SBpipe documentation

Release 3.20.0

Piero Dalle Pezze and Nicolas Le Novère

Apr 11, 2017

CONTENTS

CHAPTER

ONE

USER MANUAL

Metadata

Contacts: Dr Piero Dalle Pezze (piero.dallepezze AT babraham.ac.uk) and Dr Nicolas Le Novère (lenov AT babraham.ac.uk)

Affiliation: The Babraham Institute, Cambridge, CB22 3AT, UK

Mailing list: sbpipe AT googlegroups.com

Forum: https://groups.google.com/forum/#!forum/sbpipe

Citation:

Dalle Pezze, P and Le Novère, N. (2017) *BMC Systems Biology* **11**:46. SBpipe: a collection of pipelines for automating repetitive simulation and analysis tasks. DOI:10.1186/s12918-017-0423-3 (https://doi.org/10.1186/s12918-017-0423-3)

Introduction

This package contains a collection of pipelines for dynamic modelling of biological systems. It aims to automate common processes and speed up productivity for tasks such as model simulation, single/double parameter scan, and parameter estimation.

Requirements

In order to use SBpipe, the following software must be installed:

- Python 2.7+ or 3.4+ - https://www.python.org/
- R 3.3.0+ - https://cran.r-project.org/

SBpipe can work with the following simulators:

- COPASI 4.19+ - http://copasi.org/ (for model simulation, parameter scan, and parameter estimation)
- Python (directly or as a wrapper to call models coded in any programming language)

If LaTeX/PDF reports are also desired, the following software must also be installed:

- LaTeX 2013

How to install SBpipe on GNU/Linux

Install COPASI

As of 2016, COPASI is not available as a package in GNU/Linux distributions. Users must add the path to COPASI binary files manually editing the GNU/Linux `$HOME/.bashrc` file as follows:

```
# Path to CopasiSE (update this accordingly)
export PATH=$PATH:/path/to/CopasiSE/
```

The correct installation of CopasiSE can be tested with:

```
# Reload the .bashrc file
$ source $HOME/.bashrc

$ CopasiSE -h
COPASI 4.19 (Build 140)
```

Install LaTeX

Users are recommended to install LaTeX/texlive using the package manager of their GNU/Linux distribution. On GNU/Linux Ubuntu machines the following package is required:

```
texlive-latex-base
```

The correct installation of LaTeX can be tested with:

```
$ pdflatex -v
pdfTeX 3.14159265-2.6-1.40.16 (TeX Live 2015/Debian)
kpathsea version 6.2.1
Copyright 2015 Peter Breitenlohner (eTeX)/Han The Thanh (pdfTeX).
```

Preparation of SBpipe

SBpipe can be downloaded from the website or cloned using `git`. To run sbpipe from any shell, users should add 'sbpipe/scripts' to their `PATH` environment variable by adding the following lines to their `$HOME`/.bashrc file:

```
# SBPIPE (update this accordingly)
export PATH=$PATH:/path/to/sbpipe/scripts
```

Now you should reload the .bashrc file to apply the previous edits:

```
# Reload the .bashrc file
$ source $HOME/.bashrc
```

Install Python and R packages

Python and R packages required by SBpipe can be installed via Anaconda3/Miniconda3 (recommended) or using the GNU/Linux distribution package manager. This will be explained in the following two sections.

Install Python and R dependencies via Anaconda3/Miniconda3

Users need to download and install Anaconda3 (https://www.continuum.io/downloads) or Miniconda3 (https://conda.io/miniconda.html).

From a GNU/Linux shell:

```
cd path/to/sbpipe

# install dependencies into isolated environment using Anaconda3/Miniconda3
conda env create --name sbpipe --file environment.yaml

# activate environment. The following line can be
# added to the .bashrc file to skip the activation
# of this environment every time SBpipe is used.
source activate sbpipe
```

Install Python and R dependencies via the distribution package manager

Users can install Python and R using the package manager of their GNU/Linux distribution. Users need to make sure that the package `python-pip` is installed. In most cases, the installation via the package manager will automatically configure the correct environment variables.

The correct installation of Python and R can be tested by running the commands:

```
$ python -V
Python 2.7.12
$ pip -V
pip 8.1.2 from /home/ariel/.local/lib/python2.7/site-packages (python 2.7)

$ R --version
R version 3.2.3 (2015-12-10) -- "Wooden Christmas-Tree"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
```

Users need to install Python and R dependency packages used by SBpipe. Two scripts are provided to perform these tasks automatically.

To install SBpipe Python dependencies on GNU/Linux, run:

```
$ cd path/to/sbpipe
$ ./install_pydeps.py
```

To install SBpipe R dependencies on GNU/Linux, run:

```
$ cd path/to/sbpipe
$ R
# Inside R environment, answer 'y' to install packages locally
> source('install_rdeps.r')
```

NOTE: If R package dependencies must be compiled, it is worth checking that the following additional packages are installed in your machine: `build-essential`, `liblapack-dev`, `libblas-dev`, `libcairo-dev`, `libssl-dev`, `libcurl4-openssl-dev`, and `gfortran`. Other packages might be needed, depending on R dependencies. After installing these packages, `install_rdeps.r` must be executed again.

How to install SBpipe on Windows

Installation of COPASI and LaTeX

Windows users need to install the Windows versions of COPASI and LaTeX MikTeX https://miktex.org/.

Install MINGW

We advise users to install `Git for Windows` https://git-for-windows.github.io/ as a simple Shell (MINGW) running on Windows. Leave the default setting during installation.

Preparation of SBpipe and COPASI with MINGW

Once `Git for Windows` is started, a Shell-like window appears and enables users to run commands. The first step is to clone SBpipe from GitHub using the command:

```
$ git clone https://github.com/pdp10/sbpipe.git
```

We now need to set up the SBpipe environment variable:

```
$ touch .bashrc
$ wordpad .bashrc
```

A Wordpad window should be visible, loading the file `.bashrc` . The following lines must be copied into this file:

```
#!/bin/bash/

# SBPIPE
export PATH=$PATH:~/sbpipe/scripts

# COPASI (update this accordingly. Use \ to escape spaces)
export PATH=/path/to/copasi/bin/:$PATH

# Optional: activate Anaconda3 environment for SBpipe automatically
source activate sbpipe
```

Save the file and close wordpad. Now you should reload the .bashrc file to apply the previous changes:

```
# Reload the .bashrc file
$ source $HOME/.bashrc
```

Install Python and R dependencies via Anaconda3/Miniconda3

Users need to download and install Anaconda3 (https://www.continuum.io/downloads) or Miniconda3 (https://conda.io/miniconda.html).

From a MINGW shell (`Git for Windows`) type:

```
cd path/to/sbpipe

# install dependencies into isolated environment using Anaconda3/Miniconda3
conda env create --name sbpipe --file environment.yaml

# activate environment. The following line can be added to the .bashrc file to␣
↪skip the activation
# of this environment every time SBpipe is used.
source activate sbpipe
```

Complete installation via Anaconda3/Miniconda3 (in progress)

SBpipe can also be installed via Anaconda3/Miniconda3 using the command:

```
# create an environment for SBpipe
$ conda create -n sbpipe_env

# activate the environment
$ source activate sbpipe_env

# install pyaml and colorlog from conda-forge
$ conda install -c conda-forge pyaml colorlog
```

```
# install sbpipe
$ conda install -c pdp10 sbpipe
```

The last command will install sbpipe and its Python/R dependencies automatically. The required latex packages need to be installed separately either using a package manager or using anaconda. The following command works on GNU/Linux or Mac OS X, but not on Windows. Windows users need to install LaTeX MikTeX instead.

```
$ conda install -c pkgw texlive-core texlive-selected
```

Check installation of SBpipe

The correct installation of SBpipe and its dependencies can be checked by running the following commands inside the SBpipe folder. For the correct execution of all tests, LaTeX must be installed.

```
# SBpipe version:
$ sbpipe -V
sbpipe 3.12.0
```

```
# run model simulation using COPASI (see results in tests/copasi_models):
$ cd path/to/sbpipe/tests
$ nosetests test_copasi_sim.py --nocapture --verbose
```

```
# run all tests:
$ nosetests test_suite.py --nocapture --verbose
```

```
# regenerate the manuscript figures (see results in tests/insulin_receptor):
$ nosetests test_suite_manuscript.py --nocapture --verbose
```

How to use SBpipe

Preliminary configuration steps

Pipelines using COPASI

Before using these pipelines, a COPASI model must be configured as follow using COPASI_UI:

pipeline: simulation

- Tick the flag *executable* in the Time Course Task.
- Select a report template for the Time Course Task.
- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv (extensions .txt, .tsv, or .dat are also accepted by SBpipe).

pipelines: single or double parameter scan

- Tick the flag *executable* in the Parameter Scan Task.
- Select a report template for the Parameter Scan Task.
- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv (extensions .txt, .tsv, or .dat are also accepted by SBpipe)

pipeline: parameter estimation

- Tick the flag *executable* in the Parameter Estimation Task.
- Select the report template for the Parameter Estimation Task.

- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv (extensions .txt, .tsv, or .dat are also accepted by SBpipe)

For tasks such as parameter estimation using COPASI, it is recommended to move the data set into the folder `Models/` so that the COPASI model file and its associated experimental data files are stored in the same folder.

