sb_pipe documentation

Release 0.59.0

Piero Dalle Pezze

September 16, 2016

CHAPTER

ONE

USER MANUAL

Mailing list: sb_pipe AT googlegroups.com

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

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 and double parameter scan, and parameter estimation.

Requirements

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

- Copasi 4.16 - http://copasi.org/
- Python 2.7+ - https://www.python.org/
- R 3.3.0+ - https://cran.r-project.org/
- LaTeX 2013 (optional) https://latex-project.org/ftp.html

You should also make sure that the following packages are installed in your machine: `python-pip`, and (optionally) `texlive-latex-base`.

Before installing sb_pipe Python and R dependencies the following environment variables must be added to your GNU/Linux $HOME/.bashrc file:

```
# SB_PIPE
export SB_PIPE=/path/to/sb_pipe
export PATH=$PATH:${SB_PIPE}/sb_pipe

# Path to CopasiSE
export PATH=$PATH:/path/to/CopasiSE
```

The .bashrc file can then be reloaded from your shell using the command:

```
$ source $HOME/.bashrc
```

On Windows platforms, these environment variables are configured as any other Windows environment variable.

Now it is the time to install Python and R packages used by sb_pipe. Two scripts are provided to perform these tasks automatically.

To install sb_pipe Python dependencies, run:

```
cd ${SB_PIPE}/
./install_pydeps.py
```

To install sb_pipe R dependencies, run:

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

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`. After installing these packages, `install_rdeps.r` must be executed again.

Installation

Run the command inside the sb_pipe folder:

```
python setup.py install
```

The correct installation of sb_pipe and its dependencies can be checked by running the following commands inside the sb_pipe folder:

```
cd tests
./test_suite.py
```

How to use sb_pipe

Preliminary configuration steps

Pipelines using Copasi

Before using these pipelines, a Copasi model must be configured as follow using CopasiUI:

pipeline: simulate

- 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.

pipeline: 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.

pipeline: param-estim

- 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.

Running sb_pipe

sb_pipe is executed via the command *run_sb_pipe.py*. The syntax for this command and its complete list of options can be retrieved by running *run_sb_pipe.py -h*.

As of Sep 2016 the output is as follows:

```
pdp@ariel:~/sb_pipe$ run_sb_pipe.py -h
Usage: run_sb_pipe.py [OPTION] [FILE]
Pipelines for systems modelling of biological networks.

List of mandatory options:
        -h, --help
                Shows this help.
        -c, --create-project
                Create a project structure using the argument as name.
        -s, --simulate
                Simulate a model.
        -p, --single-param-scan
                Simulate a single parameter scan.
        -d, --double-param-scan
                Simulate a double parameter scan.
        -e, --param-estim
                Generate a parameter fit sequence.
Exit status:
 0  if OK,
 1  if minor problems (e.g., a pipeline did not execute correctly),
 2  if serious trouble (e.g., cannot access command-line argument).

Report bugs to sb_pipe@googlegroups.com
sb_pipe home page: <https://pdp10.github.io/sb_pipe>
For complete documentation, see README.md .
```

The first step is to create a new project. This can be done with the command:

```
run_sb_pipe.py --create-project project_name
```

This generates the following structure:

```
project_name/
    | - Data/
    | - Models/
    | - Working_Folder/
```

Models must be stored in the Models/ folder. The folder Data/ is meant for collecting experimental data files and analyses in one place. Once the data files for Copasi (e.g. for parameter estimation) are generated, **it is advised** to move them into the Models/ folder so that the Copasi (.cps) file and its associated experimental data files are stored in the same folder. To run sb_pipe, 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 Working_Folder/. This folder will eventually contain all the results generated by sb_pipe.

For instance, the pipeline for parameter estimation configured with a certain configuration file can be executed by typing:

```
run_sb_pipe.py -e my_config_file.conf
```

Pipeline configuration files

Pipelines are configured using files (here called configuration files). These files are INI files and are therefore structured as follows:

```
[pipeline_name]
option1=value1
option2=value2
...
```

In sb_pipe each pipeline executes three tasks: data generation, data analysis, and report generation. Each task depends on the previous one. This choice allows user to analyse the same data without having to generate it every time, or to skip the report generation if not wanted. Assuming that the configuration files are placed in the Working_Folder of a certain project, examples are given as follow:

Example 1: configuration file for the pipeline *simulate*

```
[simulate]
# True if data must be generated, False otherwise
generate_data=True
# True if data must be analysed, False otherwise
analyse_data=True
# True if a report must be generated, False otherwise
generate_report=True
# The relative path to the project directory (from Working_Folder)
project_dir=..
# The Copasi model name
model=insulin_receptor_stoch.cps
# The cluster type. pp if the model is run locally, sge/lsf if run on cluster.
cluster=pp
# The number of CPU if pp is used, ignored otherwise
pp_cpus=7
# The number of simulations to perform. n>=1 for stochastic simulations.
runs=40
# The label for the x axis.
simulate__xaxis_label=Time [min]
```

