If you haven't yet cloned your repository, Sumatra will do it for you if you give the `--repository` argument:

```
$ smt init --repository=https://github.com/someuser/MyCode
```

If you usually run the same main program, you can store this as a default using the `--executable` option:

```
$ smt init --executable=python
```

Here you can give a full path, or just the name; in the latter case Sumatra will use the `PATH` environment variable to search for an executable with that name, and use the first one it finds.

For interpreted languages, it is also possible to set a default script file using the `--main` option:

```
$ smt init --main=myscript.py
```

In this case you must give the full path, either relative to the working directory or an absolute path.

Any time you run some code with Sumatra, it checks whether the code has changed (using your version control system). If it has, then by default Sumatra will refuse to run until you have committed your changes. This is so that the exact version of the code you run is always recorded. As an alternative, Sumatra can store the difference between the code in version control and the code you run. To enable this option, run:

```
$ smt init --on-changed=store-diff
```

You can always change back to the “strict” setting with:

```
$ smt configure --on-changed=error
```

Handling input and output data

Sumatra tracks both input and output data files. For output data files it is important to tell Sumatra in which directory the files will be created, to avoid it having to search the entire disk:

```
$ smt init --datapath=results
```

Now Sumatra will recursively search `results` and all sub-directories for any new files created during the computation. If your computations overlap in time there is a risk that Sumatra will mix up the files. A solution to this problem is given in the [Frequently asked questions](#). If you don't specify the output data directory, Sumatra will assume it is a directory called "Data".

The default directory for input data files is the filesystem root; this can also be changed with the `--input` option. Note that input data files are only tracked if they are passed to your program as command-line arguments. This limitation will be removed in future.

For more on data handling, see [Input and output data](#).

Storing Sumatra records

Sumatra supports multiple back-end databases for storing records. More information about how to choose a storage back-end is given in [Record stores](#).

Running your code

To track a computation with Sumatra, you can either use the `smt` tool or write your own Python scripts using the Sumatra API (see [Using the Sumatra API within your own scripts](#)).

A typical way to run a computation with `smt` is:

```
$ smt run --executable=matlab --main=myscript.m input_file1 input_file2
```

or:

```
$ smt run -e matlab -m myscript.m input_file1 input_file2
```

using the short versions of the arguments. Note that `input_file1` and `input_file2` may be parameter/configuration files or data files. If the former, they will be treated specially, see [Parameter files](#).

Note that if you are not using an interpreted language, only the `-executable` argument is needed. If you have set default values for the executable and/or main script in the program configuration, the `smt run` command can be simplified, e.g.:

```
$ smt configure -e matlab -m myscript.m
$ smt run input_file1 input_file2
```

Running different versions

If you want to run a previous version of your code, rather than the currently checked-out version, use the `--version` option:

```
$ smt run --version=3e6f02a
```

Note that this will not overwrite any uncommitted changes; rather Sumatra will refuse to run until you have committed, stashed, reverted, etc. your changes. Sumatra will also not return to the most recent version after the run: future runs with no version specified will continue to use the older version.

Labels

To identify your computation, a unique label is required. Sumatra can generate this for you automatically, or this can be specified using the `--label` option:

```
$ smt run --label=test0237
```

Two formats are available for automatically-generated labels, timestamp-based (the default), and uuid-based.

```
$ smt configure --labelgenerator=uuid $ smt configure --labelgenerator=timestamp --
timestamp_format=%Y%m%d-%H%M%S
```

Command-line options

If your own program has its own command-line options of the form `--option=value`, **smt run** will try to interpret these as Sumatra command-line parameters (options of the form `--option value`, without the equals sign, are fine). To avoid this, use the `--plain` configuration option:

```
$ smt configure --plain
```

(`--no-plain` turns this off).

Reading from stdin, writing to stdout

If your program reads from stdin and/or writes to stdout, i.e. you would normally run it using:

```
$ myprog < input.txt > output.txt
```

then you can tell Sumatra to run it the same way, but in addition to track the input/output file, using:

```
$ smt run -e myprog -i input.txt -o output.txt
```

Commenting

Sumatra offers two ways to attach comments to your computations. When you launch a computation, you can give the reason for running it, e.g. what hypothesis you are testing:

```
$ smt run --reason="Test the effect of using a low-pass filter"
```

Once the computation is finished, you can comment on the outcome:

```
$ smt comment "Doesn't seem to make much of a difference"
```

By default, the comment is attached to the most recent computation. You can also comment on an older record, by giving its label:

```
$ smt comment 20150423-235351 "Didn't work due to a bug"
```

You can comment multiple times on the same record. By default, the new comment will be appended to the old one. To overwrite the old comment, use the `--replace` flag. If you would like to attach a longer comment than will fit on one line, or a more structured comment, you can write your comment in a temporary text file and then attach that to the record:

```
$ smt comment --file comment.txt
```

Both the “reason” and “outcome” fields can be edited in the web browser interface. To add headings, sub-headings, hyperlinks, emphasis, etc. in a comment, you can use **reStructuredText** markup, which will be rendered as HTML.

Tagging

To structure your project, and make it easier to find the most interesting results, you can add tags to your records, either through the web browser interface or on the command line, e.g.:

```
$ smt tag "Figure 5" 20141203-093401
```

If you omit the record label, the most recent computation will be tagged.

Tags may contain spaces, but in this case must be contained in quotes. You can tag multiple records at the same time:

```
$ smt tag modelA 20141203-093401 20141203-122344 20150109-194344
```

You can also remove tags:

```
$ smt tag --remove modelA 20141203-122344
```

Viewing and searching results

The easiest way to review your project is to use the web browser interface - see [Using the web interface](#) for more information on this. It is also possible, however, to view computation records on the command line, or to export the information to other formats such as HTML and LaTeX.

```
$ smt list
```

lists the labels of all records. When you have a lot of records, it will probably be more useful to filter by tags:

```
$ smt list tag1 tag2 tag3
```

will only show records that have been tagged with *all* the tags in the list. To show fuller information about each record, use the “`--long/-l`” option:

```
$ smt list -l
```

The order of records can be reversed using the “`--reverse/-r`” flag.

By default, the output is formatted for the console. Several other output formats are also available, for example LaTeX:

```
$ smt list --long --format=latex > myproject.tex
$ pdflatex myproject
```

You can customize the LaTeX output by copying the default template from `sumatra/formatting/latex_template.tex` to the `.smt` subdirectory of your project, and then modifying it. The template uses the Jinja2 templating language.

Comparing records

The web browser interface allows side-by-side comparison of pairs of records. A more limited comparison is available on the command-line with `smt diff`.

Deleting records

If your last computation failed because of a silly bug:

```
$ smt delete
```

will remove it. Older records can be deleted by giving a list of labels:


```
$ smt delete 20141203-093401 20141203-122344
```

or a list of tags:

```
$ smt delete --tag tag1
```

If you want to also delete the data files generated by the computations, add the “`--data`” flag. Records can also be deleted through the web browser interface.

Relocating a project

If you need to move a Sumatra project to a new directory or a new computer, first copy all the files, ensuring that the `.smt` directory and its contents are also copied. We strongly suggest you also take a backup, and check carefully that everything is working correctly in the new location before deleting the original.

You will next need to update the project configuration to reflect the new location. Supposed you created your project with the default settings, so that your record store is in `.smt/records` and your output data is stored under the `Data` subdirectory, run the following in the new location:

```
$ smt configure --store .smt/records --repository . --datapath Data
```

Alternatively, you can manually edit `.smt/project`.

If you have also moved the data associated with your project, you will need to update the paths stored in the record store:

```
$ smt migrate --datapath Data
```

If you are using the archive or mirror options for your data (see *Input and output data*) you may also need to migrate those paths/URLs.

Using the web interface

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

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 have a large dataset and the webpage loads slowly, using the serverside processing increases the performance of the website. This method fetches only displayed records from the server instead of all records. You can specify the `-s` option to start the local server with serverside processing:

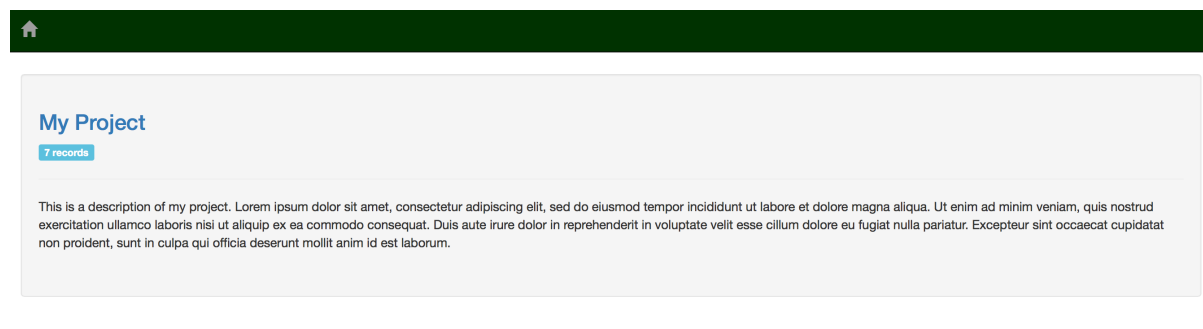
```
$ smtweb -s
```

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
```

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.

List of records

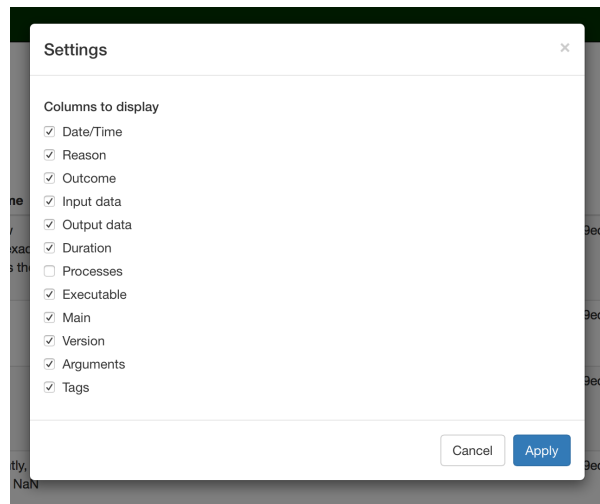
By default, your project is viewed from a process-centric point of view, i.e. one record for each computation performed. The list of records page contains a table with one row per computation, with the following columns:

- label
- date/time
- reason
- outcome
- input data
- output data
- duration
- number of processes
- executable name and version
- main file
- code version
- command line arguments
- tags

You can change which columns to display by clicking on the settings icon.

Selecting records

Each record is represented as one row in the table. The rows can be selected and unselected by clicking on them. Actions that can be performed on selected rows are:



- delete records
- compare records

MyProject
⚙

My Project

Record View
Data View
About

Delete selected
Compare selected

Search:

Label	Date/Time	Reason	Outcome	Input data	Output data	Duration	Executable	Main	Version	Tags
20150526-140238	26/05/2015 14:02:38	Changed filename			example.dat	0.06s	Python 2.7.9	main.py	9f99c53062cf49bb93e5ab0828d0ce897fcc372	
20150526-135246	26/05/2015 13:52:46	Repeat experiment haggling	The new record exactly matches the original.		example2.dat	0.07s	Python 2.7.9	main.py	bac1db1b9ec5c90ffe8aabe59e8a5aab5f56340f	
20150526-135236	26/05/2015 13:52:36	Use a different seed			example2.dat	0.06s	Python 2.7.9	main.py	bac1db1b9ec5c90ffe8aabe59e8a5aab5f56340f	
20150526-135206	26/05/2015 13:52:06	test effect of a smaller time constant			example2.dat	0.06s	Python 2.7.9	main.py	bac1db1b9ec5c90ffe8aabe59e8a5aab5f56340f	
haggling	26/05/2015 13:51:21	determine whether the gourd is worth 3 or 4 shekels	apparently, it is worth NaN shekels.		example2.dat	0.07s	Python 2.7.9	main.py	bac1db1b9ec5c90ffe8aabe59e8a5aab5f56340f	foobar
20150526-135104	26/05/2015 13:51:04		Eureka! Nobel prize here we come.		example2.dat	0.07s	Python 2.7.9	main.py	bac1db1b9ec5c90ffe8aabe59e8a5aab5f56340f	

Deleting records

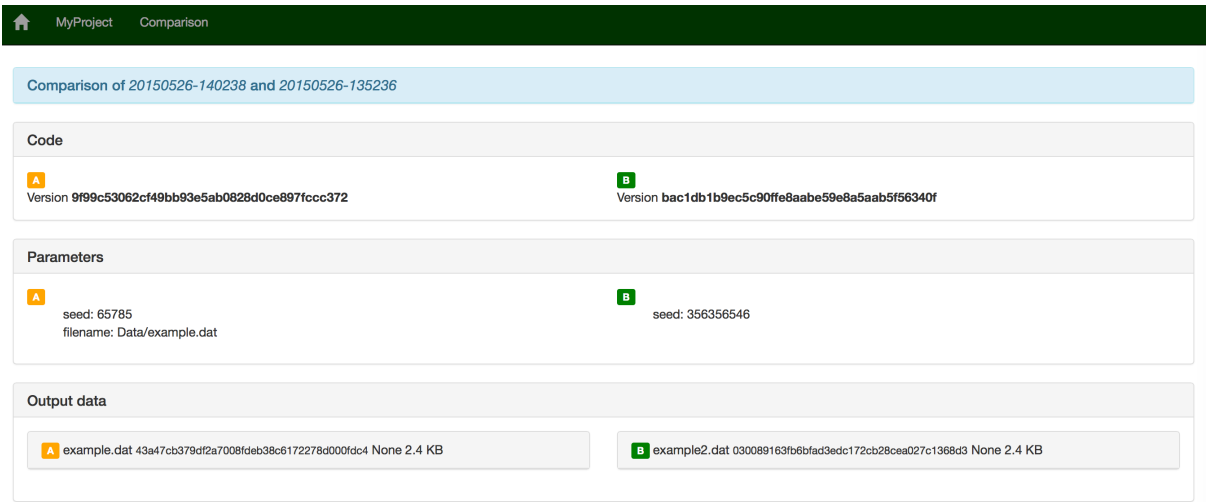
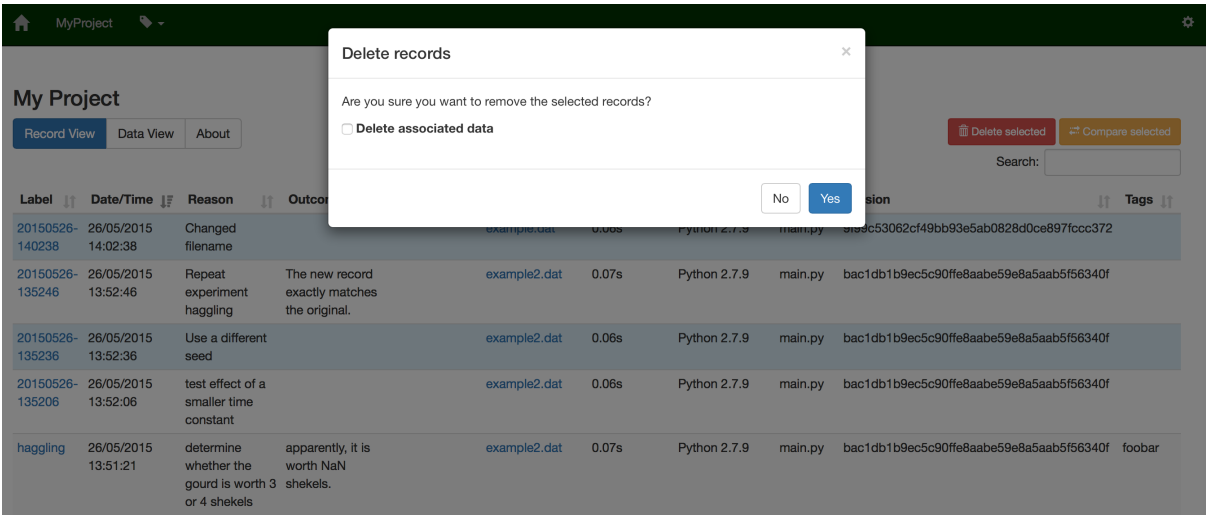
When deleting records, you have the option of also deleting any data generated by that simulation or analysis.

Comparing records

Any pair of records can be compared.

Filtering the records

You can filter the records by clicking on the ‘tag’ icon or by using the search field.



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
- parameters
- input data
- output data
- dependencies
- platform information
- stdout & stderr

MyProject 20150526-140238

20150526-140238
Delete record