Pipelines running Python models

pipelines: model simulation

- The model coded in Python must be functional and invokable via *python* command.
- The program must receive the report file name as input argument (see examples in sbpipe/tests/).
- The program must save the report to file including the *Time* column. Report fields must be separated by TAB, and row names must be discarded.

pipeline: parameter estimation

- The model coded in Python must be functional and invokable via *python* command.
- The program must receive the report file name as input argument (see examples in sbpipe/tests/).
- The program must save the report to file. This includes the objective value as first column column, and the estimated parameters as following columns. Rows are the evaluated functions. Report fields must be separated by TAB, and row names must be discarded.

Python as a wrapper Users can use Python as a wrapper to execute models coded in ANY programming language. The following Python model is essentially a wrapper invoking an R model called `sde_periodic_drift.r`. This Python wrapper and `sde_periodic_drift.r` are stored in the `Models/` folder. The configuration file calls the Python wrapper. This wrapper code must receive the report file name as input argument and forward it to the R script. This R script will run a model and store the results in the received report file name. These data must be stored as described above.

Python wrapper `sde_periodic_drift.py`. This runs `sde_periodic_drift.r`

```
import os
import sys
import subprocess
import shlex

# This is a Python wrapper used to run an R model.
# The R model receives the report_filename as input
# and must add the results to it.

# Retrieve the report file name
report_filename = "sde_periodic_drift.csv"
if len(sys.argv) > 1:
    report_filename = sys.argv[1]

command = 'Rscript --vanilla ' + os.path.join(os.path.dirname(__file__), 'sde_
↪periodic_drift.r') + \
          ' ' + report_filename

# Block until command is finished
subprocess.call(shlex.split(command))
```

Configuration file invoking the Python wrapper `sde_periodic_drift.py`

```
generate_data: True
analyse_data: True
generate_report: True
project_dir: "."
simulator: "Python"
model: "sde_periodic_drift.py"
```

```
cluster: "local"
local_cpus: 7
runs: 14
exp_dataset: ""
plot_exp_dataset: False
xaxis_label: "Time"
yaxis_label: "#"
```

How to run SBpipe

SBpipe is executed via the command *sbpipe*. The syntax for this command and its complete list of options can be retrieved by running *sbpipe -h*. The first step is to create a new project. This can be done with the command:

```
$ sbpipe --create-project project_name
```

This generates the following structure:

```
project_name/
    | - Models/
    | - Results/
    | - (store configuration files here)
```

Models must be stored in the Models/ folder. COPASI data sets used by a model should also be stored in Models. To run SBpipe, users need to create a configuration file for each pipeline they intend to run (see next section). These configuration files should be placed in the root project folder. In Results/ users will eventually find all the results generated by SBpipe.

Each pipeline is invoked using a specific option (type `sbpipe -h` for the complete command set):

```
# runs model simulation.
sbpipe -s config_file.yaml

# runs parameter estimation.
sbpipe -e config_file.yaml

# runs single parameter scan.
sbpipe -p config_file.yaml

# runs double parameter scan
sbpipe -d config_file.yaml
```

Pipeline configuration files

Pipelines are configured using files (here called configuration files). These files are YAML files. In SBpipe each pipeline executes three tasks: data generation, data analysis, and report generation. These tasks can be activated in each configuration files using the options:

- generate_data: True
- analyse_data: True
- generate_report: True

The `generate_data` task runs a simulator accordingly to the options in the configuration file. Hence, this task collects and organises the reports generated from the simulator. The `analyse_data` task processes the reports to generate plots and compute statistics. Finally, the `generate_report` task generates a LaTeX report containing the computed plots and invokes the utility `pdflatex` to produce a PDF file. This modularisation allows users to analyse the same data without having to re-generate it, or to skip the report generation if not wanted.

Pipelines for parameter estimation or stochastic model simulation can be computationally intensive. SBpipe allows users to generate simulated data in parallel using the following options in the pipeline configuration file:

- cluster: "local"
- local_cpus: 7
- runs: 250

The `cluster` option defines whether the simulator should be executed locally (`local`: Python multiprocessing), or in a computer cluster (`sge`: Sun Grid Engine (SGE), `lsf`: Load Sharing Facility (LSF)). If `local` is selected, the `local_cpus` option determines the maximum number of CPUs to be allocated for local simulations. The `runs` option specifies the number of simulations (or parameter estimations for the pipeline `param_estim`) to be run.

Assuming that the configuration files are placed in the root directory of a certain project (e.g. project_name/), examples are given as follow:

Example 1: configuration file for the pipeline *simulation*

```
# True if data should be generated, False otherwise
generate_data: True
# True if data should be analysed, False otherwise
analyse_data: True
# True if a report should be generated, False otherwise
generate_report: True
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi, Rscript, Python, Java)
simulator: "Copasi"
# The model name
model: "insulin_receptor_stoch.cps"
# The cluster type. local if the model is run locally,
# sge/lsf if run on cluster.
cluster: "local"
# The number of CPU if local is used, ignored otherwise
local_cpus: 7
# The number of simulations to perform.
# n>: 1 for stochastic simulations.
runs: 40
# An experimental data set (or blank) to add to the
# simulated plots as additional layer
exp_dataset: "insulin_receptor_dataset.csv"
# True if the experimental data set should be plotted.
plot_exp_dataset: True
# The label for the x axis.
xaxis_label: "Time [min]"
# The label for the y axis.
yaxis_label: "Level [a.u.]"
```

Example 2: configuration file for the pipeline *single parameter scan*

```
# True if data should be generated, False otherwise
generate_data: True
# True if data should be analysed, False otherwise
analyse_data: True
# True if a report should be generated, False otherwise
generate_report: True
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi)
simulator: "Copasi"
# The model name
model: "insulin_receptor_inhib_scan_IR_beta.cps"
# The variable to scan (as set in Copasi Parameter Scan Task)
scanned_par: "IR_beta"
# The cluster type. local if the model is run locally,
# sge/lsf if run on cluster.
```

```
cluster: "local"
# The number of CPU if local is used, ignored otherwise
local_cpus: 7
# The number of simulations to perform per run.
# n>: 1 for stochastic simulations.
runs: 1
# The number of intervals in the simulation
simulate__intervals: 100
# True if the variable is only reduced (knock down), False otherwise.
ps1_knock_down_only: True
# True if the scanning represents percent levels.
ps1_percent_levels: True
# The minimum level (as set in Copasi Parameter Scan Task)
min_level: 0
# The maximum level (as set in Copasi Parameter Scan Task)
max_level: 100
# The number of scans (as set in Copasi Parameter Scan Task)
levels_number: 10
# True if plot lines are the same between scans
# (e.g. full lines, same colour)
homogeneous_lines: False
# The label for the x axis.
xaxis_label: "Time [min]"
# The label for the y axis.
yaxis_label: "Level [a.u.]"
```

Example 3: configuration file for the pipeline *double parameter scan*

```
# True if data should be generated, False otherwise
generate_data: True
# True if data should be analysed, False otherwise
analyse_data: True
# True if a report should be generated, False otherwise
generate_report: True
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi)
simulator: "Copasi"
# The model name
model: "insulin_receptor_inhib_dbl_scan_InsulinPercent__IRbetaPercent.cps"
# The 1st variable to scan (as set in Copasi Parameter Scan Task)
scanned_par1: "InsulinPercent"
# The 2nd variable to scan (as set in Copasi Parameter Scan Task)
scanned_par2: "IRbetaPercent"
# The cluster type. local if the model is run locally,
# sge/lsf if run on cluster.
cluster: "local"
# The number of CPU if local is used, ignored otherwise
local_cpus: 7
# The number of simulations to perform.
# n>: 1 for stochastic simulations.
runs: 1
# The simulation length (as set in Copasi Time Course Task)
sim_length: 10
```

Example 4: configuration file for the pipeline *parameter estimation*

```
# True if data should be generated, False otherwise
generate_data: True
# True if data should be analysed, False otherwise
analyse_data: True
# True if a report should be generated, False otherwise
generate_report: True
```

```
# True if a zipped tarball should be generated, False otherwise
generate_tarball: True
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi)
simulator: "Copasi"
# The model name
model: "insulin_receptor_param_estim.cps"
# The cluster type. local if the model is run locally,
# sge/lsf if run on cluster.
cluster: "local"
# The number of CPU if local is used, ignored otherwise
local_cpus: 7
# The parameter estimation round which is used to distinguish
# phases of parameter estimations when parameters cannot be
# estimated at the same time
round: 1
# The number of parameter estimations
# (the length of the fit sequence)
runs: 250
# The threshold percentage of the best fits to consider
best_fits_percent: 75
# The number of available data points
data_point_num: 33
# True if 2D all fits plots for 66% confidence levels
# should be plotted. This can be computationally expensive.
plot_2d_66cl_corr: True
# True if 2D all fits plots for 95% confidence levels
# should be plotted. This can be computationally expensive.
plot_2d_95cl_corr: True
# True if 2D all fits plots for 99% confidence levels
# should be plotted. This can be computationally expensive.
plot_2d_99cl_corr: True
# True if parameter values should be plotted in log space.
logspace: True
# True if plot axis labels should be plotted in scientific notation.
scientific_notation: True
```

Additional examples of configuration files can be found in:

```
sbpipe/tests/insulin_receptor/
```

How to run SBpipe with Snakemake (in progress)

SBpipe can also be executed using Snakemake (https://snakemake.readthedocs.io). Snakemake offers an infrastructure for running software pipelines using declarative rules.