Example 2: configuration file for the pipeline *single_param_scan*

```
[single_param_scan]
generate_data=True
analyse_data=True
generate_report=True
project_dir=..
model=insulin_receptor_inhib_scan_IR_beta.cps
# The variable to scan (as set in Copasi Parameter Scan Task)
scanned_par=IR_beta
# The number of intervals in the simulation
simulate__intervals=100
simulate__xaxis_label=Time [min]
# The number of simulations to perform for each scan
single_param_scan_simulations_number=1
# True if the variable is only reduced (knock down), False otherwise.
single_param_scan_knock_down_only=True
# True if the scanning represents percent levels.
single_param_scan_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
```

Example 3: configuration file for the pipeline *double_param_scan*

```
[double_param_scan]
generate_data=True
```

```
analyse_data=True
generate_report=True
project_dir=..
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 length of the simulation (as set in Copasi Time Course Task)
sim_length=10
```

Example 4: configuration file for the pipeline *param_estim*

```
[param_estim]
generate_data=True
analyse_data=True
generate_report=True
generate_tarball=True
project_dir=..
model=insulin_receptor_param_estim.cps
cluster=pp
pp_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=40
# 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% and 95% confidence levels
# should be plotted. This is computationally expensive.
plot_2d_66_95cl_corr=True
# True if parameter values should plotted in log space.
logspace=True
```

Additional examples of configuration files can be found in:

```
${SB_PIPE}/tests/insulin_receptor/Working_Folder/
```

How to report issues or request new features

sb_pipe is a relatively young project and there is a chance that some error occurs. If this is the case, users should report problems using the following mailing list:

```
sb_pipe AT googlegroups.com
```

To help us better identify and reproduce your problem, some technical information is needed. This detail data can be found in sb_pipe log files which are stored in ${HOME}/.sb_pipe/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 sb_pipe at the web page: https://github.com/pdp10/sb_pipe/issues.

CHAPTER

TWO

DEVELOPER MANUAL

Mailing list: sb_pipe AT googlegroups.com

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

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. The `develop` branch is versionless (just call it *-dev*).

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
```

Package structure

This section presents the structure of the sb_pipe package. The root of the project contains general management scripts for cleaning the package (clean_pacakge.py), installing Python and R dependencies (install_pydeps.py and install_rdeps.r), and installing sb_pipe (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:

```
sb_pipe:
  | - docs
  | - sb_pipe
        | - pipelines
        | - utils
  | - tests
```

These folders will be discussed in the next sections. In sb_pipe, Python is the project main language. Instead, R is essentially used for computing statistics within the *data analysis tasks* (see section configuration file in User manual) and for generating plots. This choice allows users to run these scripts independently from sb_pipe 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.

Documentation

The folder *docs/* contains the documentation for this project. In order to generate the complete documentation for sb_pipe, 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 sb_pipe documentation are provided below.

To generate the source code documentation:

```
$ ./gen_doc.sh
```

To clean the documentation:

```
$ ./clean_doc.sh
```

If new folders containing new Python modules are added to the project, it is necessary to update the sys.path in *source/conf.py* to include these additional paths.

sb_pipe

This folder contains the main script for running sb_pipe (run_sb_pipe.py). This script is an interface for the project.

Pipelines

The folder */sb_pipe/pipelines/* contains the following pipelines within folders:

- *create_project*: creates a new project
- *simulate*: simulates a model deterministically or stochastically using Copasi (this must be configured first), generate plots and report;
- *single_param_scan*: runs Copasi (this must be configured first), generate plots and report;
- *double_param_scan*: runs Copasi (this must be configured first), generate plots and report;
- *param_estim*: generate a fits sequence using Copasi (this must be configured first), generate tables for statistics.

These pipelines are invoked directly via the script *sb_pipe/run_sb_pipe.py*. Each pipeline extends the class *Pipeline*, which represents a generic and abstract pipeline. Each pipeline must implement the following methods of *Pipeline*:

```
def run(self, config_file)
def read_configuration(self, lines)
```

The method *run()* contains the procedure to execute for a specific configuration file. The method *read_configuration()* is needed for reading the options required by the pipeline to execute. The class *Pipeline* contains already implements the INI parser and returns each pipeline the configuration file as a list of lines.

Utils

The folder *sb_pipe/utils/* contains the following structure:

- *python*: a collection of python utils.
- *R*: a collection of R utils (plots and statistics).

Tests

The folder *tests/* contains the script *run_tests.py* to run a test suite. It should be used for testing the correct installation of sb_pipe dependencies as well as reference for configuring a project before running any pipeline. Projects inside the folder tests/ have the sb_pipe project structure:

- *Data*: (e.g. training / testing data sets for the model);
- *Model*: (e.g. Copasi models, datasets directly used by Copasi models);
- *Working_Folder*: (e.g. pipelines configurations and parameter estimation results, time course, parameter scan, etc).

Examples of configuration files (*.conf) can be found in ${SB_PIPE}/tests/insulin_receptor/Working_Folder/.

Travis-CI runs sb_pipe tests using `nosetests`. Please see .travis.yml for detail.

Miscellaneous of useful commands

Git

Startup

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

Update

```
$ git pull [--rebase] origin BRANCH  # ONLY use --rebase for private
branches. Never use it for shared branches otherwise it breaks the
history. --rebase moves your commits ahead. I think for shared
branches, you should use `git fetch && git merge --no-ff`.
**[FOR NOW, DON'T USE REBASE BEFORE AGREED]**.
```

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

- genindex
- modindex
- search