```
$ /Users/andrew/anaconda/envs/sumatra/bin/python main.py <parameters>
```

Run on 26/05/2015 14:02:38 by Andrew Davison <andrew.davison@unic.cnrs-gif.fr>

Working directory: /Users/andrew/tmp/sumatra_example_projects/myproject
Code version: 9f99c53062cf49bb93e5ab0828d0ce897fcc372
Repository: /Users/andrew/tmp/sumatra_example_projects/myproject
Python version: 2.7.9
Reason: Changed filename
Tags:

Edit
Edit

+ Add outcome

Parameters

n: 100
seed: 65785
filename: Data/example.dat
distr: uniform

Output data

Filename	Path	Digest	Size	Date/Time	Output of	Input to
example.dat	example.dat	43a47cb37...	2.4 KB	26/05/2015 14:02:38	20150526-140238	

Dependencies

Name	Path	Version
nose	/Users/andrew/anaconda/envs/sumatra/lib/python2.7/site-packages/nose	1.3.4
numpy	/Users/andrew/anaconda/envs/sumatra/lib/python2.7/site-packages/numpy	1.9.0
scipy	/Users/andrew/anaconda/envs/sumatra/lib/python2.7/site-packages/scipy	0.14.0

Platform information

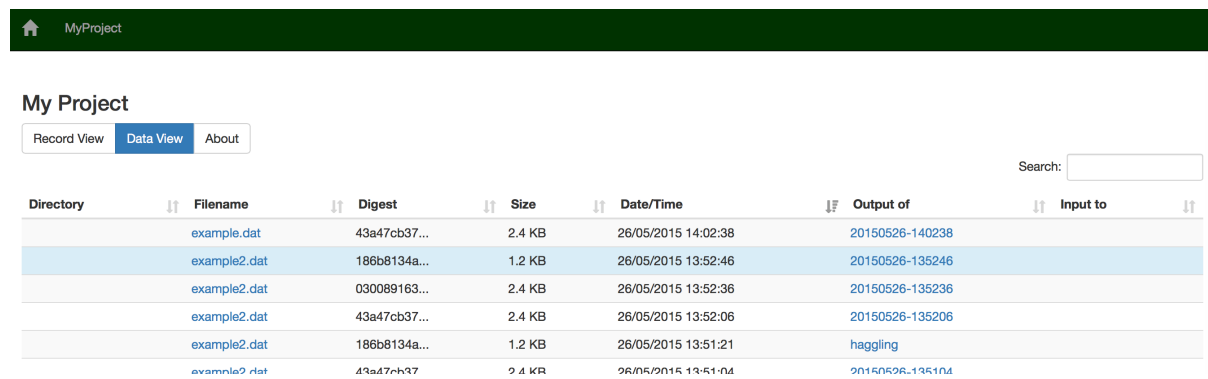
Name	IP address	Processor	Architecture	System type	Release	Version
unic193.inaf.cnrs-gif.fr	127.0.0.1	i386 x86_64	64bit	Darwin	14.3.0	Darwin Kernel Version 14.3.0: Mon Mar 23 11:59:05 PDT 2015; root:xnu-2782.20.48~5/RELEASE_X86_64

Stdout & Stderr

No output.

Data view

To view the history of your project from a data-centric point of view, click on the “Data View” tab.



The screenshot shows the 'My Project' page of the Sumatra web interface. It features a dark green header with a home icon and the text 'MyProject'. Below the header, there's a 'My Project' section with tabs for 'Record View', 'Data View' (which is active), and 'About'. A search bar is located on the right. The main content is a table with columns: Directory, Filename, Digest, Size, Date/Time, Output of, and Input to. The table contains several rows of data, including files named 'example.dat' and 'example2.dat' with various digests and sizes.

Directory	Filename	Digest	Size	Date/Time	Output of	Input to
	example.dat	43a47cb37...	2.4 KB	26/05/2015 14:02:38	20150526-140238	
	example2.dat	186b8134a...	1.2 KB	26/05/2015 13:52:46	20150526-135246	
	example2.dat	030089163...	2.4 KB	26/05/2015 13:52:36	20150526-135236	
	example2.dat	43a47cb37...	2.4 KB	26/05/2015 13:52:06	20150526-135206	
	example2.dat	186b8134a...	1.2 KB	26/05/2015 13:51:21	haggling	
	example2.dat	43a47cb37...	2.4 KB	26/05/2015 13:51:04	20150526-135104	

You can also see details about individual data items, including a preview of the file contents for file formats which Sumatra understands (e.g. images, CSV data).



The screenshot shows the 'Data' view of the Sumatra web interface. It features a dark green header with a home icon and the text 'MyProject Data'. Below the header, there's a section for 'example.dat' with a blue header bar. The details show the file's digest, size (2.4 KB), and a 'None' output. Below this, it says 'Generated by 20150526-140238 on 26/05/2015 at 14:02:38' and 'File contents cannot be shown.' A 'Download' button is at the bottom.

Image View

To view the image-rendered outputs of the records, click on the “Image view” button.

To switch view option from table to thumbgrid click on symbols in top right corner.

In thumbgrid the site starts to fetch the first 8 images. If you want to load 8 more images click ‘load more’ button at the end of the page. Otherwise by clicking ‘load all’ button the page will display all images.

Finishing up

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

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 best way to proceed is to copy the default templates from “/path/to/sumatra/web/templates” to “/path/to/myproject/.smt/templates” and then modify them.

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.

If the name of your parameter file ends in a recognized extension, such as “.json” or “.yaml”, Sumatra will use the associated format, otherwise it chooses the appropriate format based on file contents.

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/>.

YAML

See <http://www.yaml.org>

Hierarchical parameter set format

The `parameters` package (formerly part of NeuroTools) provides a format based on 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,
  }
}
```

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 NTPParameterSet, 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.

Input and output data

If your simulation, data analysis or other computation consumes and/or produces data, Sumatra will try to track them for you.

Sumatra does not, by default, make copies of your files, although this is an option. Rather, it stores the location of the data (e.g. the filesystem path) and, importantly, the SHA-1 hash of the file contents. This hash can later be used to check that the datafiles have not been corrupted or over-written.

Note: in its underlying design, Sumatra is agnostic as to how and where data files are stored. At the moment, however, Sumatra assumes that data are stored in files (rather than in a relational database, for example), that input

files are available on your local file system, and that output files will be written to the local file system. If this does not fit with your current workflow, please contact the developers.

Telling Sumatra about your input data files

Sumatra currently handles two cases:

- your script or executable reads data from standard input (`stdin`);
- the names of the input data files are given in the command-line invocation of your program.

If your workflow is different, e.g. if the filenames are hard-coded or provided in a parameter/configuration file, Sumatra will not at present track the files: please contact the developers if you need this functionality.

Reading data from `stdin`

If your usual invocation is something like:

```
$ myprog < mydata.dat
```

Then to run and track this with Sumatra, use:

```
$ smt run -e myprog -i mydata.dat
```

Reading data from files

If you normally launch computations with something like:

```
$ python main.py config_file data_file1 data_file2
```

Then with Sumatra, run:

```
$ smt configure -e python -m main.py
$ smt run config_file data_file1 data_file2
```

If your parameter/configuration file is in a format Sumatra recognizes, it will be treated specially (see [Parameter files](#)). Otherwise, it will be treated in the same way as the data files.

Specifying relative paths

By default, Sumatra will store the full path of input data files. If you would prefer to store the path relative to your working directory or some other directory (this is useful if running computations for the same project on multiple computers, or if you need to relocate your project directory), you can specify this as follows:

```
$ smt configure --input . # relative to working directory
```

Telling Sumatra where to find your output data files

Sumatra will try to automatically detect any new data files created by your simulation or analysis script. To avoid the overhead of having to monitor the entire file system, you need to give Sumatra the path of a directory to monitor. By default, Sumatra will look in a subdirectory “Data” of the working directory, but this is rarely a very useful default, so you will usually want to configure it yourself:

```
$ smt configure --datapath /path/to/data
```

Keeping a copy of output data

In general, it is up to you and your scripts/programs to manage your output data files, for example including the date and time in each output filename to ensure it is unique and doesn't get over-written, making backups, etc.

However, Sumatra does have an option to automatically make an archive copy of your output data, relieving you of the need to do so. To activate this, you need to choose a base directory in which to store your archives, e.g.:

```
$ smt configure --archive ./archive
```

For each computation, Sumatra will then create a compressed tar archive of all your output data files, label it with the date and time, and store it in the archive directory. This is particularly useful if your program always uses the same output filename (such as "output.dat") as it avoids accidental over-writing.

Dropbox, and other data-mirrors

If you are collaborating with others, or if you wish to disseminate your results publicly (for example, in conjunction with the [Sumatra Server](#) remote record store) then you need to put your results online where they are accessible, and still let Sumatra know where to find them.

Sumatra does not take care of putting your data online, this is up to you, but if there is a simple mapping between the local filesystem path and the online URL (e.g. using Dropbox with a public folder), you can tell Sumatra about it using the mirror option, e.g.:

```
$ smt configure --mirror https://dl.dropboxusercontent.com/u/xyzxyz/
```

You will have to figure out what "xyzxyz" should be for your own public folder.

Running multiple computations at the same time

Sumatra infers which files have been created by your computation by taking a snapshot of the designated output directory immediately before launching your computation and then determining what has changed once the computation is finished. This means that if you have multiple processes writing to the same directory, Sumatra will get confused about which files were created by which process.

A workaround is to ensure that each computation writes to a different directory, and then use **smt configure --datapath** immediately before each run to tell Sumatra which directory to look in.

A more streamlined version of this workflow is to use the **--addlabel** option to have Sumatra automatically add the label/id of the computation to either the command line or the parameter file. It is then up to your script or program to read this label and use it as the subdirectory within which to write data. Sumatra will then look only in that directory for output data. This is simpler than it sounds. Suppose I have a Python script "myscript.py" which reads a data file, does some calculations and then writes data to a file "mydata/output.csv":

```
filename = os.path.join("mydata", "output.csv")
with open(filename, "wb") as fp:
    fp.write(data)
```

Sumatra is configured as follows:

```
$ smt configure --datapath=mydata --addlabel=cmdline --executable=python --
↪main=myscript.py
```

The script should be modified as follows:

```
label = sys.argv[-1]    # Sumatra appends the label to the command line
subdir = os.path.join("mydata", label)
os.mkdir(subdir)
filename = os.path.join(subdir, "output.csv")
with open(filename, "wb") as fp:
    fp.write(data)
```

Now we can run the script many times in parallel, e.g.:

```
$ for input_file in *.dat; do smt run $input_file & done
```

Each run will be given a separate, unique label by Sumatra (by default, based on the current date and time); the Python script reads this label from the command line and uses it to create a unique subdirectory into which it saves the output data; Sumatra knows to look only in this directory for files associated with the given run, so there is no chance of mixing up data from different runs.