Snakemake can be installed manually via package manager or using the conda command:

```
# As of April 2017, Snakemake requires python 3.5 to run
$ conda install -c conda-forge python=3.5

# Install snakemake
$ conda install -c bioconda snakemake
```

The SBpipe pipelines for parameter estimation, single/double parameter scan, and model simulation are also implemented as snakemake files (which contain the set of rules for each pipeline). These are:

- sbpipe_pe.snake
- sbpipe_ps1.snake

- sbpipe_ps2.snake
- sbpipe_sim.snake

and are stored on the root folder of SBpipe. The advantage of using snakemake as pipeline infrastructure is that it offers an extended command sets compared to the one provided with the standard sbpipe. For details, run

```
snakemake -h
```

Snakemake also offers a strong support for dependency management at coding level and reentrancy at execution level. The former is defined as a way to precisely define the dependency order of functions. The latter is the capacity of a program to continue from the last interrupted task. Benefitting of dependency declaration and execution reentrancy can be beneficial for running SBpipe on clusters or on the cloud.

Under the current implementation of SBpipe snakefile, the configuration files described above require the additional field:

```
# The name of the report variables
report_variables: ['IR_beta_pY1146']
```

which contain the names of the variables exported by the simulator. For the parameter estimation pipeline, `report_variables` will contain the names of the estimated parameters.

For the parameter estimation pipeline, the following option must also be added:

```
# An experimental data set (or blank) to add to the
# simulated plots as additional layer
exp_dataset: "insulin_receptor_dataset.csv"
```

A complete example of configuration file for the parameter estimation pipeline is the following:

```
simulator: "Copasi"
model: "insulin_receptor_param_estim.cps"
round: 1
runs: 4
best_fits_percent: 75
data_point_num: 33
plot_2d_66cl_corr: True
plot_2d_95cl_corr: True
plot_2d_99cl_corr: True
logspace: True
scientific_notation: True
report_variables: ['k1','k2','k3']
exp_dataset: "insulin_receptor_dataset.csv"
```

NOTE: As it can be noticed, a configuration files for SBpipe using snakemake requires less options than the corresponding configuration file using SBpipe directly. This because Snakemake files is more automated than SBpipe. Nevertheless, the removal of those additional options is not necessary for running the configuration file using Snakemake.

Examples of configuration files for running SBpipe using Snakemake are in `tests/snakemake`.

Examples of commands running SBpipe pipelines using Snakemake are:

```
# run model simulation
$ snakemake -s path/to/sbpipe/sbpipe_sim.snake --configfile SBPIPE_CONFIG_FILE.
↪yaml --cores 7

# run model parameter estimation using 40 jobs
$ snakemake -s path/to/sbpipe/sbpipe_pe.snake --configfile SBPIPE_CONFIG_FILE.yaml␣
↪-p -j 40 --verbose --cluster "qsub"

# run model parameter parameter scan using 5 jobs
$ snakemake -s path/to/sbpipe/sbpipe_ps1.snake --configfile SBPIPE_CONFIG_FILE.
↪yaml -p -j 5 --verbose --cluster "bsub"
```

```
# run model parameter parameter scan using 5 jobs
$ snakemake -s path/to/sbpipe/sbpipe_ps2.snake --configfile SBPIPE_CONFIG_FILE.
↪yaml -p -j 1 --verbose --cluster "qsub"
```

See `snakemake -h` for a complete list of commands.

How to report bugs or request new features

SBpipe is a relatively young project and there is a chance that some error occurs. The following mailing list should be used for general questions:

```
sbpipe AT googlegroups.com
```

All the topics discussed in this mailing list are also available at the website:

https://groups.google.com/forum/#!forum/sbpipe

To help us better identify and reproduce your problem, some technical information is needed. This detail data can be found in SBpipe log files which are stored in ${HOME}/.sbpipe/logs/. When using the mailing list above, it would be worth providing this extra information.

Issues and feature requests can also be notified using the github issue tracking system for SBpipe at the web page:

https://github.com/pdp10/sbpipe/issues.

CHAPTER

TWO

DEVELOPER MANUAL

Mailing list: sbpipe AT googlegroups.com

Forum: https://groups.google.com/forum/#!forum/sbpipe

Introduction

This guide is meant for developers and contains guidelines for developing this project.

Development model

This project follows the Feature-Branching model. Briefly, there are two main branches: `master` and `develop`. The former contains the history of stable releases, the latter contains the history of development. The `master` branch contains checkout points for production hotfixes or merge points for release-x.x.x branches. The `develop` branch is used for feature-bugfix integration and checkout point in development. Nobody should directly develop in here.

Conventions

To manage the project in a more consistent way, here is a list of conventions to follow:

- Each new feature is developed in a separate branch forked from *develop*. This new branch is called *featureNUMBER*, where *NUMBER* is the number of the GitHub Issue discussing that feature. The first line of each commit message for this branch should contain the string *Issue #NUMBER* at the beginning. Doing so, the commit is automatically recorded by the Issue Tracking System for that specific Issue. Note that the sharp (#) symbol is required.
- The same for each new bugfix, but in this case the branch name is called bugfixNUMBER.
- The same for each new hotfix, but in this case the branch name is called hotfixNUMBER and is forked from *master*.

Work flow

The procedure for checking out a new feature from the `develop` branch is:

```
$ git checkout -b feature10 develop
```

This creates the `feature10` branch off `develop`. This feature10 is discussed in *Issue #10* in GitHub. When you are ready to commit your work, run:

```
$ git commit -am "Issue #10, summary of the changes. Detailed
description of the changes, if any."
$ git push origin feature10       # sometimes and at the end.
```

As of June 2016, the branches `master` and `develop` are protected and a status check using Travis-CI must be performed before merging or pushing into these branches. This automatically forces a merge without fast-forward. In order to merge **any** new feature, bugfix or simple edits into `master` or `develop`, a developer **must** checkout a new branch and, once committed and pushed, **merge** it to `master` or `develop` using a `pull request`. To merge `feature10` to `develop`, the pull request output will look like this in GitHub Pull Requests:

```
base:develop  compare:feature10   Able to merge. These branches can be
automatically merged.
```

A small discussion about feature10 should also be included to allow other users to understand the feature.

Finally delete the branch:

```
$ git branch -d feature10      # delete the branch feature10 (locally)
```

New releases

When the `develop` branch includes all the desired feature for a release, it is time to checkout this branch in a new one called `release-x.x.x`. It is at this stage that a version is established. Only bugfixes or hotfixes are applied to this branch. When this testing/correction phase is completed, the `master` branch will merge with the `release-x.x.x` branch, using the commands above. To record the release add a tag:

```
git tag -a v1.3 -m "PROGRAM_NAME v1.3"
```

To transfer the tag to the remote server:

```
git push origin v1.3   # Note: it goes in a separate 'branch'
```

To see all the releases:

```
git show
```

How to build SBpipe conda package

This is a short guide for building SBpipe as a conda package. Anaconda (or Miniconda) must be installed. In order to proceed, the package `conda-build` must be installed:

```
$ conda install conda-build

# DON'T FORGET TO SET THIS so that your built package is not uploaded automatically
$ conda config --set anaconda_upload no
```

The recipe for SBpipe is already prepared (file: `meta.yaml`). To create the conda package for SBpipe:

```
$ cd path/to/sbpipe
$ conda build conda_recipe/meta.yaml
```

To test this package locally:

```
# install
$ conda install sbpipe --use-local

# uninstall
$ conda remove sbpipe
```

To upload the package to the Anaconda cloud repository:

```
anaconda upload ~/miniconda/conda-bld/noarch/sbpipe-x.x.x-py_y.tar.bz2
```

Package structure

This section presents the structure of the SBpipe package. The root of the project contains general management scripts for installing Python and R dependencies (install_pydeps.py and install_rdeps.r), and installing SBpipe (setup.py). Additionally, the logging configuration file (logging_config.ini) is also at this level.

In order to automatically compile and run the test suite, Travis-CI is used and configured accordingly (.travis.yml).

The project is structured as follows:

```
sbpipe:
  | - docs/
  | - sbpipe/
        | - R
        | - pl
        | - report
        | - simul
        | - tasks
        | - utils
  | - scripts/
  | - tests/
```

These folders will be discussed in the next sections. In SBpipe, Python is the project main language. Instead, R is essentially used for computing statistics (see section configuration file in the user manual) and for generating plots. This choice allows users to run these scripts independently of SBpipe if needed using an R environment like Rstudio. This can be convenient if further data analysis are needed or plots need to be annotated or edited.

docs

The folder `docs/` contains the documentation for this project. The user and developer manuals in markdown format are contained in `docs/source`. In order to generate the complete documentation for SBpipe, the following packages must be installed:

- python-sphinx
- pandoc
- texlive-fonts-recommended
- texlive-latex-extra

By default the documentation is generated in html and LaTeX/PDF. Instruction for generating or cleaning SBpipe documentation are provided below.