Writing data to stdout

If your program writes data to standard output, i.e. you would normally run it using:

```
$ myprog > output.txt
```

Then you can tell Sumatra to run it the same way, but in addition to track the output file, using:

```
$ smt run -o output.txt
```

Parallel computations

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 computation 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](#).

Reproducible publications: including and linking to provenance information in documents

Sumatra provides tools to include figures and other results generated by Sumatra-tracked computations in documents, with links to full provenance information: i.e. the full details of the code, input data and computational environment used to generate the figure/result.

LaTeX

To include figures from a Sumatra project in a LaTeX document, copy the file `sumatra.sty` from the `sumatra/publishing/latex` directory in the Sumatra source distribution into your working directory, then add

```
\usepackage{sumatra}
```

in the preamble of your LaTeX document. If your LaTeX working directory is not the same as your Sumatra project directory, you can specify the location of the record store and the project name as package options:

```
\usepackage[recordstore=/path/to/db, project=MyProject]{sumatra}
```

You can then use the `\smtincludegraphics` command in place of the usual `\includegraphics`:

```
\smtincludegraphics{20120907-153528:my_figure.png}
```

Here the argument is the label of a Sumatra record. This command will look up the record in your Sumatra project, find the location of the image file (whether a local file path or a URL) and include the figure in the document. If you are using the `HttpRecordStore`, the figure will also be a hyperlink to the corresponding Sumatra record, so you can easily check the full provenance of the figure.

If `my_figure.png` is the only image file produced by that Sumatra run, then you can use just:

```
\smtincludegraphics{20120907-153528}
```

without the path part.

You can also use just a fragment of the image file name as a query term, replacing the “:” separator with “?”:

```
\smtincludegraphics{20120907-153528?my_fig}
```

All the usual `\includegraphics` options are supported, e.g.:

```
\smtincludegraphics[width=\textwidth]{20120907-153528:my_figure.png}
```

You can also include as an option the SHA1 hash of the image file contents, as captured by Sumatra, to ensure that the image you include is the correct one, and that it hasn’t been accidentally overwritten or replaced by another:

```
\smtincludegraphics[width=\textwidth,↵↵digest=e2d1054c2893f19f50c43ddd5a344b59383df648]{20120907-153528:my_figure.png}
```

Note: you will have to run **latex/pdflatex** with the `-shell-escape` option.

Sphinx

To include figures from a Sumatra project, or links to Sumatra records of simulations or analyses, in a [Sphinx](#) document, you should first add `'sumatra.publishing.sphinxext'` to the `extensions` list in your `conf.py` and then set the following options:

```
sumatra_record_store = "/path/to/record/store"
sumatra_project = "MyProject"
sumatra_link_icon = "icon_info.png"
```

The `sumatra.publishing.sphinxext` extension provides a `reStructuredText` directive `smtimage` and a `reStructuredText` role `smtlink`

The `smtimage` directive

`smtimage` includes an image file retrieved from a Sumatra project in your document. If you are using the `HttpRecordStore`, the image will also be a hyperlink to the corresponding Sumatra record.

```
.. smtimage:: 20120907-153528:my_figure.png
```

All the usual options to the normal `image` directive can be used (`'width'`, `'height'`, `'align'`, etc.), as well as the Sumatra-specific options “digest” (which is used to check the identity of the included image, as detailed in the section on LaTeX above) “record_store” and “project”, which can be used to over-ride the global options specified in `conf.py`, and so mix results from multiple projects in a single document. Here is a rather complete example.

```
.. smtimage:: 20120907-153528?my_fig
:record_store: http://sumatra.example.com/
:project: MyOtherProject
:digest: e2d1054c2893f19f50c43ddd5a344b59383df648
:width: 800 px
:align: center
:class: some-css-class
```

The smtlink role

If using a `HttpRecordStore`, the `smtlink` role inserts an icon in the document, which is a hyperlink to a given record in the store.

To use this, you must set the `sumatra_link_icon` option, as discussed above.

Exporting and reporting

The `smt list` command has various output formats: text (the default), HTML, LaTeX, and shell. The latter produces a shell script that could be run to repeat all the computations recorded in your Sumatra project.

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.

Record stores

Sumatra supports multiple back-end databases for storing records. Sumatra uses the name “record store” for these databases, to distinguish them from other databases you may be using. Which one you should choose depends on your needs.

Single user, one record store per project

This is the default setting. If you create a new Sumatra project without specifying the `--store` option, a file `.smt/records` is created, which contains an [SQLite](#) database (if you have installed [Django](#)) or a [shelve](#) database (but in this case, you will not be able to use the web browser interface).

Single user, all projects in a single record store

If you have multiple projects, and you wish to store all the records in a single database, you can specify where this should be located, e.g.:

```
$ smt init --store=~/.sumatra.db MyProject
```

If the database file does not already exist, it will be created.

For better performance (e.g., if you are running many Sumatra jobs at the same time) you can use PostgreSQL instead of SQLite. For this, you will need to install the [psycopg2](#) package, then specify the database connection parameters in the form:

```
$ smt init --store=postgres://username:password@hostname/databasename MyProject
```

Collaborating within a single lab

If two or more people within your lab are using Sumatra, you can all use the same Sumatra record store. For this, the easiest option is to use a shared file system (e.g., using SAMBA or NFS) or run your PostgreSQL database on its own server. If this is not possible, you should use the network record store described in the next section.

Collaborating with people in a different lab

To collaborate with people in a different lab you can set up a server running [Sumatra Server](#) (distributed separately from Sumatra) or any other server software implementing the same API. You then set up your project as follows:

```
$ smt init --store=https://username:password@hostname/
```

Now when you run computations, Sumatra will send the records to the server, from where your collaborators can access them if they have an account on the server, or the project is set to public. Note that this does not transfer data files to the server, this has to be taken care of separately, using a mirroring data store.

Note: at present, if the network connection fails the simulation record will be lost, so if you have a poor network connection or very long-running computations, it is probably safer to use a local record store. In the future we plan to add a caching mechanism which will keep a local copy of the record and retry the connection to the server later.

The easiest way to try Sumatra Server out is to use Docker.

Open science

If you want to try [open notebook science](#) you should use the network record store as described in the previous section, and set your projects to public.

Changing record stores

The **smt configure** command also accepts the `--store` option. In this case, Sumatra will copy all the records from the old store to the new one. It will not delete the old one.

Warning: this feature is quite new, so we recommend making a backup of your Sumatra project before attempting to change record stores.

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 the 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.

If you accidentally upgraded your Sumatra version without exporting the old project first, you will need to roll back to the previous version in order to be able to do the export. See below for an example how to do this using `pip`.

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`.

Install the new version and upgrade

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

```
$ pip install --upgrade 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_20140209132422`.

Downgrading an accidentally upgraded Sumatra to export the project

Exporting a project will need to be done with the same Sumatra version that the project was run with last. If you accidentally upgraded your Sumatra version without exporting the old project first, you will need to roll back to the previously installed version. You can explicitly specify a version number with `pip` as follows:

```
$ pip install sumatra==0.5.3
```

Replace 0.5.3 with the correct version number for your project. In case you are unsure, it may help to run the following command in your project directory:

```
$ cat .smt/project | grep sumatra_version
```

Extending Sumatra with plug-ins

To support the wide diversity of scientific workflows, Sumatra aims to make it easy (or at least possible) to customize the way it works. The following components can be customized:

- launching computations (local execution, distributed using MPI, batch jobs using SLURM, ...)
- parameter file formats (JSON, YAML, ...)
- executable programs/programming languages (Python, MATLAB, R, ...)
- data stores (local disk, network storage, WebDAV, Dropbox, ...)
- record stores (SQLite, PostgreSQL, web-based, ...)
- output formats (plain text, HTML, LaTeX, JSON, ...)
- version control systems (Git, Mercurial, Subversion, ...)

Here is an example of wrapping the PHP command-line interface. (In itself, this gives little or no benefit, as Sumatra can automatically obtain version information from most command-line tools, but (a) it is a prerequisite if we wish to add dependency tracking, (b) it makes a nicely simple example of writing a plug-in!)

```
from sumatra.core import component
from sumatra.programs import Executable

@component
class PHPcli(Executable):
    name = "PHP"
    executable_names = ('php',)
    file_extensions = ('.php',)
    default_executable_name = 'php'
    requires_script = True
```

To use this plug-in in a Sumatra project, it should be saved in a file (e.g. `php.py`) that is on the Python module search path (for example, installed in `site-packages` or added to the `PYTHONPATH` environment variable), and then added to the project using:

```
$ smt configure --add-plugin=php
```

You can also manually edit `.smt/project` and add the module path to the “plugins” list.

To list the installed plug-ins, use `smt info`.

To remove a plug-in from a project:

```
$ smt configure --remove-plugin=php
```

Component interfaces

As shown in the PHP example above, writing a custom component involves writing a new class which inherits from a Sumatra base class, and then registering the class with the component registry.

The available base classes are:

- `sumatra.datastore.base.DataStore`
- `sumatra.datastore.base.DataItem`
- `sumatra.formatting.Formatter`
- `sumatra.recordstore.base.RecordStore`
- `sumatra.versioncontrol.base.Repository`
- `sumatra.versioncontrol.base.WorkingCopy`
- `sumatra.launch.LaunchMode`
- `sumatra.parameters.ParameterSet`
- `sumatra.programs.Executable`

For full details of which methods need to be implemented in your sub-class, see the [API reference](#).

Note: At present, there is no mechanism for plug-ins to change the list of options for command-line tools such as smt. This is planned for the future.

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.

smt command reference

comment

```
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. By default, comments will be appended to **any** existing comments. To overwrite existing comments, use the **'-r/--replace flag'**.

positional arguments:

LABEL	the record to which the comment will be added
comment	a string of text, or the name of a file containing the

```
comment.
```

optional arguments:

```
-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
```

configure

```
usage: smt configure [options]
```

Modify the settings for the current project.

optional arguments:

```
-h, --help          show this help message and exit
-d PATH, --datapath PATH
                    set the path to the directory in which smt will search
                    for datafiles generated by the simulation or analysis.
-i PATH, --input PATH
                    set the path to the directory in which smt will search
                    for input datafiles.
-l OPTION, --addlabel OPTION
                    If this option is set, smt will append the record
                    label either to the command line (option 'cmdline') or
                    to the parameter file (option 'parameters'), and will
                    add the label to the datapath when searching for
                    datafiles. It is up to the user to make use of this
                    label inside their program to ensure files are created
                    in the appropriate location.
-e PATH, --executable PATH
                    set the path to the executable.
-r REPOSITORY, --repository REPOSITORY
                    the URL of a Subversion or Mercurial repository
                    containing the code. This will be checked out/cloned
                    into the current directory.
-m MAIN, --main MAIN
                    the name of the script that would be supplied on the
                    command line if running the simulator normally, e.g.
                    init.hoc.
-c {store-diff,error}, --on-changed {store-diff,error}
                    may be 'store-diff' or 'error': the action to take if
                    the code in the repository or any of the dependencies
                    has changed.
-g OPTION, --labelgenerator OPTION
                    specify which method Sumatra should use to generate
                    labels (options: timestamp, uuid)
-t TIMESTAMP_FORMAT, --timestamp_format TIMESTAMP_FORMAT
                    the timestamp format given to strftime
-L {serial,distributed,slurm-mpi}, --launch_mode {serial,distributed,slurm-mpi}
                    how computations should be launched.
-o LAUNCH_MODE_OPTIONS, --launch_mode_options LAUNCH_MODE_OPTIONS
                    extra options for the given launch mode, to be given
                    in quotes with a leading space, e.g. ' --foo=3'
-p, --plain          pass arguments to the 'run' command straight through
                    to the program. Otherwise arguments of the form
                    name=value can be used to overwrite default parameter
                    values.
--no-plain           arguments to the 'run' command of the form name=value
                    will overwrite default parameter values. This is the
```

opposite of the `--plain` option.

`-s STORE, --store STORE` Change the record store to the specified path, URL or URI (must be specified). The argument can take the following forms: (1) ``/path/to/sqlitedb`` - DjangoRecordStore is used with the specified SQLite database, (2) ``http[s]://location`` - remote HTTPRecordStore is used with a remote Sumatra server, (3) ``postgres://username:password@hostname/databasename`` - DjangoRecordStore is used with specified Postgres database.

`-W URL, --webdav URL` specify a webdav URL (with username@password: if needed) as the archiving location for data

`-A PATH, --archive PATH` specify a directory in which to archive output datafiles. If not specified, or if `'false'`, datafiles are not archived.

`-M URL, --mirror URL` specify a URL at which your datafiles will be mirrored.

`--add-plugin ADD_PLUGIN` name of a Python module containing one or more plugins.