To generate the source code documentation:

```
$ cd path/to/sbpipe/docs
$ ./gen_doc.sh
```

To clean the documentation:

```
$ cd path/to/sbpipe/docs
$ ./cleanup_doc.sh
```

The complete source code documentation for this project is stored in `docs/build/html` (html format) and `docs/build/latex` (LaTeX/PDF format). A shortcut to the documentation in html format is available at the page `docs/index.html`.

sbpipe

This folder contains the source code of the project SBpipe. At this level a file called `__main__.py` enables users to run SBpipe programmatically as a Python module via the command:

```
$ python sbpipe
```

Alternatively `sbpipe` can programmatically be imported within a Python environment as shown below:

```
$ cd path/to/sbpipe
$ python
# Python environment
>>> from sbpipe.main import sbpipe
>>> sbpipe(simulate="my_model.yaml")
```

The following subsections describe sbpipe subpackages.

R

This folder contains a collection of R utility methods for plotting and generating statistics. These utilities are used by the pipelines during data analysis.

pl

The subpackage `sbpipe.pl` contains the class `Pipeline` in the file `pipeline.py`. This class represents a generic pipeline which is extended by SBpipe pipelines. These are organised in the following subpackages:

- `create`: creates a new project
- `ps1`: scan a model parameter, generate plots and report;
- `ps2`: scan two model parameters, generate plots and report;
- `pe`: generate a parameter fit sequence, tables of statistics, plots and report;
- `sim`: generate deterministic or stochastic model simulations, plots and report.

All these pipelines can be invoked directly via the script `sbpipe/scripts/sbpipe`. Each SBpipe pipeline extends the class `Pipeline` and therefore must implement the following methods:

```
# executes a pipeline
def run(self, config_file)

# process the dictionary of the configuration file loaded by Pipeline.load()
def parse(self, config_dict)
```

- The method run() can invoke Pipeline.load() to load the YAML config_file as a dictionary. Once the configuration is loaded and the parameters are imported, run() executes the pipeline.
- The method parse() parses the dictionary and collects the values.

report

The subpackage `sbpipe.report` contains Python modules for generating LaTeX/PDF reports.

simul

The subpackage `sbpipe.simul` contains the class `Simul` in the file `simul.py`. This is a generic simulator interface used by the pipelines in SBpipe. This mechanism uncouples pipelines from specific simulators which can therefore be configured in each pipeline configuration file. As of 2016, the following simulators are available in SBpipe:

- `Copasi`, package `sbpipe.simul.copasi`, which implements all the methods of the class `Simul`;
- `Python`, package `sbpipe.simul.python`.

Pipelines can dynamically load a simulator via the class method `Pipeline.get_simul_obj(simulator)`. This method instantiates an object of subtype `Simul` by refractoring the simulator name as parameter. A simulator class (e.g. `Copasi`) must have the same name of their package (e.g. `copasi`) but start with an upper case letter. A simulator class must be contained in a file with the same name of their package (e.g. `copasi`). Therefore, for each simulator package, exactly one simulator class can be instantiated. Simulators can be configured in the configuration file using the field `simulator`.

tasks

The subpackage `sbpipe.tasks` contains the Python scripts to invoke the single SBpipe tasks. These are invoked by the rules in the SBpipe snakemake files. These snakemake files are:

- sbpipe_pe.snake
- sbpipe_ps1.snake
- sbpipe_ps2.snake
- sbpipe_sim.snake

and are stored on the root folder of SBpipe.

utils

The subpackage `sbpipe.utils` contains a collection of Python utility modules which are used by sbpipe. Here are also contained the functions for running commands in parallel.

scripts

The folder `scripts` contains the scripts: `cleanup_sbpipe` and `sbpipe`. `sbpipe` is the main script and is used to run the pipelines. `cleanup_sbpipe.py` is used for cleaning the package including the test results.

tests

The package `tests` contains the script `test_suite.py` which executes all sbpipe tests. It should be used for testing the correct installation of SBpipe dependencies as well as reference for configuring a project before running any pipeline. Projects inside the folder `sbpipe/tests/` have the SBpipe project structure:

- `Models`: (e.g. models, COPASI models, Python models, data sets directly used by Copasi models);
- `Results`: (e.g. pipelines results, etc).

Examples of configuration files (*.yaml) using COPASI can be found in sbpipe/tests/insulin_receptor/.

To run tests for Python models, the Python packages `numpy`, `scipy`, and `pandas` must be installed. In principle, users may define their Python models using arbitrary packages.

As of 2016, the repository for SBpipe source code is `github.com`. This is configured to run Travis-CI every time a `git push` into the repository is performed. The exact details of execution of Travis-CI can be found in Travis-CI configuration file `sbpipe/.travis.yml`. Importantly, Travis-CI runs all SBpipe tests using `nosetests`.

Miscellaneous of useful commands

Git

Startup

```
# clone master
$ git clone https://github.com/pdp10/sbpipe.git
# get develop branch
$ git checkout -b develop origin/develop
# to get all the other branches
$ for b in `git branch -r | grep -v -- '->'`; do git branch
--track ${b##origin/} $b; done
# to update all the branches with remote
$ git fetch --all
```

Update

```
# ONLY use --rebase for private branches. Never use it for shared
# branches otherwise it breaks the history. --rebase moves your
# commits ahead. For shared branches, you should use
# `git fetch && git merge --no-ff`
$ git pull [--rebase] origin BRANCH
```

Managing tags

```
# Update an existing tag to include the last commits
# Assuming that you are in the branch associated to the tag to update:
git tag -f -a tagName
# push your new commit:
git push
# force push your moved tag:
git push -f --tags

# rename a tag
git tag new old
git tag -d old
git push origin :refs/tags/old
git push --tags
# make sure that the other users remove the deleted tag. Tell them(co-workers) to␣
↪run the following command:
git pull --prune --tags


# removing a tag remotely and locally
git push --delete origin tagName
git tag -d tagName
```

File system

```
$ git rm [--cache] filename
$ git add filename
```

Information

```
$ git status
$ git log [--stat]
$ git branch       # list the branches
```

Maintenance

```
$ git fsck      # check errors
$ git gc        # clean up
```

Rename a branch locally and remotely

```
git branch -m old_branch new_branch         # Rename branch locally
git push origin :old_branch                 # Delete the old branch
```

```
git push --set-upstream origin new_branch    # Push the new branch, set
local branch to track the new remote
```

Reset

```
git reset --hard HEAD    # to undo all the local uncommitted changes
```

Syncing a fork (assuming upstreams are set)

```
git fetch upstream
git checkout develop
git merge upstream/develop
```

CHAPTER

THREE

SOURCE CODE

Python modules

sbpipe package

Subpackages

sbpipe.R package

Subpackages

sbpipe.R.misc package

Module contents

Module contents

sbpipe.pl package

Subpackages

sbpipe.pl.create package

Submodules

sbpipe.pl.create.newproj module

class sbpipe.pl.create.newproj.**NewProj**(*models_folder='Models'*, *working_folder='Results'*)

Bases: *sbpipe.pl.pipeline.Pipeline* (page 27)

This module initialises the folder tree for a new project.

Parameters

- **models_folder** – the folder containing the models
- **working_folder** – the folder to store the results

run(*project_name*)

Create a project directory tree.

Parameters **project_name** – the name of the project

Returns 0

Module contents

sbpipe.pl.pe package

Submodules

sbpipe.pl.pe.parest module

class sbpipe.pl.pe.parest.**ParEst**(*models_folder='Models'*, *working_folder='Results'*, *sim_data_folder='param_estim_data'*, *sim_plots_folder='param_estim_plots'*)

Bases: *sbpipe.pl.pipeline.Pipeline* (page 27)

This module provides the user with a complete pipeline of scripts for running model parameter estimations

classmethod analyse_data(*simulator*, *model*, *inputdir*, *outputdir*, *fileout_final_estims*, *fileout_all_estims*, *fileout_param_estim_details*, *fileout_param_estim_summary*, *sim_plots_dir*, *best_fits_percent*, *data_point_num*, *cluster='local'*, *plot_2d_66cl_corr=False*, *plot_2d_95cl_corr=False*, *plot_2d_99cl_corr=False*, *logspace=True*, *scientific_notation=True*)

The second pipeline step: data analysis.

Parameters

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model name
- **inputdir** – the directory containing the simulation data
- **outputdir** – the directory to store the results
- **fileout_final_estims** – the name of the file containing final parameter sets with the objective value
- **fileout_all_estims** – the name of the file containing all the parameter sets with the objective value
- **fileout_param_estim_details** – the name of the file containing the detailed statistics for the estimated parameters
- **fileout_param_estim_summary** – the name of the file containing the summary for the parameter estimation
- **sim_plots_dir** – the directory of the simulation plots
- **best_fits_percent** – the percent to consider for the best fits
- **data_point_num** – the number of data points
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **plot_2d_66cl_corr** – True if 2 dim plots for the parameter sets within 66% should be plotted
- **plot_2d_95cl_corr** – True if 2 dim plots for the parameter sets within 95% should be plotted
- **plot_2d_99cl_corr** – True if 2 dim plots for the parameter sets within 99% should be plotted
- **logspace** – True if parameters should be plotted in log space
- **scientific_notation** – True if axis labels should be plotted in scientific notation

Returns True if the task was completed successfully, False otherwise.

classmethod **generate_data**(*simulator*, *model*, *inputdir*, *cluster*, *local_cpus*, *runs*, *outputdir*, *sim_data_dir*)

The first pipeline step: data generation.