`--remove-plugin REMOVE_PLUGIN` name of a plug-in module to remove from the project.

delete

usage: `smt delete [options] LIST`

LIST should be a space-separated list of labels **for** individual records **or** of tags. If it contains tags, you must **set** the `--tag/-t` option (see below). The special value `"last"` allows you to delete the most recent simulation/analysis. If you want to delete **all** records, just delete the `.smt` directory **and** use `smt init` to create a new, empty project.

positional arguments:

LIST a space-separated list of labels **for** individual records **or** of tags

optional arguments:

`-h, --help` show this help message **and** exit
`-t, --tag` interpret LIST **as** containing tags. Records **with any** of these tags will be deleted.
`-d, --data` also delete **any** data associated **with** the record(s).

diff

usage: `smt diff [options] LABEL1 LABEL2`

Show the differences, **if any**, between two records.

positional arguments:

label1
 label2

optional arguments:

`-h, --help` show this help message **and** exit

```
-i IGNORE, --ignore IGNORE
                        a regular expression pattern for filenames to ignore
                        when evaluating differences in output data. To supply
                        multiple patterns, use the -i option multiple times.
-l, --long              prints full information for each record
```

export

usage: smt export

Export a Sumatra project **and** its records to JSON. This **is** needed before running upgrade.

optional arguments:

```
-h, --help  show this help message and exit
```

help

usage: smt help CMD

Get help on an smt command.

positional arguments:

```
cmd
```

optional arguments:

```
-h, --help  show this help message and exit
```

info

usage: smt info

Print information about the current project.

optional arguments:

```
-h, --help  show this help message and exit
```

init

usage: smt init [options] NAME

Create a new project called NAME in the current directory.

positional arguments:

```
NAME                a short name for the project; should not contain
                    spaces.
```

optional arguments:

```
-h, --help          show this help message and exit
-d PATH, --datapath PATH
                    set the path to the directory in which smt will search
                    for output datafiles generated by the
                    simulation/analysis. Defaults to ./Data.
-i PATH, --input PATH
```

```

        set the path to the directory relative to which input
        datafile paths will be given. Defaults to the
        filesystem root.
-l OPTION, --addlabel OPTION
        If this option is set, smt will append the record
        label either to the command line (option 'cmdline') or
        to the parameter file (option 'parameters'), and will
        add the label to the datapath when searching for
        datafiles. It is up to the user to make use of this
        label inside their program to ensure files are created
        in the appropriate location.
-e PATH, --executable PATH
        set the path to the executable. If this is not set,
        smt will try to infer the executable from the value of
        the --main option, if supplied, and will try to find
        the executable from the PATH environment variable,
        then by searching various likely locations on the
        filesystem.
-r REPOSITORY, --repository REPOSITORY
        the URL of a Subversion or Mercurial repository
        containing the code. This will be checked out/cloned
        into the current directory.
-m MAIN, --main MAIN
        the name of the script that would be supplied on the
        command line if running the simulation or analysis
        normally, e.g. init.hoc.
-c ON_CHANGED, --on-changed ON_CHANGED
        the action to take if the code in the repository or
        any of the dependencies has changed. Defaults to
        error
-s STORE, --store STORE
        Specify the path, URL or URI to the record store (must
        be specified). This can either be an existing record
        store or one to be created. The argument can take the
        following forms: (1) `/path/to/sqlitedb` -
        DjangoRecordStore is used with the specified Sqlite
        database, (2) `http[s]://location` - remote
        HTTPRecordStore is used with a remote Sumatra server,
        (3)
        `postgres://username:password@hostname/databasename` -
        DjangoRecordStore is used with specified Postgres
        database. Not using the `--store` argument defaults to
        a DjangoRecordStore with Sqlite in `.smt/records`
-g OPTION, --labelgenerator OPTION
        specify which method Sumatra should use to generate
        labels (options: timestamp, uuid)
-t TIMESTAMP_FORMAT, --timestamp-format TIMESTAMP_FORMAT
        the timestamp format given to strftime
-L {serial,distributed,slurm-mpi}, --launch_mode {serial,distributed,slurm-mpi}
        how computations should be launched. Defaults to
        serial
-o LAUNCH_MODE_OPTIONS, --launch_mode_options LAUNCH_MODE_OPTIONS
        extra options for the given launch mode
-W URL, --webdav URL
        specify a webdav URL (with username@password: if
        needed) as the archiving location for data
-A PATH, --archive PATH
        specify a directory in which to archive output
        datafiles. If not specified, or if 'false', datafiles
        are not archived.
-M URL, --mirror URL
        specify a URL at which your datafiles will be
        mirrored.

```

list

```
usage: smt list [options] [TAGS]
```

If TAGS (optional) **is** specified, then only records tagged **with all** the tags **in** TAGS will be listed.

positional arguments:

TAGS

optional arguments:

-h, --help show this help message **and** exit
-l, --long prints full information **for** each record
-T, --table prints information **in** tab-separated columns
-f FMT, --format FMT FMT can be 'text' (default), 'html', 'json', 'latex' or 'shell'.
-r, --reverse list records **in** reverse order (default: newest first)
-m NAME, --main_file NAME filter list of records by main file
-P, --parameter_view list records **with** parameter values

migrate

```
usage: smt migrate [options]
```

If you have moved your data files to a new location, update the record store to reflect the new paths.

optional arguments:

-h, --help show this help message **and** exit
-d PATH, --datapath PATH modify the path to the directory **in** which your results are stored.
-i PATH, --input PATH modify the path to the directory **in** which your input data files are stored.
-A PATH, --archive PATH modify the directory **in** which your results are archived.
-M URL, --mirror URL modify the URL at which your data files are mirrored.

repeat

```
usage: smt repeat LABEL
```

Re-run a previous simulation/analysis under (**in** theory) identical conditions, **and** check that the results are unchanged.

positional arguments:

LABEL label of record to be repeated

optional arguments:

-h, --help show this help message **and** exit
-l NEW_LABEL, --label NEW_LABEL specify a label **for** the new experiment. If no label **is** specified, one will be generated automatically.

run

```
usage: smt run [options] [arg1, ...] [param=value, ...]
```

The `list` of arguments will be passed on to the simulation/analysis script. It should normally contain at least the name of a parameter file, but can also contain `input` files, flags, etc. If the parameter file should be `in` a `format` that Sumatra understands (see documentation), then the parameters will be stored to allow future searching, comparison, etc. of records. For convenience, it `is` possible to specify a file `with` default parameters `and` then specify those parameters that are different `from the` default values on the command line `with any` number of `param=value` pairs (note no space around the equals sign).

optional arguments:

```
-h, --help          show this help message and exit
-v REV, --version REV
                    use version REV of the code (if this is not the same
                    as the working copy, it will be checked out of the
                    repository). If this option is not specified, the most
                    recent version in the repository will be used. If
                    there are changes in the working copy, the user will
                    be prompted to commit them first
-l LABEL, --label LABEL
                    specify a label for the experiment. If no label is
                    specified, one will be generated automatically.
-r REASON, --reason REASON
                    explain the reason for running this
                    simulation/analysis.
-e PATH, --executable PATH
                    Use this executable for this run. If not specified,
                    the project's default executable will be used.
-m MAIN, --main MAIN
                    the name of the script that would be supplied on the
                    command line if running the simulation/analysis
                    normally, e.g. init.hoc. If not specified, the
                    project's default will be used.
-n N, --num_processes N
                    run a distributed computation on N processes using
                    MPI. If this option is not used, or if N=0, a normal,
                    serial simulation/analysis is run.
-t TAG, --tag TAG   tag you want to add to the project
-D, --debug         print debugging information.
-i STDIN, --stdin STDIN
                    specify the name of a file that should be connected to
                    standard input.
-o STDOUT, --stdout STDOUT
                    specify the name of a file that should be connected to
                    standard output.
```

sync

```
usage: smt sync PATH1 [PATH2]
```

Synchronize two record stores. If both `PATH1` `and` `PATH2` are given, the record stores at those locations will be synchronized. If only `PATH1` `is` given, `and` the command `is` run `in` a directory containing a Sumatra project, only that project's records be synchronized with the store at `PATH1`. Note that `PATH1` and `PATH2` may be either filesystem paths `or` URLs.

positional arguments:

```
path1
```

```
path2
```

optional arguments:

```
-h, --help  show this help message and exit
```

tag

```
usage: smt tag [options] TAG [LIST]
```

If TAG contains spaces, it must be enclosed in quotes. LIST should be a space-separated list of labels for individual records. If it is omitted, only the most recent record will be tagged. If the '-r/--remove' option is set, the tag will be removed from the records.

positional arguments:

```
  TAG          tag to add
  LIST         a space-separated list of records to be tagged
```

optional arguments:

```
-h, --help      show this help message and exit
-r, --remove    remove the tag from the record(s), rather than adding it.
```

upgrade

```
usage: smt upgrade
```

Upgrade an existing Sumatra project. You must have previously run "smt export" or the standalone 'export.py' script.

optional arguments:

```
-h, --help  show this help message and exit
```

version

```
usage: smt version
```

Print the Sumatra version.

optional arguments:

```
-h, --help  show this help message and exit
```

Developers' guide

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

Requirements

- Python 2.6, 2.7, 3.4 or 3.5
- Django >= 1.6
- django-tagging >= 0.3
- parameters >= 0.2.1

- `nose` `>= 0.11.4`
- `future` `>= 0.14`
- if using Python `< 3.4`, `pathlib` `>= 1.0.0`
- if using Python `2.6`, `ordereddict`, `unittest2` `>= 0.5.1`
- `docutils`
- `Jinja2`

Optional:

- `mpi4py` `>= 1.2.2`
- `coverage` `>= 3.3.1` (for measuring test coverage)
- `httplib2` (for the remote record store)
- `GitPython` (for Git support)
- `mercurial` and `hgapi` (for Mercurial support)
- `bzr` (for Bazaar support)
- `PyYAML` (for YAML support)
- `psycopg2` (for PostgreSQL support)
- `dexml` and `fs` (for WebDAV support)

We strongly recommend developing within a `virtualenv`.

Getting the source code

We use the Git version control system. To get a copy of the code you should fork the main [Sumatra repository](#) on [Github](#), then clone your own fork.:

```
$ cd /some/directory
$ git clone https://github.com/<username>/sumatra.git
```

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
$ 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 installing using `pip` with the “editable” option:

```
$ pip install --editable sumatra
```

To ensure you always have access to the most recent version, add the main repository as “upstream”:

```
$ git remote add upstream https://github.com/open-research/sumatra.git
```

To update to the latest version from the repository:

```
$ git pull upstream master
```

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/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.

If any of the tests fail, check out the [continuous integration server](#) to see if these are "known" failures, otherwise please [open a bug report](#).

(many thanks to the [NEST Initiative](#) for hosting the CI server).

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
```

Committing your changes

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

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

and then push them to your Github repository:

```
$ git push
```

Before pushing, run the test suite again to check that you have not introduced any new bugs.

Once you are ready for your work to be merged into the main Sumatra repository, please open a pull request. You are encouraged to use a separate branch for each feature or bug-fix, as it makes merging changes easier.

Coding standards and style

All code should conform as much as possible to [PEP 8](#), and should run with Python 2.6, 2.7, 3.4 and 3.5. Lines should be no longer than 99 characters.

Reviewing pull requests

All contributors are encouraged to review pull requests, and all pull requests must have at least one review before merging.

Things to check for:

- Does the pull request implement a single, well-defined piece of functionality? (pull requests which perform system-wide refactoring are sometimes necessary, but need much more careful scrutiny)
- Does the code work? Is the logic correct? Does it anticipate possible failure conditions (e.g. lack of internet connection)?
- Is the code easily understood?
- Does the code conform to the coding standards (see above)?
- Does the code implement a general solution, or is the code too specific to a particular (language/version control system/storage backend)?

- Do all public functions/classes have docstrings?
- Are there tests for all new/changed functionality?
- Has the documentation been updated?
- Has the Travis CI build passed?
- Is the syntax compatible with both Python 2 and 3? (even if we don't yet support Python 3, any new code should try to do so)
- Is there any redundant or duplicate code?
- Is the code as modular as possible?
- Is there any commented out code, or print statements used for debugging?

API reference

Input and output data

The `datastore` module provides an abstraction layer around data storage, allowing different methods of storing simulation/analysis results (local filesystem, remote filesystem, database, etc.) to provide a common interface.

The interface is built around three types of object: a `DataStore` may contain many `DataItems`, each of which is identified by a `DataKey`.

There is a single `DataKey` class. `DataStore` and `DataItem` are abstract base classes, and must be subclassed to provide different functionality.

Base classes

class `sumatra.datastore.DataKey` (*path, digest, creation, **metadata*)

Identifies a `DataItem`, and may be used to retrieve a `DataItem` from a `DataStore`.

May also be used to store metadata (e.g. file size, mimetype) and be used as a proxy for the `DataItem` on a system where the actual data is not available.

path

a token used to retrieve a `DataItem`. For filesystem-based `DataStores`, this will be a relative path. For database-backed stores (none of which have been implemented yet :-)) it could be a primary key or an object encapsulating a query.

digest

the SHA1 digest of the contents of the associated `DataItem`. This attribute is calculated on creation of the `DataKey`.

metadata

a dict containing metadata, such as file size and mimetype.

next ()

class `sumatra.datastore.base.DataItem`

Base class for data item classes, that may represent files or database records.

digest

docstring

generate_key ()

Generate a `DataKey` uniquely identifying this data item.

get_content (*max_length=None*)

Return the contents of the data item as a string.

If *max_length* is specified, return that number of bytes, otherwise return the entire content.

next ()

save_copy (*path*)

Save a copy of the data to a local file.

If *path* is an existing directory, the data item path will be appended to it, otherwise *path* is treated as a full path including filename, either absolute or relative to the working directory.

Return the full path of the final file.

sorted_content ()

Return the contents of the data item, sorted by line.

class `sumatra.datastore.base.DataStore`

Base class for data storage abstractions.

contains_path (*path*)

Does the store contain a data item with the given path?

copy ()

delete (**keys*)

Delete the files corresponding to the given keys.

find_new_data (*timestamp*)

Finds newly created/changed data items

generate_keys (**paths*)

Given a number of “paths”, return a list of keys enabling the data at those paths to be retrieved from this store later.

get_content (*key*, *max_length=None*)

Return the contents of a file identified by a key.

If *max_length* is given, the return value will be truncated.

get_data_item (*key*)

Return the file that matches the given key.

next ()

required_attributes = (u'find_new_data', u'get_data_item', u'delete')

Storing data on the local filesystem

class `sumatra.datastore.FileSystemDataStore` (*root*)

Bases: `sumatra.datastore.base.DataStore`

Represents a locally-mounted filesystem. The root of the data store will generally be a subdirectory of the real filesystem.

root

The absolute path on the underlying file system to the root directory of the data store.

class `sumatra.datastore.filesystem.DataFile` (*path*, *store*, *creation=None*)

Bases: `sumatra.datastore.base.DataItem`

A file-like object, that represents a file in a local filesystem.

path

path relative to the `FileSystemDataStore` root

full_path

absolute path relative to the underlying filesystem.

size

file size in bytes

name
file name

extension
file extension

mimetype
if the mimetype cannot be guessed, this will be None

Automatic archiving of data written to the local filesystem

class `sumatra.datastore.ArchivingFileSystemDataStore` (*root*, *archive=u'.smt/archive'*)
Bases: `sumatra.datastore.filesystem.FileSystemDataStore`

Represents a locally-mounted filesystem that archives any new files created in it. The root of the data store will generally be a subdirectory of the real filesystem.

archive_store
Directory within which data will be archived.

class `sumatra.datastore.archivingfs.ArchivedDataFile` (*path*, *store*, *creation=None*)
Bases: `sumatra.datastore.base.DataItem`

A file-like object, that represents a file inside a tar archive

Mirroring data to a remote webserver

class `sumatra.datastore.MirroredFileSystemDataStore` (*root*, *mirror_base_url*)
Bases: `sumatra.datastore.filesystem.FileSystemDataStore`

Represents a locally-mounted filesystem whose contents are mirrored on a webserver, so that the files can be accessed via an HTTP URL.

mirror_base_url
URL to which the file path will be appended to obtain the final URL of a file

class `sumatra.datastore.mirroredfs.MirroredDataFile` (*path*, *store*, *creation=None*)
Bases: `sumatra.datastore.base.DataItem`

A file-like object, that represents a file existing both on a local file system and on a webserver.

Finding dependencies

The `dependency_finder` sub-package attempts to determine all the dependencies of a given script, including the version of each dependency.

For each executable that is supported there is a sub-module containing a `find_dependencies()` function, and a series of heuristics for finding version information. There is also a sub-module `core`, which contains heuristics that are independent of the language, e.g. where the dependencies are under version control.

copyright Copyright 2006-2015 by the Sumatra team, see `doc/authors.txt`

license BSD 2-clause, see `LICENSE` for details.

For users of the API, the principal function of interest is the following.

`sumatra.dependency_finder.find_dependencies` (*filename*, *executable*)
Return a list of dependencies for a given script and programming language.

filename: the path to the script whose dependencies should be found.

executable: an instance of `Executable` or one of its subclasses.

This function returns a list of `Dependency` objects. There is a different `Dependency` subclass for each programming language, but all have the following attributes:

```
class sumatra.dependency_finder.core.BaseDependency (name, path=None, version=u'unknown', diff=u'', source=None)
```

Contains information about a program component, and tries to determine version information.

name: an identifying name, e.g. the module name in Python

path: the location of the dependency file in the local filesystem

version: the version of the dependency, if that can be determined, otherwise 'unknown'. Always a string, even if the version can also be represented as a number.

diff: if the dependency is under version control and has been modified, the diff between the actual version and the last-committed version.

source: an identifier for where the dependency came from, if known, e.g. the url of a version control repository or the name of a Linux package.

If you are interested in improving the dependency finder for an existing program/language, or in adding a dependency finder for a new program or language, you may be interested in the following.

Language-independent heuristics and utilities

```
sumatra.dependency_finder.core.find_versions (dependencies, heuristics)
```

Try to find version information by calling a series of functions in turn.

dependencies: a list of Dependency objects.

heuristics: a list of functions that accept a component as the single argument and return a version number or 'unknown'.

Returns a possibly modified list of dependencies

```
sumatra.dependency_finder.core.find_versions_from_versioncontrol (dependencies)
```

Determine whether a file is under version control, and if so, obtain version information from this.

```
sumatra.dependency_finder.core.find_file (path, current_directory, search_dirs)
```

Look for path as an absolute path then relative to the current directory, then relative to *search_dirs*. Return the absolute path.

Python

```
sumatra.dependency_finder.python.find_versions_by_attribute (dependencies, executable)
```

Try to find version information from the attributes of a Python module.

```
sumatra.dependency_finder.python.find_versions_from_egg (dependencies)
```

Determine whether a Python module is provided as an egg, and if so, obtain version information from this.

```
sumatra.dependency_finder.python.find_imported_packages (filename, executable_path, debug=0, exclude_stdlib=True)
```

Find all imported top-level packages for a given Python file.

We cannot assume that the version of Python being used to run Sumatra is the same as that used to run the simulation/analysis. Therefore we need to run all the dependency finding and version checking in a subprocess with the correct version of Python.

```
sumatra.dependency_finder.python.find_dependencies (filename, executable)
```

Return a list of Dependency objects representing all the top-level modules or packages imported (directly or indirectly) by a given Python file.

Matlab

`sumatra.dependency_finder.matlab.find_dependencies (filename, executable)`

`sumatra.dependency_finder.matlab.save_dependencies (cmd, filename)`
save all dependencies to the file in the current folder

NEURON

`sumatra.dependency_finder.neuron.find_xopened_files (file_path)`

Find all files that are xopened, whether directly or indirectly, by a given Hoc file. Note that this only handles cases whether the path is given directly, not where it has been previously assigned to a `strdef`.

`sumatra.dependency_finder.neuron.find_loaded_files (file_path, executable_path)`

Find all files that are loaded with `load_file()`, whether directly or indirectly, by a given Hoc file. Note that this only handles cases whether the path is given directly, not where it has been previously assigned to a `strdef`. Also note that this is more complicated than `xopen()`, since NEURON also looks in any directories in `$HOC_LIBRARY_PATH` and `$NEURONHOME/lib/hoc`.

`sumatra.dependency_finder.neuron.find_dependencies (filename, executable)`

Return a list of Dependency objects representing all Hoc files imported (directly or indirectly) by a given Hoc file.

GENESIS

`sumatra.dependency_finder.genesis.find_included_files (file_path)`

Find all files that are included, whether directly or indirectly, by a given .g file.

`sumatra.dependency_finder.genesis.find_dependencies (filename, executable)`

Return a list of Dependency objects representing all files included, whether directly or indirectly, by a given .g file.

R

`sumatra.dependency_finder.r.find_dependencies (filename, executable)`

Return list of dependencies.

First determines dependency info for filename. This is done through an external call (using the Rscript from `executable.path`) to a custom R script that uses `parse` and `simple` pattern-matching to find all calls in filename that load external packages (i.e., the R calls “library” and “require”). The result is returned in a string with package info delimited by pre-set tokens. Info includes: name, version, local path, and repo source (repo name but no URLs).

Second, parses the dependency info into Dependency objects, returned in a list.

Storing provenance information

The `recordstore` module provides an abstraction layer around storage of simulation/analysis records, providing a common interface to different storage methods (simple serialisation, relational database, etc.)

Base class

All record store classes have the following methods. Some stores have additional methods (see below).

class `sumatra.recordstore.base.RecordStore`

Base class for record store implementations.

delete (*project_name, label*)

Delete the record with the given label from the given project.

delete_all ()

Delete all records from the store.

delete_by_tag (*project_name, tag*)

Delete all records from the given project that have been tagged with the given tag.

export (*project_name, indent=2*)

Returns a string with a JSON representation of the project record store.

export_records (*records, indent=2*)

Returns a string with a JSON representation of the given records.

get (*project_name, label*)

Retrieve the record with the given label from the given project.

has_project (*project_name*)

Does the store contain any records for the given project?

import_ (*project_name, content*)

Import records in JSON format.

labels (*project_name*)

Return the labels of all records in the given project.

list (*project_name, tags=None*)

Return a list of records for the given project.

If *tags* is not provided, list all records, otherwise list only records that have been tagged with one or more of the tags.

list_projects ()

Return the names of all projects that have records in this store.

most_recent (*project_name*)

Return the most recent record from the given project.

required_attributes = (u'list_projects', u'save', u'get', u'list', u'labels', u'delete', u'delete_all', u'delete_by_tag')

save (*project_name, record*)

Store the given record under the given project.

sync (*other, project_name*)

Synchronize two record stores so that they contain the same records for a given project.

Where the two stores have the same label (within a project) for different records, those records will not be synced. The method returns a list of non-synchronizable records (empty if the sync worked perfectly).

sync_all (*other*)

Synchronize all records from all projects between two record stores.

update (*project_name, field, value, tags=None*)

Modify the records for a given project.

Arguments: *field*: the name of a record attribute, e.g. "datastore.root" *value*:

Minimal record store

class `sumatra.recordstore.ShelveRecordStore` (*shelf_name=u'.smt/records'*)

Bases: `sumatra.recordstore.base.RecordStore`

Handles storage of simulation/analysis records based on the Python standard `shelve` module.

The advantage of this record store is that it has no dependencies. The disadvantages are that it allows only local access and does not support the *smtweb* interface.

Django-based record store

class `sumatra.recordstore.DjangoRecordStore` (*db_file=u'.smt/records'*)

Bases: `sumatra.recordstore.base.RecordStore`

Handles storage of simulation/analysis records in a relational database, via the Django object-relational mapper (ORM), which means that any database supported by Django could in principle be used, although for now we assume SQLite or PostgreSQL.

This record store is needed for the *smtweb* interface.

Client for remote record store

class `sumatra.recordstore.HttpRecordStore` (*server_url*, *username=None*, *password=None*,
disable_ssl_certificate_validation=True)

Bases: `sumatra.recordstore.base.RecordStore`

Handles storage of simulation/analysis records on a remote server using HTTP.

The server should support the following URL structure and HTTP methods:

/	GET
/<project_name>/[?tags=<tag1>,<tag2>,...]	GET
/<project_name>/tag/<tag>/	GET, DELETE
/<project_name>/<record_label>/	GET, PUT, DELETE

and should both accept and return JSON-encoded data when the Accept header is “application/json”.

The required JSON structure can be seen in `recordstore.serialization`.

create_project (*project_name*, *long_name=u''*, *description=u''*)

Create an empty project in the record store.

project_info (*project_name*)

Return a project’s long name and description.

update_project_info (*project_name*, *long_name=u''*, *description=u''*)

Update a project’s long name and description.

Module functions

`sumatra.recordstore.get_record_store` (*uri*)

Return the `RecordStore` object found at the given URI (which may be a URL or filesystem path).

Transferring provenance information

Handles serialization/deserialization of record store contents to/from JSON.

copyright Copyright 2006-2015 by the Sumatra team, see `doc/authors.txt`

license BSD 2-clause, see `LICENSE` for details.

`sumatra.recordstore.serialization.encode_record` (*record*, *indent=None*)

`sumatra.recordstore.serialization.encode_project_info` (*long_name*, *description*)

Encode a Sumatra project as JSON

`sumatra.recordstore.serialization.build_record` (*data*)

Create a Sumatra record from a nested dictionary.

`sumatra.recordstore.serialization.decode_record` (*content*)

Create a Sumatra record from a JSON string.

`sumatra.recordstore.serialization.decode_records` (*content*)

Create multiple Sumatra records from a JSON string.

Version control

The `versioncontrol` sub-package provides an abstraction layer around different revision/version control systems (VCSs). Only the functionality required for recording version numbers and switching the working copy between different versions is wrapped - for more complex tasks such as merging, branching, etc., the version control tool should be used directly.

Repository objects

A `Repository` object represents a version control repository. Its main roles in Sumatra are

1. to contain the information necessary to identify the repository for reproducibility purposes (i.e. its URL)
2. to provide a uniform interface for obtaining a working copy (“checkout” in Subversion parlance, “clone” for Git/Mercurial)

There are four subclasses of the abstract base `Repository` class:

- `SubversionRepository`,
- `MercurialRepository`,
- `GitRepository`,
- `BazaarRepository`.

These subclasses are only available if the appropriate Python bindings for the underlying VCS are installed - see [Installation](#). Each of the subclasses implements the following interface:

```
class sumatra.versioncontrol.base.Repository (url, upstream=None)
    Represents, and enables limited interaction with, the version control system repository located at url.
    If upstream is not provided, this information will be obtained, if possible, from the version control system.

    url
        The repository URL, generally a local file system path for distributed VCSs.

    upstream
        For distributed VCSs, the repository from which the local repository was cloned.

    checkout (path=u'.')
        Clone a repository (“checkout” in Subversion) from self.url to the local filesystem at path.