Parameters

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model to process
- **inputdir** – the directory containing the model
- **cluster** – local, lsf for load sharing facility, sge for sun grid engine
- **local_cpus** – the number of cpu
- **runs** – the number of fits to perform
- **outputdir** – the directory to store the results
- **sim_data_dir** – the directory containing the simulation data sets

Returns True if the task was completed successfully, False otherwise.

classmethod **generate_report**(*model*, *outputdir*, *sim_plots_folder*)

The third pipeline step: report generation.

Parameters

- **model** – the model name
- **outputdir** – the directory to store the report
- **sim_plots_folder** – the folder containing the plots

Returns True if the task was completed successfully, False otherwise.

parse(*my_dict*)

run(*config_file*)

Module contents

sbpipe.pl.ps1 package

Submodules

sbpipe.pl.ps1.parscan1 module

class sbpipe.pl.ps1.parscan1.**ParScan1**(*models_folder='Models'*, *working_folder='Results'*, *sim_data_folder='single_param_scan_data'*, *sim_plots_folder='single_param_scan_plots'*)

Bases: *sbpipe.pl.pipeline.Pipeline* (page 27)

This module provides the user with a complete pipeline of scripts for computing single parameter scans.

classmethod **analyse_data**(*model*, *knock_down_only*, *outputdir*, *sim_data_folder*, *sim_plots_folder*, *runs*, *local_cpus*, *percent_levels*, *min_level*, *max_level*, *levels_number*, *homogeneous_lines*, *cluster='local'*, *xaxis_label=''*, *yaxis_label=''*)

The second pipeline step: data analysis.

Parameters

- **model** – the model name
- **knock_down_only** – True for knock down simulation, false if also scanning over expression.

- **outputdir** – the directory containing the results
- **sim_data_folder** – the folder containing the simulated data sets
- **sim_plots_folder** – the folder containing the generated plots
- **runs** – the number of simulations
- **local_cpus** – the number of cpus
- **percent_levels** – True if the levels are percents.
- **min_level** – the minimum level
- **max_level** – the maximum level
- **levels_number** – the number of levels
- **homogeneous_lines** – True if generated line style should be homogeneous
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **xaxis_label** – the name of the x axis (e.g. Time [min])
- **yaxis_label** – the name of the y axis (e.g. Level [a.u.])

Returns True if the task was completed successfully, False otherwise.

classmethod generate_data(*simulator*, *model*, *scanned_par*, *cluster*, *local_cpus*, *runs*, *simulate_intervals*, *single_param_scan_intervals*, *inputdir*, *outputdir*)

The first pipeline step: data generation.

Parameters

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model to process
- **scanned_par** – the scanned parameter
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPU.
- **runs** – the number of model simulation
- **simulate_intervals** – the time step of each simulation
- **single_param_scan_intervals** – the number of scans to perform
- **inputdir** – the directory containing the model
- **outputdir** – the directory to store the results

Returns True if the task was completed successfully, False otherwise.

classmethod generate_report(*model*, *scanned_par*, *outputdir*, *sim_plots_folder*)

The third pipeline step: report generation.

Parameters

- **model** – the model name
- **scanned_par** – the scanned parameter
- **outputdir** – the directory containing the report
- **sim_plots_folder** – the folder containing the plots

Returns True if the task was completed successfully, False otherwise.

parse(*my_dict*)

run(*config_file*)

Module contents

sbpipe.pl.ps2 package

Submodules

sbpipe.pl.ps2.parscan2 module

class sbpipe.pl.ps2.parscan2.**ParScan2**(*models_folder='Models'*, *working_folder='Results'*, *sim_data_folder='double_param_scan_data'*, *sim_plots_folder='double_param_scan_plots'*)

Bases: *sbpipe.pl.pipeline.Pipeline* (page 27)

This module provides the user with a complete pipeline of scripts for computing double parameter scans.

classmethod analyse_data(*model*, *scanned_par1*, *scanned_par2*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*)

The second pipeline step: data analysis.

Parameters

- **model** – the model name
- **scanned_par1** – the first scanned parameter
- **scanned_par2** – the second scanned parameter
- **inputdir** – the directory containing the simulated data sets to process
- **outputdir** – the directory to store the performed analysis
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPU.
- **runs** – the number of model simulation

Returns True if the task was completed successfully, False otherwise.

classmethod generate_data(*simulator*, *model*, *sim_length*, *inputdir*, *outputdir*, *cluster*, *local_cpus*, *runs*)

The first pipeline step: data generation.

Parameters

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model to process
- **sim_length** – the length of the simulation
- **inputdir** – the directory containing the model
- **outputdir** – the directory to store the results
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPU.
- **runs** – the number of model simulation

Returns True if the task was completed successfully, False otherwise.

classmethod generate_report(*model*, *scanned_par1*, *scanned_par2*, *outputdir*, *sim_plots_folder*)

The third pipeline step: report generation.

Parameters

- **model** – the model name

- **scanned_par1** – the first scanned parameter
- **scanned_par2** – the second scanned parameter
- **outputdir** – the directory containing the report
- **sim_plots_folder** – the folder containing the plots.

Returns True if the task was completed successfully, False otherwise.

parse(*my_dict*)

run(*config_file*)

Module contents

sbpipe.pl.sim package

Submodules

sbpipe.pl.sim.sim module

class sbpipe.pl.sim.sim.**Sim**(*models_folder='Models'*, *working_folder='Results'*, *sim_data_folder='simulate_data'*, *sim_plots_folder='simulate_plots'*)

Bases: *sbpipe.pl.pipeline.Pipeline* (page 27)

This module provides the user with a complete pipeline of scripts for running model simulations

classmethod analyse_data(*model*, *inputdir*, *outputdir*, *sim_plots_dir*, *exp_dataset*, *plot_exp_dataset*, *cluster='local'*, *xaxis_label=''*, *yaxis_label=''*)

The second pipeline step: data analysis.

Parameters

- **model** – the model name
- **inputdir** – the directory containing the data to analyse
- **outputdir** – the output directory containing the results
- **sim_plots_dir** – the directory to save the plots
- **exp_dataset** – the full path of the experimental data set
- **plot_exp_dataset** – True if the experimental data set should also be plotted
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **xaxis_label** – the label for the x axis (e.g. Time [min])
- **yaxis_label** – the label for the y axis (e.g. Level [a.u.])

Returns True if the task was completed successfully, False otherwise.

classmethod generate_data(*simulator*, *model*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=2*, *runs=1*)

The first pipeline step: data generation.

Parameters

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model to process
- **inputdir** – the directory containing the model
- **outputdir** – the directory containing the output files
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.

- **local_cpus** – the number of CPUs.
- **runs** – the number of model simulation

Returns True if the task was completed successfully, False otherwise.

classmethod generate_report(*model*, *outputdir*, *sim_plots_folder*)
The third pipeline step: report generation.

Parameters

- **model** – the model name
- **outputdir** – the output directory to store the report
- **sim_plots_folder** – the folder containing the plots

Returns True if the task was completed successfully, False otherwise.

parse(*my_dict*)

run(*config_file*)

Module contents

Submodules

sbpipe.pl.pipeline module

class sbpipe.pl.pipeline.**Pipeline**(*models_folder='Models'*, *working_folder='Results'*, *sim_data_folder='sim_data'*, *sim_plots_folder='sim_plots'*)
Bases: object

Generic pipeline.

Parameters

- **models_folder** – the folder containing the models
- **working_folder** – the folder to store the results
- **sim_data_folder** – the folder to store the simulation data
- **sim_plots_folder** – the folder to store the graphic results

get_models_folder()
Return the folder containing the models.

Returns the models folder.

get_sim_data_folder()
Return the folder containing the in-silico generated data sets.

Returns the folder of the simulated data sets.

get_sim_plots_folder()
Return the folder containing the in-silico generated plots.

Returns the folder of the simulated plots.

classmethod get_simul_obj(*simulator*)
Return the simulator object if this exists. Otherwise throws an exception. The simulator name starts with an upper case letter. Each simulator is in a package within *sbpipe.simulator*.

Parameters **simulator** – the simulator name

Returns the simulator object.

get_working_folder()
Return the folder containing the results.

Returns the working folder.

classmethod load(*config*)
Safely load a YAML configuration file and return its structure as a dictionary object.

Parameters **config** – a YAML configuration file

Returns the dictionary structure of the configuration file

Raise yaml.YAMLError if the config cannot be loaded.

parse(*config_dict*)
Read a dictionary structure containing the pipeline configuration. This method is abstract.

Returns a tuple containing the configuration

run(*config_file*)
Run the pipeline.

Parameters **config_file** – a configuration file for this pipeline.

Returns True if the pipeline was executed correctly, False otherwise.

Module contents

sbpipe.report package

Submodules

sbpipe.report.latex_reports module

sbpipe.report.latex_reports.**get_latex_header**(*pdftitle='SBpipe report'*, *title='SBpipe report'*, *abstract='Generic report.'*)
Initialize a Latex header with a title and an abstract.

Parameters

- **pdftitle** – the pdftitle for the LaTeX header
- **title** – the title for the LaTeX header
- **abstract** – the abstract for the LaTeX header

Returns the LaTeX header

sbpipe.report.latex_reports.**latex_report**(*outputdir*, *plots_folder*, *model_noext*, *filename_prefix*, *caption=False*)
Generate a generic report.

Parameters

- **outputdir** – the output directory
- **plots_folder** – the folder containing the simulated plots
- **model_noext** – the model name
- **filename_prefix** – the prefix for the LaTeX file
- **caption** – True if figure captions (=figure file name) should be added

sbpipe.report.latex_reports.**latex_report_pe**(*outputdir*, *plots_folder*, *model_noext*, *filename_prefix*)
Generate a report for a parameter estimation task.

Parameters

- **outputdir** – the output directory
- **plots_folder** – the folder containing the simulated plots
- **model_noext** – the model name
- **filename_prefix** – the prefix for the LaTeX file

sbpipe.report.latex_reports.**latex_report_ps1**(*outputdir*, *plots_folder*, *filename_prefix*, *model_noext*, *scanned_par*)

Generate a report for a single parameter scan task.

Parameters

- **outputdir** – the output directory
- **plots_folder** – the folder containing the simulated plots
- **filename_prefix** – the prefix for the LaTeX file
- **model_noext** – the model name
- **scanned_par** – the scanned parameter

sbpipe.report.latex_reports.**latex_report_ps2**(*outputdir*, *plots_folder*, *filename_prefix*, *model_noext*, *scanned_par1*, *scanned_par2*)

Generate a report for a double parameter scan task.

Parameters

- **outputdir** – the output directory
- **plots_folder** – the folder containing the simulated plots
- **filename_prefix** – the prefix for the LaTeX file
- **model_noext** – the model name
- **scanned_par1** – the 1st scanned parameter
- **scanned_par2** – the 2nd scanned parameter

sbpipe.report.latex_reports.**latex_report_sim**(*outputdir*, *plots_folder*, *model_noext*, *filename_prefix*)

Generate a report for a time course task.

Parameters

- **outputdir** – the output directory
- **plots_folder** – the folder containing the simulated plots
- **model_noext** – the model name
- **filename_prefix** – the prefix for the LaTeX file

sbpipe.report.latex_reports.**pdf_report**(*outputdir*, *filename*)

Generate a PDF report from LaTeX report using pdflatex.

Parameters

- **outputdir** – the output directory
- **filename** – the LaTeX file name

Module contents

sbpipe.simul package

Subpackages

sbpipe.simul.copasi package

Submodules

sbpipe.simul.copasi.copasi module

class sbpipe.simul.copasi.copasi.**Copasi**
Bases: *sbpipe.simul.simul.Simul* (page 31)

Copasi simulator.

pe(*model*, *inputdir*, *cluster*, *local_cpus*, *runs*, *outputdir*, *sim_data_dir*, *output_msg=False*)

ps1(*model*, *scanned_par*, *simulate_intervals*, *single_param_scan_intervals*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)

ps2(*model*, *sim_length*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)

sim(*model*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)

Module contents

sbpipe.simul.python package

Submodules

sbpipe.simul.python.python module

class sbpipe.simul.python.python.**Python**
Bases: *sbpipe.simul.pl_simul.PLSimul* (page 30)

Python Simulator.

Module contents

Submodules

sbpipe.simul.pl_simul module

class sbpipe.simul.pl_simul.**PLSimul**(*lang*, *lang_err_msg*, *options*)
Bases: *sbpipe.simul.simul.Simul* (page 31)

A generic simulator for models coded in a programming language.

get_lang()
Return the programming language name :return: the name

get_lang_err_msg()
Return the error if the programming language is not found :return: the error message

get_lang_options()
Return the options for the programming language command :return: the options. Return None, if no options are used.

pe(*model*, *inputdir*, *cluster*, *local_cpus*, *runs*, *outputdir*, *sim_data_dir*, *output_msg=False*)

ps1 (*model*, *scanned_par*, *simulate_intervals*, *single_param_scan_intervals*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)

ps2 (*model*, *sim_length*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)

replace_str_in_report (*report*)

sim (*model*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)

sbpipe.simul.simul module

class sbpipe.simul.simul.**Simul**
Bases: object

Generic simulator.

get_all_fits (*path_in='.'*, *path_out='.'*, *filename_out='all_estimates.csv'*)
Collect all the parameter estimates. Results are stored in filename_out.

Parameters

- **path_in** – the path to the input files
- **path_out** – the path to the output files
- **filename_out** – a global file containing all fits from independent parameter estimations.

Returns the number of retrieved files

get_best_fits (*path_in='.'*, *path_out='.'*, *filename_out='final_estimates.csv'*)
Collect the final parameter estimates. Results are stored in filename_out.

Parameters

- **path_in** – the path to the input files
- **path_out** – the path to the output files
- **filename_out** – a global file containing the best fits from independent parameter estimations.

Returns the number of retrieved files

pe (*model*, *inputdir*, *cluster*, *local_cpus*, *runs*, *outputdir*, *sim_data_dir*, *output_msg=False*)
parameter estimation.

Parameters

- **model** – the model to process
- **inputdir** – the directory containing the model
- **cluster** – local, lsf for load sharing facility, sge for sun grid engine
- **local_cpus** – the number of cpu
- **runs** – the number of fits to perform
- **outputdir** – the directory to store the results
- **sim_data_dir** – the directory containing the simulation data sets
- **output_msg** – print the output messages on screen (available for cluster='local' only)

ps1 (*model*, *scanned_par*, *simulate_intervals*, *single_param_scan_intervals*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)
Single parameter scan.

Parameters

- **model** – the model to process
- **scanned_par** – the scanned parameter
- **simulate_intervals** – the time step of each simulation
- **single_param_scan_intervals** – the number of scans to perform
- **inputdir** – the directory containing the model
- **outputdir** – the directory to store the results
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPU used.
- **runs** – the number of model simulation
- **output_msg** – print the output messages on screen (available for cluster='local' only)

ps1_postproc(*model*, *scanned_par*, *simulate_intervals*, *single_param_scan_intervals*, *outputdir*)
Perform post processing organisation to single parameter scan report files.

Parameters

- **model** – the model to process
- **scanned_par** – the scanned parameter
- **simulate_intervals** – the time step of each simulation
- **single_param_scan_intervals** – the number of scans to perform
- **outputdir** – the directory to store the results

ps2(*model*, *sim_length*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)
Double paramter scan.

Parameters

- **model** – the model to process
- **sim_length** – the length of the simulation
- **inputdir** – the directory containing the model
- **outputdir** – the directory to store the results
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPU.
- **runs** – the number of model simulation
- **output_msg** – print the output messages on screen (available for cluster='local' only)

ps2_postproc(*model*, *sim_length*, *outputdir*)
Perform post processing organisation to double parameter scan report files.

Parameters

- **model** – the model to process
- **sim_length** – the length of the simulation
- **outputdir** – the directory to store the results

replace_str_in_report(*report*)
Replaces strings in a report file.

Parameters **report** – a report file with its absolute path

sim(*model*, *inputdir*, *outputdir*, *cluster='local'*, *local_cpus=1*, *runs=1*, *output_msg=False*)
Time course simulator.

Parameters

- **model** – the model to process
- **inputdir** – the directory containing the model
- **outputdir** – the directory containing the output files
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPU.
- **runs** – the number of model simulation
- **output_msg** – print the output messages on screen (available for cluster='local' only)

Module contents

sbpipe.tasks package

Submodules

sbpipe.tasks.generate_data module

sbpipe.tasks.generate_data.**generate_data**(*infile*, *copasi=False*)
Replicate a copasi model and adds an id.