    exists
        Does the repository represented by this object actually exist?

    get_working_copy (path=None)
        Return a WorkingCopy object corresponding to a checkout of this repository.

    next ()

    required_attributes = (u'exists', u'checkout', u'get_working_copy')

    vcs_type
```

Working copy objects

`WorkingCopy` objects provide functionality for inspecting the status of a version control working copy (which files have been modified, what version is currently checked out) and to change the version in use (to repeat previous computations, etc.)

There are four subclasses of the abstract base `WorkingCopy` class:

- `SubversionWorkingCopy`,
- `MercurialWorkingCopy`,
- `GitWorkingCopy`,

- `BazaarWorkingCopy`.

Each of these subclasses implements the following interface:

class `sumatra.versioncontrol.base.WorkingCopy` (*path=None*)

Represents, and enables limited interaction with, the version control system working copy located in the *path* directory.

If *path* is not specified, the current working directory is assumed.

For each version control system supported by Sumatra, there is a specific subclass of the abstract `WorkingCopy` base class.

contains (*path*)

Does the repository contain the file with the given path?

where *path* is relative to the working copy root.

current_version ()

Return the version of the current state of the working copy.

diff ()

Return the difference between working copy and repository.

exists

Does the working copy represented by this object actually exist?

get_username ()

Return the username and e-mail of the current user, as understood by the version control system, in the format 'username <e-mail>'.

has_changed ()

Are there any uncommitted changes to the working copy?

next ()

required_attributes = (u'contains', u'current_version', u'use_version', u'use_latest_version', u'status', u'has_c

status ()

Return a dict containing the sets of files that have been modified, added, removed, are missing, not under version control ('unknown'), are being ignored, or are unchanged ('clean').

use_latest_version ()

Switch the working copy to the most recent version.

Any uncommitted changes are retained/merged in.

use_version (*version*)

Switch the working copy to *version*.

If the working copy has uncommitted changes, raises an `UncommittedModificationsError`.

Functions

It is seldom necessary to create a `Repository` or `WorkingCopy` object directly, or even to know which version control system is in use. Instead, the following functions will return the correct object, given a URL or a filesystem path.

`sumatra.versioncontrol.get_repository` (*url*)

Return a `Repository` object which represents, and enables limited interaction with, the version control repository at *url*.

If no repository is found at *url*, raises a `VersionControlError`.

`sumatra.versioncontrol.get_working_copy` (*path=None*)

Return a `WorkingCopy` object which represents, and enables limited interaction with, the version control working copy at *path*.

If *path* is not specified, the current working directory is used. If no working copy is found at *path*, raises a *VersionControlError*.

Exceptions

class `sumatra.versioncontrol.VersionControlError`

class `sumatra.versioncontrol.UncommittedModificationsError`

Formatting output

The formatting module provides classes for formatting simulation/analysis records in different ways: summary, list or table; and in different mark-up formats: currently text or HTML.

copyright Copyright 2006-2015 by the Sumatra team, see doc/authors.txt

license BSD 2-clause, see LICENSE for details.

Formatting Sumatra records

class `sumatra.formatting.TextFormatter` (*records*, *project=None*, *tags=None*)

Bases: `sumatra.formatting.Formatter`

Format the information from a list of Sumatra records as text.

format (*mode=u'short'*)

Format a record according to the given mode. mode may be 'short', 'long' or 'table'.

long (*text_width=80*, *left_column_width=17*)

Return detailed information about a list of records, as text with a limited column width. Lines that are too long will be wrapped round.

name = `u'text'`

next ()

parameter_table ()

Return parameter information about a list of records as text, in a simple tabular format.

required_attributes = (`u'short'`, `u'long'`)

short ()

Return a list of record labels, one per line.

table ()

Return information about a list of records as text, in a simple tabular format.

class `sumatra.formatting.HTMLFormatter` (*records*, *project=None*, *tags=None*)

Bases: `sumatra.formatting.Formatter`

Format information about a group of Sumatra records as HTML fragments, to be included in a larger document.

format (*mode=u'short'*)

Format a record according to the given mode. mode may be 'short', 'long' or 'table'.

long ()

Return detailed information about a list of records as an HTML description list.

name = `u'html'`

next ()

required_attributes = (`u'short'`, `u'long'`)

short ()

Return a list of record labels as an HTML unordered list.

table ()

Return detailed information about a list of records as an HTML table.

`sumatra.formatting.get_formatter(format)`

Return a `Formatter` object of the appropriate type. *format* may be 'text', 'html' or 'textdiff'

Formatting the difference between two records

class `sumatra.formatting.TextDiffFormatter(diff)`

Bases: `sumatra.formatting.Formatter`

Format information about the differences between two Sumatra records in text format.

format (*mode=u'short'*)

Format a record according to the given mode. mode may be 'short', 'long' or 'table'.

long ()

Return a detailed description of the differences between two records.

name = u'textdiff'

next ()

required_attributes = (u'short', u'long')

short ()

Return a summary of the differences between two records.

`sumatra.formatting.get_diff_formatter()`

Return a `DiffFormatter` object of the appropriate type. Only text format is currently available.

Launching programs

The launch module handles launching of simulations/analyses as sub-processes, and obtaining information about the platform(s) on which the simulations are run.

copyright Copyright 2006-2015 by the Sumatra team, see doc/authors.txt

license BSD 2-clause, see LICENSE for details.

class `sumatra.launch.SerialLaunchMode(working_directory=None, options=None)`

Enable running serial computations.

check_files (*executable, main_file*)

generate_command (*executable, main_file, arguments*)

Return a string containing the command to be launched.

get_platform_information ()

Return a list of *PlatformInformation* objects, containing information about the machine(s) and environment(s) the computations are being performed on/in.

name = u'serial'

next ()

pre_run (*executable*)

Run tasks before the simulation/analysis proper.

required_attributes = (u'check_files', u'generate_command')

run (*executable*, *main_file*, *arguments*, *append_label=None*)

Run a computation in a shell, with the given executable, script and arguments. If *append_label* is provided, it is appended to the command line. Return True if the computation finishes successfully, False otherwise.

```
class sumatra.launch.DistributedLaunchMode (n=1, mpirun=u'mpiexec',  
                                             hosts=[], options=None,  
                                             pfi_path=u'/usr/local/bin/pfi.py', working_directory=None)
```

Enable running distributed computations using MPI.

The current implementation is specific to MPICH2, but this will be generalised in future releases.

check_files (*executable*, *main_file*)

generate_command (*executable*, *main_file*, *arguments*)

Return a string containing the command to be launched.

get_platform_information ()

Return a list of *PlatformInformation* objects, containing information about the machine(s) and environment(s) the computations are being performed on/in.

Requires the script `pfi.py` to be placed on the user's path on each node of the machine.

This is currently not useful, as I don't think there is any guarantee that we get the same *n* nodes that the command is run on. Need to look more into this.

name = `u'distributed'`

next ()

pre_run (*executable*)

Run tasks before the simulation/analysis proper.

required_attributes = (`u'check_files'`, `u'generate_command'`)

run (*executable*, *main_file*, *arguments*, *append_label=None*)

Run a computation in a shell, with the given executable, script and arguments. If *append_label* is provided, it is appended to the command line. Return True if the computation finishes successfully, False otherwise.

```
class sumatra.launch.PlatformInformation (**kwargs)
```

A simple container for information about the machine and environment the computations are being performed on/in.

Parameter file formats

The parameters module handles different parameter file formats.

The original idea was that all parameter files will be converted to a single internal parameter format, the NeuroTools ParameterSet class. This will allow fancy searching/comparisons based on parameters. However, we don't do this at the moment, the only methods that are used are *update()* and *save()*

Classes

NTParameterSet: handles parameter files in the NeuroTools parameter set format, based on nested dictionaries.

SimpleParameterSet: handles parameter files in a simple "name = value" format, with no nesting or grouping.

ConfigParserParameterSet handles parameter files in traditional config file format, as parsed by the standard Python `ConfigParser` module.

JSONParameterSet handles parameter files in JSON format

YAMLParameterSet handles parameter files in YAML format

copyright Copyright 2006-2015 by the Sumatra team, see doc/authors.txt

license BSD 2-clause, see LICENSE for details.

Providing information about programs

The programs module handles simulator and analysis programs, i.e. executable files, to support the ability to customize Sumatra's behaviour for specific tools.

Classes

Executable represents a generic executable, about which nothing is known except its name. The base class for specific simulator/analysis tool classes.

PythonExecutable represents the Python interpreter executable.

MatlabExecutable represents the Matlab interpreter executable.

NESTSimulator represents the NEST neuroscience simulator.

NEURONSimulator represents the NEURON neuroscience simulator.

GENESISSimulator represents the GENESIS neuroscience simulator.

RExecutable represents the Rscript CLI to R interpreter executable.

Functions

get_executable() Return an appropriate subclass of Executable, given either the path to an executable file or a script file that can be run with a given tool.

copyright Copyright 2006-2015 by the Sumatra team, see doc/authors.txt

license BSD 2-clause, see LICENSE for details.

class `sumatra.programs.Executable` (*path*, *version=None*, *options=u''*, *name=None*)
Bases: `future.types.newobject.newobject`

name = None

next ()

required_attributes = ('executable_names', 'file_extensions')

requires_script = False

static write_parameters (*parameters*, *filename*)

`sumatra.programs.get_executable` (*path=None*, *script_file=None*)

Given the path to an executable, determine what program it is, if possible. Given the name of a script file, try to infer the program that runs that script. Return an appropriate subclass of Executable

Managing projects

The projects module defines the `Project` class, which stores information about a computation-based project and contains a number of methods for managing and running computational experiments, whether simulations, analyses or whatever. This is the main class that is used directly when using Sumatra within your own scripts.

```
class sumatra.projects.Project (name, default_executable=None, default_repository=None,
                                default_main_file=None, default_launch_mode=None,
                                data_store=u'default', record_store=u'default',
                                on_changed=u'error', description=u'', data_label=None,
                                input_datastore=None, label_generator=u'timestamp',
                                timestamp_format=u'%Y%m%d-%H%M%S', al-
                                low_command_line_parameters=True, plugins=[])

Bases: future.types.newobject.newobject

add_comment (label, comment, replace=False)

add_record (record)
    Add a simulation or analysis record.

add_tag (label, tag)

backup (remove_original=False)
    Create a new backup directory in the same location as the project directory and copy the contents of
    the project directory into the backup directory. Uses _get_project_file to extract the path to the project
    directory.

    Returns
    • backup_dir: the directory used for the backup

change_record_store (new_store)
    Change the record store that is used by this project.

compare (label1, label2, ignore_mimetypes=[], ignore_filenames=[])

data_label

delete_by_tag (tag, delete_data=False)
    Delete all records with a given tag. Return the number of records deleted.

delete_record (label, delete_data=False)
    Delete a record. Return 1 if the record is found. Otherwise return 0.

export ()

find_records (tags=None, reverse=False, *args, **kwargs)

format_records (format=u'text', mode=u'short', tags=None, reverse=False, *args, **kwargs)

get_labels (tags=None, reverse=False)

get_record (label)
    Search for a record with the supplied label and return it if found. Otherwise return None.

info ()
    Show some basic information about the project.

launch (parameters={}, input_data=[], script_args=u'', executable=u'default', reposi-
        tory=u'default', main_file=u'default', version=u'current', launch_mode=u'default',
        label=None, reason=None, timestamp_format=u'default', repeats=None)
    Launch a new simulation or analysis.

load_plugins (*plugins)

most_recent ()

new_record (parameters={}, input_data=[], script_args=u'', executable=u'default', reposi-
            tory=u'default', main_file=u'default', version=u'current', launch_mode=u'default',
            label=None, reason=None, timestamp_format=u'default')

next ()

remove_plugins (*plugins)

remove_tag (label, tag)
```

repeat (*original_label*, *new_label*=None)

save ()

Save state to some form of persistent storage. (file, database).

show_diff (*label1*, *label2*, *mode*=u'short', *ignore_mimetypes*=[], *ignore_filenames*=[])

update_code (*working_copy*, *version*=u'current')

Check if the working copy has modifications and prompt to commit or revert them.

valid_name_pattern = u'(?P<project>\\w+([\\w\\-]*)'

`sumatra.projects.load_project` (*path*=None)

Read project from directory passed as the argument and return Project object. If no argument is given, the project is read from the current directory.

Handling provenance information

The records module defines the *Record* class, which gathers and stores information about an individual simulation or analysis run.