Parameters

- **infile** – the input file
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.generate_data.**main**(*argv=None*)

sbpipe.tasks.generate_data.**run_copasi_model**(*infile*)
Run a Copasi model

Parameters **infile** – the input file

sbpipe.tasks.generate_data.**run_generic_model**(*infile*)
Run a generic model

Parameters **infile** – the input file

sbpipe.tasks.pe_analyse_data module

sbpipe.tasks.pe_analyse_data.**main**(*argv=None*)

sbpipe.tasks.pe_analyse_data.**pe_analyse_data**(*model*, *outputdir*, *fileout_final_estims*, *fileout_all_estims*, *fileout_param_estim_details*, *fileout_param_estim_summary*, *plots_dir*, *best_fits_percent*, *data_point_num*, *plot_2d_66cl_corr=False*, *plot_2d_95cl_corr=False*, *plot_2d_99cl_corr=False*, *logspace=True*, *scientific_notation=True*)
Plot parameter estimation results (Python wrapper).

param model the model name

param outputdir the directory to store the results

param fileout_final_estims the name of the file containing final parameter sets with the objective value

param fileout_all_estims the name of the file containing all the parameter sets with the objective value

param fileout_param_estim_details the name of the file containing the detailed statistics for the estimated parameters

param fileout_param_estim_summary the name of the file containing the summary for the parameter estimation

param plots_dir the directory of the simulation plots

param best_fits_percent the percent to consider for the best fits

param data_point_num the number of data points

param plot_2d_66cl_corr True if 2 dim plots for the parameter sets within 66% should be plotted

param plot_2d_95cl_corr True if 2 dim plots for the parameter sets within 95% should be plotted

param plot_2d_99cl_corr True if 2 dim plots for the parameter sets within 99% should be plotted

param logspace True if parameters should be plotted in log space

param scientific_notation True if axis labels should be plotted in scientific notation

return True if the task was completed successfully, False otherwise.

sbpipe.tasks.pe_analyse_data_all_fits module

sbpipe.tasks.pe_analyse_data_all_fits.**main**(*argv=None*)

sbpipe.tasks.pe_analyse_data_all_fits.**pe_analyse_data_all_fits**(*model*, *outputdir*, *fileout_all_estims*, *fileout_param_estim_details*, *fileout_param_estim_summary*, *plots_dir*, *data_point_num*, *plot_2d_66cl_corr=False*, *plot_2d_95cl_corr=False*, *plot_2d_99cl_corr=False*, *logspace=True*, *scientific_notation=True*)

Plot parameter estimation results (Python wrapper).

param model the model name

param outputdir the directory to store the results

param fileout_all_estims the name of the file containing all the parameter sets with the objective value

param fileout_param_estim_details the name of the file containing the detailed statistics for the estimated parameters

param fileout_param_estim_summary the name of the file containing the summary for the parameter estimation

param plots_dir the directory of the simulation plots

param data_point_num the number of data points

param plot_2d_66cl_corr True if 2 dim plots for the parameter sets within 66% should be plotted

param plot_2d_95cl_corr True if 2 dim plots for the parameter sets within 95% should be plotted

param plot_2d_99cl_corr True if 2 dim plots for the parameter sets within 99% should be plotted

param logspace True if parameters should be plotted in log space

param scientific_notation True if axis labels should be plotted in scientific notation

return True if the task was completed successfully, False otherwise.

sbpipe.tasks.pe_analyse_data_best_fits module

sbpipe.tasks.pe_analyse_data_best_fits.**main**(*argv=None*)

sbpipe.tasks.pe_analyse_data_best_fits.**pe_analyse_data_best_fits**(*model*, *outputdir*, *fileout_final_estims*, *plots_dir*, *best_fits_percent*, *logspace=True*, *scientific_notation=True*)

Plot parameter estimation results (Python wrapper).

param model the model name

param outputdir the directory to store the results

param fileout_final_estims the name of the file containing final parameter sets with the objective value

param plots_dir the directory of the simulation plots

param best_fits_percent the percent to consider for the best fits

param logspace True if parameters should be plotted in log space

param scientific_notation True if axis labels should be plotted in scientific notation

return True if the task was completed successfully, False otherwise.

sbpipe.tasks.pe_collect module

sbpipe.tasks.pe_collect.**main**(*argv=None*)

sbpipe.tasks.pe_collect.**pe_collect**(*inputdir*, *outputdir*, *fileout_final_estims*, *fileout_all_estims*, *copasi=True*)

Collect the results so that they can be processed. :param inputdir: the input folder containing the data :param outputdir: the output folder to stored the collected results :param fileout_final_estims: the name of the file containing the best estimations :param fileout_all_estims: the name of the file containing all the estimations :param copasi: True if COPASI was used to generate the data.

sbpipe.tasks.pe_postproc module

sbpipe.tasks.pe_postproc.**generic_postproc**(*infile*, *outfile*, *copasi=True*)
Perform post processing file editing for the *pe* pipeline

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.pe_postproc.**main**(*argv=None*)

sbpipe.tasks.pe_postproc.**pe_postproc**(*infile*, *outfile*, *copasi=True*)
Perform post processing file editing for the *pe* pipeline

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.preproc module

sbpipe.tasks.preproc.**copasi_preproc**(*infile*, *outfile*)
Replicate a copasi model and adds an id.

Parameters

- **infile** – the input file
- **outfile** – the output file

sbpipe.tasks.preproc.**generic_preproc**(*infile*, *outfile*)
Copy the model file

Parameters

- **infile** – the input file
- **outfile** – the output file

sbpipe.tasks.preproc.**main**(*argv=None*)

sbpipe.tasks.preproc.**preproc**(*infile*, *outfile*, *copasi=False*)
Replicate a copasi model and adds an id.

Parameters

- **infile** – the input file
- **outfile** – the output file
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.ps1_analyse_data module

sbpipe.tasks.ps1_analyse_data.**main**(*argv=None*)

sbpipe.tasks.ps1_analyse_data.**ps1_analyse_data**(*model_name*, *inhibition_only*, *outputdir*, *sim_data_folder*, *sim_plots_folder*, *repeat*, *percent_levels*, *min_level*, *max_level*, *levels_number*, *homogeneous_lines*, *xaxis_label*, *yaxis_label*)

Plot model single parameter scan time courses (Python wrapper).

Parameters

- **model_name** – the model name without extension
- **inhibition_only** – true if the scanning only decreases the variable amount (inhibition only)
- **outputdir** – the output directory
- **sim_data_folder** – the name of the folder containing the simulated data
- **sim_plots_folder** – the name of the folder containing the simulated plots
- **repeat** – the simulation number
- **percent_levels** – true if scanning levels are in percent
- **min_level** – the minimum level
- **max_level** – the maximum level
- **levels_number** – the number of levels
- **homogeneous_lines** – true if lines should be plotted homogeneously
- **xaxis_label** – the label for the x axis (e.g. Time [min])
- **yaxis_label** – the label for the y axis (e.g. Level [a.u.])

sbpipe.tasks.ps1_postproc module

sbpipe.tasks.ps1_postproc.**generic_postproc**(*infile*, *outfile*, *scanned_par*, *simulate_intervals*, *single_param_scan_intervals*, *copasi=True*)

Perform post processing organisation to single parameter scan report files.

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **scanned_par** – the scanned parameter
- **simulate_intervals** – the time step of each simulation
- **single_param_scan_intervals** – the number of scans to perform
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.ps1_postproc.**main**(*argv=None*)

sbpipe.tasks.ps1_postproc.**ps1_header_init**(*report*, *scanned_par*)

Header report initialisation for single parameter scan pipeline.

Parameters

- **report** – a report
- **scanned_par** – the scanned parameter

:return a list containing the header or an empty list if no header was created.

sbpipe.tasks.ps1_postproc.**ps1_postproc**(*infile*, *outfile*, *scanned_par*, *simulate_intervals*, *single_param_scan_intervals*, *copasi=True*)

Perform post processing organisation to single parameter scan report files.

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **scanned_par** – the scanned parameter
- **simulate_intervals** – the time step of each simulation
- **single_param_scan_intervals** – the number of scans to perform
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.ps2_analyse_data module

sbpipe.tasks.ps2_analyse_data.**main**(*argv=None*)

sbpipe.tasks.ps2_analyse_data.**ps2_analyse_data**(*model*, *scanned_par1*, *scanned_par2*, *inputdir*, *outputdir*, *id*)

Plot model double parameter scan time courses (Python wrapper).

Parameters

- **model** – the model name without extension
- **scanned_par1** – the 1st scanned parameter
- **scanned_par2** – the 2nd scanned parameter
- **inputdir** – the input directory
- **outputdir** – the output directory
- **run** – the simulation number

sbpipe.tasks.ps2_postproc module

sbpipe.tasks.ps2_postproc.**generic_postproc**(*infile*, *outfile*, *sim_length*, *copasi=True*)

Perform post processing organisation to double parameter scan report files.

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **sim_length** – the length of the simulation
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.ps2_postproc.**main**(*argv=None*)

sbpipe.tasks.ps2_postproc.**ps2_postproc**(*infile*, *outfile*, *sim_length*, *copasi=True*)

Perform post processing organisation to double parameter scan report files.

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **sim_length** – the length of the simulation
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.sim_analyse_data module

sbpipe.tasks.sim_analyse_data.**main**(*argv=None*)

sbpipe.tasks.sim_analyse_data.**sim_analyse_data**(*model*, *inputdir*, *outputdir*, *sim_plots_dir*, *exp_dataset*, *plot_exp_dataset*, *xaxis_label=''*, *yaxis_label=''*)

Plot model simulation time courses (Python wrapper).

Parameters

- **model** – the model name
- **inputdir** – the directory containing the data to analyse
- **outputdir** – the output directory containing the results
- **sim_plots_dir** – the directory to save the plots
- **exp_dataset** – the full path of the experimental data set
- **plot_exp_dataset** – True if the experimental data set should also be plotted
- **xaxis_label** – the label for the x axis (e.g. Time [min])
- **yaxis_label** – the label for the y axis (e.g. Level [a.u.])

sbpipe.tasks.sim_postproc module

sbpipe.tasks.sim_postproc.**generic_postproc**(*infile*, *outfile*, *copasi=True*)

Perform post processing file editing for the *simulate* pipeline

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.sim_postproc.**main**(*argv=None*)

sbpipe.tasks.sim_postproc.**sim_postproc**(*infile*, *outfile*, *copasi=True*)

Perform post processing file editing for the *simulate* pipeline

Parameters

- **infile** – the model to process
- **outfile** – the directory to store the results
- **copasi** – True if the model is a Copasi model

sbpipe.tasks.utils module

Module contents

sbpipe.utils package

Submodules

sbpipe.utils.io module

sbpipe.utils.io.**files_with_pattern_recur**(*folder*, *pattern*)
Return all files with a certain pattern in folder+subdirectories

Parameters

- **folder** – the folder to search for
- **pattern** – the string to search for

Returns the files containing the pattern.

sbpipe.utils.io.**get_pattern_pos**(*pattern*, *filename*)
Return the line number (as string) of the first occurrence of a pattern in filename

Parameters

- **pattern** – the pattern of the string to find
- **filename** – the file name containing the pattern to search

Returns the line number containing the pattern or "-1" if the pattern was not found

sbpipe.utils.io.**refresh**(*path*, *file_pattern*)
Clean and create the folder if this does not exist.

Parameters

- **path** – the path containing the files to remove
- **file_pattern** – the string pattern of the files to remove

sbpipe.utils.io.**remove_file_silently**(*filename*)
Remove a filename silently, without reporting warnings or error messages. This is not really needed by Linux, but Windows sometimes fails to remove the file even if this exists.

Parameters **filename** – the file to remove

sbpipe.utils.io.**replace_str_in_file**(*filename_out*, *old_string*, *new_string*)
Replace a string with another in filename_out

Parameters

- **filename_out** – the output file
- **old_string** – the old string that should be replaced
- **new_string** – the new string replacing old_string

sbpipe.utils.io.**replace_str_in_report**(*report*)
Replace nasty strings in COPASI report file.

Parameters **report** – the report

sbpipe.utils.io.**write_mat_on_file**(*path*, *filename_out*, *data*)
Write the matrix results stored in data to filename_out

Parameters

- **path** – the path to filename_out
- **filename_out** – the output file
- **data** – the data to store in a file

sbpipe.utils.parcomp module

sbpipe.utils.parcomp.**call_proc**(*params*)
Run a command using Python subprocess.

Parameters **params** – A tuple containing (the string of the command to run, the command id)

sbpipe.utils.parcomp.**is_output_file_clean**(*filename*, *stream_type='standard output'*)
Check whether a file contains the string 'error' or 'warning'. If so a message is printed.

Parameters

- **filename** – a file
- **stream_type** – 'stderr' for standard error, 'stdout' for standard output.

Returns True

sbpipe.utils.parcomp.**parcomp**(*cmd*, *cmd_iter_substr*, *output_dir*, *cluster='local'*, *runs=1*, *local_cpus=1*, *output_msg=False*)
Generic function to run a command in parallel

Parameters

- **cmd** – the command string to run in parallel
- **cmd_iter_substr** – the substring of the iteration number. This will be replaced in a number automatically
- **output_dir** – the output directory
- **cluster** – the cluster type among local (Python multiprocessing), sge, or lsf
- **runs** – the number of runs
- **local_cpus** – the number of cpus to use at most
- **output_msg** – print the output messages on screen (available for cluster='local' only)

Returns True if the computation succeeded.

sbpipe.utils.parcomp.**quick_debug**(*cmd*, *out_dir*, *err_dir*)
Look up for *error* and *warning* in the standard output and error files. A simple debugging function checking the generated log files. We don't stop the computation because it happens that these messages are more *warnings* than real errors.

Parameters

- **cmd** – the executed command
- **out_dir** – the directory containing the standard output files
- **err_dir** – the directory contining the standard error files

Returns True

sbpipe.utils.parcomp.**run_cmd**(*cmd*)
Run a command using Python subprocess.

Parameters **cmd** – The string of the command to run

sbpipe.utils.parcomp.**run_cmd_block**(*cmd*)
Run a command using Python subprocess. Block the call until the command has finished.

Parameters **cmd** – A tuple containing the string of the command to run

sbpipe.utils.parcomp.**run_jobs_local**(*cmd*, *cmd_iter_substr*, *runs=1*, *local_cpus=1*, *output_msg=False*)
Run jobs using python multiprocessing locally.

Parameters

- **cmd** – the full command to run as a job
- **cmd_iter_substr** – the substring in command to be replaced with a number
- **runs** – the number of runs to execute

- **local_cpus** – The number of available cpus. If local_cpus <=0, only one core will be used.
- **output_msg** – print the output messages on screen (available for cluster_type='local' only)

Returns True

sbpipe.utils.parcomp.**run_jobs_lsf**(*cmd*, *cmd_iter_substr*, *out_dir*, *err_dir*, *runs=1*)
Run jobs using a Load Sharing Facility (LSF) cluster.

Parameters

- **cmd** – the full command to run as a job
- **cmd_iter_substr** – the substring in command to be replaced with a number
- **out_dir** – the directory containing the standard output from bsub
- **err_dir** – the directory containing the standard error from bsub
- **runs** – the number of runs to execute

Returns True if the computation succeeded.

sbpipe.utils.parcomp.**run_jobs_sge**(*cmd*, *cmd_iter_substr*, *out_dir*, *err_dir*, *runs=1*)
Run jobs using a Sun Grid Engine (SGE) cluster.

Parameters

- **cmd** – the full command to run as a job
- **cmd_iter_substr** – the substring in command to be replaced with a number
- **out_dir** – the directory containing the standard output from qsub
- **err_dir** – the directory containing the standard error from qsub
- **runs** – the number of runs to execute

Returns True if the computation succeeded.

sbpipe.utils.rand module

sbpipe.utils.rand.**get_rand_alphanum_str**(*length*)
Return a random alphanumeric string

Parameters **length** – the length of the string

Returns the generated string

sbpipe.utils.rand.**get_rand_num_str**(*length*)
Return a random numeric string

Parameters **length** – the length of the string

Returns the generated string

sbpipe.utils.re_utils module

sbpipe.utils.re_utils.**escape_special_chars**(*text*)
Escape ^,%, ,[,],(,),{,} from text :param text: the command to escape special characters inside :return: the command with escaped special characters

sbpipe.utils.re_utils.**nat_sort_key**(*str*)
The key to sort a list of strings alphanumerically (e.g. "file10" is correctly placed after "file2")

Parameters **str** – the string to sort alphanumerically in a list of strings

Returns the key to sort strings alphanumerically

Module contents

Submodules

sbpipe.__main__ module

sbpipe.__main__.**main**(*argv=None*)

sbpipe.main module

sbpipe.main.**main**(*argv=None*)
SBpipe main function.

Returns 0 if OK, 1 if trouble

sbpipe.main.**read_file_header**(*filename*)
Read the first line of a file

Parameters **filename** – the file name to read

Returns the first line

sbpipe.main.**sbpipe**(*create_project=''*, *simulate=''*, *parameter_scan1=''*, *parameter_scan2=''*, *parameter_estimation=''*, *logo=False*, *license=False*, *nocolor=False*, *log_level=''*, *quiet=False*, *verbose=False*)
SBpipe function.

Parameters

- **create_project** – create a project with the name as argument
- **simulate** – model simulation using a configuration file as argument
- **parameter_scan1** – model one parameter scan using a configuration file as argument
- **parameter_scan2** – model two parameters scan using a configuration file as argument
- **parameter_estimation** – model parameter estimation using a configuration file as argument
- **logo** – True to print the logo
- **license** – True to print the license
- **nocolor** – True to print logging messages without colors
- **log_level** – Set the logging level
- **quiet** – True if quiet (CRITICAL+)
- **verbose** – True if verbose (DEBUG+)

Returns 0 if OK, 1 if trouble (e.g. a pipeline did not execute correctly).

sbpipe.main.**sbpipe_logo**()
Return sbpipe logo.

Returns sbpipe logo

sbpipe.main.**set_basic_logger**(*level='INFO'*)
Set a basic StreamHandler logger. :param level: the level for this console logger

sbpipe.main.**set_color_logger**(*level='INFO'*)
Replace the current logging.StreamHandler with colorlog.StreamHandler. :param level: the level for this console logger

sbpipe.main.**set_console_logger**(*new_level='NOTSET'*, *current_level='INFO'*, *nocolor=False*)
Set the console logger to a new level if this is different from NOTSET

Parameters

- **new_level** – the new level to set for the console logger
- **current_level** – the current level to set for the console logger
- **nocolor** – True if no colors shouls be used

sbpipe.main.**set_logger**(*level='NOTSET'*, *nocolor=False*)
Set the logger :param level: the level for the console logger :param nocolor: True if no colors shouls be used

sbpipe.sbpipe_config module

sbpipe.sbpipe_config.**isPyPackageInstalled**(*package*)
Utility checking whether a Python package is installed.

Parameters **package** – a Python package name

Returns True if it is installed, false otherwise.

sbpipe.sbpipe_config.**which**(*cmd_name*)
Utility equivalent to *which* in GNU/Linux OS.

Parameters **cmd_name** – a command name

Returns return the command name with absolute path if this exists, or None

Module contents

CHAPTER

FOUR

INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

W