```
class sumatra.records.Record (executable, repository, main_file, version, launch_mode, datastore, parameters={}, input_data=[], script_arguments=u'', label=None, reason=u'', diff=u'', user=u'', on_changed=u'error', input_datastore=None, stdout_stderr=u'Not launched.', timestamp=None, timestamp_format=u'%Y%m%d-%H%M%S')
```

Bases: `future.types.newobject.newobject`

The *Record* class has two main roles: capturing information about the context of a computation, and storing this information for later retrieval.

command_line

Return the command-line string for the computation captured by this record.

delete_data ()

Delete any data files associated with this record.

describe (*format*=u'text', *mode*=u'long')

Return a description of the record.

mode: either 'long' or 'short'

format either 'text' or 'html'

difference (*other_record*, *ignore_mimetypes*=[], *ignore_filenames*=[])

Determine the difference between this computational experiment and another (code, platform, results, etc.).

Return a RecordDifference object.

next ()

register (*working_copy*)

Record information about the environment.

run (*with_label*=False)

Launch the simulation or analysis.

with_label adds the record label either to the parameter file (*with_label*=*"parameters"*) or to the end of the command line (*with_label*=*"cmdline"*), and appends the label to the datastore root. This allows the program being run to create files in a directory specific to this run.

script_content

Return the script content from the main file.

valid_name_pattern = u'(?P<label>\\w+([\\w\\-\\.:/\\s]*)'

```
class sumatra.records.RecordDifference (recordA, recordB, ignore_mimetypes=[], ig-
                                     nore_filenames=[])
    Bases: future.types.newobject.newobject
    Represents the difference between two Record objects.
    code_differs
    dependencies_differ
    dependency_differences
    ignore_filenames = [u'\\.log', u'^log']
    ignore_mimetypes = []
    input_data_differ
    input_data_differences
    launch_mode_differences
    next ()
    output_data_differ
    output_data_differences
    parameter_differences
    recordA_script_content_diff
    recordB_script_content_diff
    script_content_diff (record, other)
```

Frequently asked questions

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 :-)

 (although it has certain similarities to [Madagascar](#)).

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

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. If you think you have found a bug you can post it on the Github [issue tracker](#).

Release notes

Sumatra 0.7.0 release notes

3rd July 2015

Welcome to Sumatra 0.7.0!

This version of Sumatra brings some major improvements for users, including an improved web browser interface, improved support for the R language, Python 3 compatibility, a plug-in interface making Sumatra easier to extend and customize, and support for storing data using WebDAV.

In addition, there have been many changes under the hood, including a move to Github and improvements to the test framework, largely supported by the use of Docker.

Last but not least, we have changed licence from the CeCILL licence (GPL-compatible) to a BSD 2-Clause Licence, which should make it easier for other developers to use Sumatra in their own projects.

Updated and extended web interface

Thanks to Felix Hoffman’s Google Summer of Code project, the web browser interface now provides the option of viewing the history of your project either in a “process-centric” view, as in previous versions, where each row in the table represents a computation, or in a “data-centric” view, where each row is a data file. Where the output from one computation is the input to another, additional links make it possible to follow these connections.

The web interface has also had a cosmetic update and several other improvements, including a more powerful comparison view (see screenshot). Importantly, the interface layout no longer breaks in narrower browser windows.

BSD licence

The Sumatra project aims to provide not only tools for scientists as end users (such as `smt` and `smtweb`), but also library components for developers to add Sumatra’s functionality to their own tools. To support this second use, we have switched licence from CeCILL (GPL-compatible) to the BSD 2-Clause Licence.

MyProject Comparison	
Comparison of 20150526-140238 and 20150526-135236	
Code	
A Version 9f99c53062cf49bb93e5ab0828d0ce897fccc372	B Version bac1db1b9ec5c90fe8aabe59e8a5aab5f56340f
Parameters	
A seed: 65785 filename: Data/example.dat	B seed: 356356546
Output data	
A example.dat 43a47cb379d2a7008fde38c8172278d000f0c4 None 2.4 KB	B example2.dat 030089163fb6bfad3edc172cb28ca027c1368c9 None 2.4 KB

Python 3 support

In version 0.6.0, Sumatra already supported provenance capture for projects using Python 3, but required Python 2.6 or 2.7 to run. Thanks to Tim Tröndle, Sumatra now also runs in Python 3.4.

Plug-in interface

To support the wide diversity of workflows in scientific computing, Sumatra has always had an extensible architecture. It is intended to be easy to add support for new database formats, new programming languages, new version control systems, or new ways of launching computations.

Until now, adding such extensions has required that the code be included in Sumatra's code base. Version 0.7.0 adds a plug-in interface, so you can define your own local extensions, or use other people's.

For more information, see *Extending Sumatra with plug-ins*.

WebDAV support

The option to archive output data files has been extended to allow archiving to a remote server using the WebDAV protocol.

Support for the R language

Sumatra will now attempt to determine the versions of all external packages loaded by an R script.

Other changes

For developers, there has been a significant change - the project has moved from Mercurial to Git, and is now hosted on [Github](#). Testing has also been significantly improved, with more system/integration testing, and the use of [Docker](#) for testing PostgreSQL and WebDAV support.

Parsing of command-line parameters has been improved. The `ParameterSet` classes now have a `diff()` method, making it easier to see the difference between two parameter sets, especially for large and hierarchical sets.

Following the [recommendation of the Mercurial developers](#), and to enable the change of licence to BSD, we no longer use the Mercurial internal API. Instead we use the Mercurial command line interface via the [hgapi](#) package.

Bug fixes

A fair number of bugs have been fixed.

Sumatra 0.6.0 release notes

18th April 2014

Welcome to Sumatra 0.6.0!

This version of Sumatra sees many new features, improvements to existing ones, and bug fixes:

New export formats: LaTeX and bash script

You can now export your project history as a LaTeX document, allowing you to easily generate a PDF listing details of all the simulations or data analyses you've run.

```
$ smt list -l -f latex > myproject.tex
```

Using tags, you can also restrict the output to only a subset of the history.

You can also generate a shell script, which can be executed to repeat the sequence of computations captured by Sumatra, or saved as an executable record of your work. The shell script includes all version control commands needed to ensure the correct version of the code is used at each step.

```
$ smt list -l -f shell > myproject.sh
```

Working with multiple VCS branches

Previous versions of Sumatra would always update to the latest version of the code in your version control repository before running the computation.

Now, “smt run” will never change the working copy, which makes it much easier to work with multiple version control branches and to go back to running earlier versions of your code.

Programs that read from stdin or write to stdout

Programs that read from standard input or write to standard output can now be run with **smt run**. For example, if the program is normally run using:

```
$ myprog < input > output
```

You can run it with Sumatra using:

```
$ smt run -e myprog -i input -o output
```

Improvements to the Web browser interface

There have been a number of small improvements to the browser interface:

- a slightly more compact and easier-to-read (table-like) layout for parameters;
- the working directory is now displayed in the record detail view;
- the interface now works if record label contain spaces or forward slashes.

The minimal Django version needed is now 1.4.

Support for PostgreSQL

The default record store for Sumatra is based on the Django ORM, using SQLite as the backend. It is now possible to use PostgreSQL instead of SQLite, which gives better performance and allows the Sumatra record store to be placed on a separate server.

To set up a new Sumatra project using PostgreSQL, you will first have to create a database using the PostgreSQL tools (**psql**, etc.). You then configure Sumatra as follows:

```
$ smt init --store=postgres://username:password@hostname/databasename MyProject
```

Note that the database tables will not be created until after the first **smt run**.

Integration with the SLURM resource manager

Preliminary support for launching MPI computations via **SLURM** has been added.

```
$ smt configure --launch_mode=slurm-mpi
$ smt run -n 256 input_file1 input_file2
```

This will launch 256 tasks using **salloc** and **mpiexec**.

Command-line options for SLURM can also be set using **smt configure**

```
$ smt configure --launch_mode_options=" --tasks-per-node=1"
```

If you are using **mpiexec** on its own, without a resource manager, you can set MPI command-line options in the same way.

Improvements to the @capture decorator

The `@capture` decorator, which makes it easy to add Sumatra support to Python scripts (as an alternative to using the **smt** command), now captures stdout and stderr.

Improvements to parameter file handling

Where a parameter file has a standard mime type (like json, yaml), Sumatra uses the appropriate extension if rewriting the file (e.g. to add parameters specified on the command line), rather than the generic ".param".

Migrating data files

If you move your input and/or output data files, either within the filesystem on your current computer or to a new computer, you need to tell Sumatra about it so that it can still find your files. For this there is a new command, **smt migrate**. This command also handles changes to the location of the data archive, if you are using one, and changes to the base URL of any mirrored data stores. For usage information, run:

```
$ smt help migrate
```

Miscellaneous improvements

- **smt run** now passes unknown keyword args on to the user program. There is also a new option ‘`–plain`’ which prevents arguments of the form “`x=y`” being interpreted by Sumatra; instead they are passed straight through to the program command line.

- When repeating a computation, the label of the original is now stored in the *repeats* attribute of *Record*, rather than appending “_repeat” to the original label. The new record will get a new unique label, or a label specified by the user. This means a record can be repeated more than once, and is a more reliable method of indicating a repeat.
- A Sumatra project now knows the version of Sumatra with which it was created.
- `smt list` now has an option ‘-r’/‘--reverse’ which lists records oldest to newest.

Bug fixes

A fair number of bugs have been fixed.

Sumatra 0.5.3 release notes

April 9th 2014

Sumatra 0.5.3 is a minor release that adds support for the ISO datetime format and fixes a problem when trying to move a project to another machine.

Sumatra 0.5.1 release notes

March 31st 2013

Sumatra 0.5.1 is a bug-fix release. It adds support for Django 1.5, and fixes the following bugs:

- ticket:162 “Can’t use main files not in current directory when using git”
- ticket:163 “If main file is set but executable is unrecognized, main file isn’t run”
- ticket:164 “Command line JSON parameter files which are not intended as ‘parameter files’ get broken”
- uncaught exception in *have_internet_connection()*

Sumatra 0.5.0 release notes

February 18th 2013

Welcome to Sumatra 0.5.0!

Overview

Sumatra 0.5 development has mostly been devoted to polishing. There were a bunch of small improvements, with contributions from several new contributors. The [Bitbucket](#) pull request workflow seemed to work well for this. The main changes are:

- working directory now captured (as a parameter of `LaunchMode`);
- data differences are now based on content, not name, i.e. henceforth two files with identical content but different names (e.g. because the name contains a timestamp) will evaluate as being the same;
- improved error messages when a required version control wrapper is not installed;
- dependencies now capture the source from which the version was obtained (e.g. repository url);
- YAML-format parameter files are now supported (thanks to Tristan Webb);
- added “upstream” attribute to the `Repository` class, which may contain the URL of the repository from which your local repository was cloned;
- added `MirroredFileSystemDataStore`, which supports the case where files exist both on the local filesystem and on some web server (e.g. `DropBox`);

- the name/e-mail of the user who launched the computation is now captured (first trying `~/ .smt.rc`, then the version control system);
- there is now a choice of methods for auto-generating labels when they are not supplied by the user: timestamp-based (the default and previously the only option) and uuid-based. Use the “-g” option to **smt configure**;
- you can also specify the timestamp format to use (thanks to Yoav Ram);
- improved API reference documentation.

Interfaces to documentation systems

The one big addition to Sumatra is a set of tools to include figures and other results generated by Sumatra-tracked computations in documents, with links to full provenance information: i.e. the full details of the code, input data and computational environment used to generate the figure/result.

The following tools are available:

- for reStructuredText/Sphinx: an “smtlink” role and “smtimage” directive.
- for LaTeX, a “sumatra” package, which provides the “\smtincludegraphics” command.

see *Reproducible publications: including and linking to provenance information in documents* for more details.

Bug fixes

A *handful of bugs* have been fixed.

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.

Authors and contributors

The following people have contributed code to Sumatra. The institutional affiliations are those at the times of the contributions, 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 Hinsén [6]
- Stephan Gabler [7]
- Takafumi Arakaki [8]
- Yoav Ram
- Tristan Webb [9]
- Maximilian Albert [10]
- Daniel Wheeler [11]
- Tim Joseph Dumol
- Julia Evans
- Jaime Ashander [12]
- Tim Tröndle [13]
- Sebastian Spreizer [14]

1. Unité de Neurosciences, 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
9. Warwick University, UK
10. University of Southampton, UK
11. National Institute of Standards and Technology, USA
12. University of California–Davis, USA
13. Younicos AG, Berlin, Germany
14. Bernstein Center Freiburg, University of Freiburg, Freiburg, Germany

If we'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.

Licence

Sumatra is freely available under the BSD 2-clause license.

S

`sumatra.dependency_finder`, [43](#)
`sumatra.formatting`, [50](#)
`sumatra.launch`, [51](#)
`sumatra.parameters`, [52](#)
`sumatra.programs`, [53](#)
`sumatra.recordstore.serialization`, [47](#)
`sumatra.versioncontrol`, [48](#)

A

add_comment() (sumatra.projects.Project method), 54
 add_record() (sumatra.projects.Project method), 54
 add_tag() (sumatra.projects.Project method), 54
 archive_store (sumatra.datastore.ArchivingFileSystemDataStore attribute), 43
 ArchivedDataFile (class in sumatra.datastore.archivingfs), 43
 ArchivingFileSystemDataStore (class in sumatra.datastore), 43

B

backup() (sumatra.projects.Project method), 54
 BaseDependency (class in sumatra.dependency_finder.core), 43
 build_record() (in module sumatra.recordstore.serialization), 47

C

change_record_store() (sumatra.projects.Project method), 54
 check_files() (sumatra.launch.DistributedLaunchMode method), 52
 check_files() (sumatra.launch.SerialLaunchMode method), 51
 checkout() (sumatra.versioncontrol.base.Repository method), 48
 code_differs (sumatra.records.RecordDifference attribute), 56
 command_line (sumatra.records.Record attribute), 55
 compare() (sumatra.projects.Project method), 54
 contains() (sumatra.versioncontrol.base.WorkingCopy method), 49
 contains_path() (sumatra.datastore.base.DataStore method), 42
 copy() (sumatra.datastore.base.DataStore method), 42
 create_project() (sumatra.recordstore.HttpRecordStore method), 47
 current_version() (sumatra.versioncontrol.base.WorkingCopy method), 49

D

data_label (sumatra.projects.Project attribute), 54

DataFile (class in sumatra.datastore.filesystem), 42
 DataItem (class in sumatra.datastore.base), 41
 DataKey (class in sumatra.datastore), 41
 DataStore (class in sumatra.datastore.base), 42
 decode_record() (in module sumatra.recordstore.serialization), 47
 decode_records() (in module sumatra.recordstore.serialization), 47
 delete() (sumatra.datastore.base.DataStore method), 42
 delete() (sumatra.recordstore.base.RecordStore method), 45
 delete_all() (sumatra.recordstore.base.RecordStore method), 46
 delete_by_tag() (sumatra.projects.Project method), 54
 delete_by_tag() (sumatra.recordstore.base.RecordStore method), 46
 delete_data() (sumatra.records.Record method), 55
 delete_record() (sumatra.projects.Project method), 54
 dependencies_differ (sumatra.records.RecordDifference attribute), 56
 dependency_differences (sumatra.records.RecordDifference attribute), 56
 describe() (sumatra.records.Record method), 55
 diff() (sumatra.versioncontrol.base.WorkingCopy method), 49
 difference() (sumatra.records.Record method), 55
 digest (sumatra.datastore.base.DataItem attribute), 41
 digest (sumatra.datastore.DataKey attribute), 41
 DistributedLaunchMode (class in sumatra.launch), 52
 DjangoRecordStore (class in sumatra.recordstore), 47

E

encode_project_info() (in module sumatra.recordstore.serialization), 47
 encode_record() (in module sumatra.recordstore.serialization), 47
 environment variable
 PYTHONPATH, 30
 Executable (class in sumatra.programs), 53
 exists (sumatra.versioncontrol.base.Repository attribute), 48

`exists` (`sumatra.versioncontrol.base.WorkingCopy` attribute), 49
`export()` (`sumatra.projects.Project` method), 54
`export()` (`sumatra.recordstore.base.RecordStore` method), 46
`export_records()` (`sumatra.recordstore.base.RecordStore` method), 46
`extension` (`sumatra.datastore.DataFile` attribute), 43

F

`FileSystemDataStore` (class in `sumatra.datastore`), 42
`find_dependencies()` (in module `sumatra.dependency_finder`), 43
`find_dependencies()` (in module `sumatra.dependency_finder.genesis`), 45
`find_dependencies()` (in module `sumatra.dependency_finder.matlab`), 45
`find_dependencies()` (in module `sumatra.dependency_finder.neuron`), 45
`find_dependencies()` (in module `sumatra.dependency_finder.python`), 44
`find_dependencies()` (in module `sumatra.dependency_finder.r`), 45
`find_file()` (in module `sumatra.dependency_finder.core`), 44
`find_imported_packages()` (in module `sumatra.dependency_finder.python`), 44
`find_included_files()` (in module `sumatra.dependency_finder.genesis`), 45
`find_loaded_files()` (in module `sumatra.dependency_finder.neuron`), 45
`find_new_data()` (`sumatra.datastore.base.DataStore` method), 42
`find_records()` (`sumatra.projects.Project` method), 54
`find_versions()` (in module `sumatra.dependency_finder.core`), 44
`find_versions_by_attribute()` (in module `sumatra.dependency_finder.python`), 44
`find_versions_from_egg()` (in module `sumatra.dependency_finder.python`), 44
`find_versions_from_versioncontrol()` (in module `sumatra.dependency_finder.core`), 44
`find_xopened_files()` (in module `sumatra.dependency_finder.neuron`), 45
`format()` (`sumatra.formatting.HTMLFormatter` method), 50
`format()` (`sumatra.formatting.TextDiffFormatter` method), 51
`format()` (`sumatra.formatting.TextFormatter` method), 50
`format_records()` (`sumatra.projects.Project` method), 54
`full_path` (`sumatra.datastore.DataFile` attribute), 42

G

`generate_command()` (`sumatra.launch.DistributedLaunchMode` method), 52

`generate_command()` (`sumatra.launch.SerialLaunchMode` method), 51
`generate_key()` (`sumatra.datastore.base.DataItem` method), 41
`generate_keys()` (`sumatra.datastore.base.DataStore` method), 42
`get()` (`sumatra.recordstore.base.RecordStore` method), 46
`get_content()` (`sumatra.datastore.base.DataItem` method), 41
`get_content()` (`sumatra.datastore.base.DataStore` method), 42
`get_data_item()` (`sumatra.datastore.base.DataStore` method), 42
`get_diff_formatter()` (in module `sumatra.formatting`), 51
`get_executable()` (in module `sumatra.programs`), 53
`get_formatter()` (in module `sumatra.formatting`), 51
`get_labels()` (`sumatra.projects.Project` method), 54
`get_platform_information()` (`sumatra.launch.DistributedLaunchMode` method), 52
`get_platform_information()` (`sumatra.launch.SerialLaunchMode` method), 51
`get_record()` (`sumatra.projects.Project` method), 54
`get_record_store()` (in module `sumatra.recordstore`), 47
`get_repository()` (in module `sumatra.versioncontrol`), 49
`get_username()` (`sumatra.versioncontrol.base.WorkingCopy` method), 49
`get_working_copy()` (in module `sumatra.versioncontrol`), 49
`get_working_copy()` (`sumatra.versioncontrol.base.Repository` method), 48

H

`has_changed()` (`sumatra.versioncontrol.base.WorkingCopy` method), 49
`has_project()` (`sumatra.recordstore.base.RecordStore` method), 46
`HTMLFormatter` (class in `sumatra.formatting`), 50
`HttpRecordStore` (class in `sumatra.recordstore`), 47

I

`ignore_filenames` (`sumatra.records.RecordDifference` attribute), 56
`ignore_mimetypes` (`sumatra.records.RecordDifference` attribute), 56
`import_()` (`sumatra.recordstore.base.RecordStore` method), 46
`info()` (`sumatra.projects.Project` method), 54
`input_data_differ` (`sumatra.records.RecordDifference` attribute), 56
`input_data_differences` (`sumatra.records.RecordDifference` attribute),

56

L

labels() (sumatra.recordstore.base.RecordStore method), 46

launch() (sumatra.projects.Project method), 54

launch_mode_differences (sumatra.records.RecordDifference attribute), 56

list() (sumatra.recordstore.base.RecordStore method), 46

list_projects() (sumatra.recordstore.base.RecordStore method), 46

load_plugins() (sumatra.projects.Project method), 54

load_project() (in module sumatra.projects), 55

long() (sumatra.formatting.HTMLFormatter method), 50

long() (sumatra.formatting.TextDiffFormatter method), 51

long() (sumatra.formatting.TextFormatter method), 50

M

metadata (sumatra.datastore.DataKey attribute), 41

mimetype (sumatra.datastore.DataFile attribute), 43

mirror_base_url (sumatra.datastore.MirroredFileSystemDataStore attribute), 43

MirroredDataFile (class in sumatra.datastore.mirroredfs), 43

MirroredFileSystemDataStore (class in sumatra.datastore), 43

most_recent() (sumatra.projects.Project method), 54

most_recent() (sumatra.recordstore.base.RecordStore method), 46

N

name (sumatra.datastore.DataFile attribute), 42

name (sumatra.formatting.HTMLFormatter attribute), 50

name (sumatra.formatting.TextDiffFormatter attribute), 51

name (sumatra.formatting.TextFormatter attribute), 50

name (sumatra.launch.DistributedLaunchMode attribute), 52

name (sumatra.launch.SerialLaunchMode attribute), 51

name (sumatra.programs.Executable attribute), 53

new_record() (sumatra.projects.Project method), 54

next() (sumatra.datastore.base.DataItem method), 41

next() (sumatra.datastore.base.DataStore method), 42

next() (sumatra.datastore.DataKey method), 41

next() (sumatra.formatting.HTMLFormatter method), 50

next() (sumatra.formatting.TextDiffFormatter method), 51

next() (sumatra.formatting.TextFormatter method), 50

next() (sumatra.launch.DistributedLaunchMode method), 52

next() (sumatra.launch.SerialLaunchMode method), 51

next() (sumatra.programs.Executable method), 53

next() (sumatra.projects.Project method), 54

next() (sumatra.records.Record method), 55

next() (sumatra.records.RecordDifference method), 56

next() (sumatra.versioncontrol.base.Repository method), 48

next() (sumatra.versioncontrol.base.WorkingCopy method), 49

O

output_data_differ (sumatra.records.RecordDifference attribute), 56

output_data_differences (sumatra.records.RecordDifference attribute), 56

P

parameter_differences (sumatra.records.RecordDifference attribute), 56

parameter_table() (sumatra.formatting.TextFormatter method), 50

path (sumatra.datastore.DataFile attribute), 42

path (sumatra.datastore.DataKey attribute), 41

PlatformInformation (class in sumatra.launch), 52

pre_run() (sumatra.launch.DistributedLaunchMode method), 52

pre_run() (sumatra.launch.SerialLaunchMode method), 51

Project (class in sumatra.projects), 53

project_info() (sumatra.recordstore.HttpRecordStore method), 47

PYTHONPATH, 30

R

Record (class in sumatra.records), 55

recordA_script_content_diff (sumatra.records.RecordDifference attribute), 56

recordB_script_content_diff (sumatra.records.RecordDifference attribute), 56

RecordDifference (class in sumatra.records), 55

RecordStore (class in sumatra.recordstore.base), 45

register() (sumatra.records.Record method), 55

remove_plugins() (sumatra.projects.Project method), 54

remove_tag() (sumatra.projects.Project method), 54

repeat() (sumatra.projects.Project method), 54

Repository (class in sumatra.versioncontrol.base), 48

required_attributes (sumatra.datastore.base.DataStore attribute), 42

required_attributes (sumatra.formatting.HTMLFormatter attribute), 50

required_attributes (sumatra.formatting.TextDiffFormatter attribute), 51

`required_attributes` (`sumatra.formatting.TextFormatter` attribute), 50
`required_attributes` (`sumatra.launch.DistributedLaunchMode` attribute), 52
`required_attributes` (`sumatra.launch.SerialLaunchMode` attribute), 51
`required_attributes` (`sumatra.programs.Executable` attribute), 53
`required_attributes` (`sumatra.recordstore.base.RecordStore` attribute), 46
`required_attributes` (`sumatra.versioncontrol.base.Repository` attribute), 48
`required_attributes` (`sumatra.versioncontrol.base.WorkingCopy` attribute), 49
`requires_script` (`sumatra.programs.Executable` attribute), 53
`root` (`sumatra.datastore.FileSystemDataStore` attribute), 42
`run()` (`sumatra.launch.DistributedLaunchMode` method), 52
`run()` (`sumatra.launch.SerialLaunchMode` method), 51
`run()` (`sumatra.records.Record` method), 55

S

`save()` (`sumatra.projects.Project` method), 55
`save()` (`sumatra.recordstore.base.RecordStore` method), 46
`save_copy()` (`sumatra.datastore.base.DataItem` method), 42
`save_dependencies()` (in module `sumatra.dependency_finder.matlab`), 45
`script_content` (`sumatra.records.Record` attribute), 55
`script_content_diff()` (`sumatra.records.RecordDifference` method), 56
`SerialLaunchMode` (class in `sumatra.launch`), 51
`ShelveRecordStore` (class in `sumatra.recordstore`), 46
`short()` (`sumatra.formatting.HTMLFormatter` method), 50
`short()` (`sumatra.formatting.TextDiffFormatter` method), 51
`short()` (`sumatra.formatting.TextFormatter` method), 50
`show_diff()` (`sumatra.projects.Project` method), 55
`size` (`sumatra.datastore.DataFile` attribute), 42
`sorted_content()` (`sumatra.datastore.base.DataItem` method), 42
`status()` (`sumatra.versioncontrol.base.WorkingCopy` method), 49
`sumatra.dependency_finder` (module), 43
`sumatra.formatting` (module), 50
`sumatra.launch` (module), 51
`sumatra.parameters` (module), 52
`sumatra.programs` (module), 53
`sumatra.recordstore.serialization` (module), 47

`sumatra.versioncontrol` (module), 48
`sync()` (`sumatra.recordstore.base.RecordStore` method), 46
`sync_all()` (`sumatra.recordstore.base.RecordStore` method), 46

T

`table()` (`sumatra.formatting.HTMLFormatter` method), 51
`table()` (`sumatra.formatting.TextFormatter` method), 50
`TextDiffFormatter` (class in `sumatra.formatting`), 51
`TextFormatter` (class in `sumatra.formatting`), 50

U

`UncommittedModificationsError` (class in `sumatra.versioncontrol`), 50
`update()` (`sumatra.recordstore.base.RecordStore` method), 46
`update_code()` (`sumatra.projects.Project` method), 55
`update_project_info()` (`sumatra.recordstore.HttpRecordStore` method), 47
`upstream` (`sumatra.versioncontrol.Repository` attribute), 48
`url` (`sumatra.versioncontrol.Repository` attribute), 48
`use_latest_version()` (`sumatra.versioncontrol.base.WorkingCopy` method), 49
`use_version()` (`sumatra.versioncontrol.base.WorkingCopy` method), 49

V

`valid_name_pattern` (`sumatra.projects.Project` attribute), 55
`valid_name_pattern` (`sumatra.records.Record` attribute), 55
`vcs_type` (`sumatra.versioncontrol.base.Repository` attribute), 48
`VersionControlError` (class in `sumatra.versioncontrol`), 50

W

`WorkingCopy` (class in `sumatra.versioncontrol.base`), 49
`write_parameters()` (`sumatra.programs.Executable` static method), 53