SBpipe documentation

Release 4.10.0

Piero Dalle Pezze and Nicolas Le Novère

Apr 04, 2018

Contents

Contacts: Dr Piero Dalle Pezze (piero.dallepezze AT gmail.com) 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. https://doi.org/10.1186/s12918-017-0423-3

1 Introduction

The rapid growth of the number of mathematical models in Systems Biology fostered the development of many tools to simulate and analyse them. The reliability and precision of these tasks often depend on multiple repetitions and they can be optimised if executed as pipelines. In addition, new formal analyses can be performed on these repeat sequences, revealing important insights about the accuracy of model predictions. SBpipe allows users to automatically repeat the tasks of model simulation and parameter estimation, and extract robustness information from these repeat sequences in a solid and consistent manner, facilitating model development and analysis.

2 Installation

2.1 Requirements

In order to use SBpipe, the following packages 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 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 package must also be installed:

- LaTeX 2013+

2.2 Installation on GNU/Linux

2.2.1 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)
```

2.2.2 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).
```

2.2.3 Install SBpipe via Miniconda3

Users need to download and install Miniconda3 (https://conda.io/miniconda.html).

1st Method

This method creates a new environment and installs SBpipe dependencies in this environment. SBpipe is installed locally, enabling an easy access to the package documentation and test suite.

```
# download SBpipe
wget https://github.com/pdp10/sbpipe/tarball/master
# or clone it from GitHub
git clone https://github.com/pdp10/sbpipe.git

# move to sbpipe folder
cd path/to/sbpipe

# install the dependencies within an isolated Miniconda3 environment
conda env create --name sbpipe --file environment.yaml

# activate the environment.
# For recent versions of conda, replace `source` with `conda`.
source activate sbpipe
```

To run sbpipe from any shell, users need to add 'sbpipe/scripts' to their `PATH` environment variable by adding the following lines to their `$HOME`/.bashrc file:

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

The .bashrc file should be reloaded to apply the previous edits:

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

2nd Method

This method installs SBpipe as a conda package in a dedicated conda environment:

```
# create a new environment `sbpipe`
conda create -n sbpipe

# activate the environment.
# For recent versions of conda, replace `source` with `conda`.
source activate sbpipe

# install sbpipe and its dependencies (including sbpiper)
conda install sbpipe -c pdp10 -c conda-forge -c fbergmann -c defaults
```

2.2.4 Install SBpipe manually

For this type of installation, SBpipe must be downloaded from the website or cloned using `git`.

```
# download SBpipe
wget https://github.com/pdp10/sbpipe/tarball/master
# or clone it from GitHub
git clone https://github.com/pdp10/sbpipe.git
```

Users need to make sure that the package `python-pip` and `r-base` are installed. The correct installation of Python and R can be tested by running the commands:

```
python -V
> Python 3.6.4
pip -V
> pip 9.0.1 from /home/ariel/.local/lib/python3.6/site-packages (python 3.6)

R --version
> R version 3.4.1 (2017-06-30) -- "Single Candle"
> Copyright (C) 2017 The R Foundation for Statistical Computing
> Platform: x86_64-pc-linux-gnu (64-bit)
```

The next step is the installation of SBpipe dependencies. To install 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')
```

Finally, to run sbpipe from any shell, users need to 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
```

The .bashrc file should be reloaded to apply the previous edits:

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

NOTES:

1. 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`. These can be installed using the package manager coming with your distribution. Other packages might be needed, depending on R dependencies. After installing these packages, `install_rdeps.r` must be executed again.
2. If Python bindings for COPASI are installed, SBpipe automatically checks whether the COPASI model can be loaded and executed, before generating the data. As of January 2018, this code is released for Python 2.7 and Python 3.6 on the COPASI website and Anaconda Cloud. The installation of SBpipe via Miniconda3 automatically installs this dependency.

2.3 Installation on Windows

2.3.1 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.

2.3.2 Installation of COPASI

Windows users need to install the Windows versions of COPASI from the COPASI website. Once `Git for Windows` is started, a Shell-like window appears and enables users to run commands. A .bashrc file must be created and configured:

```
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/

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

2.3.3 Installation of LaTeX

Windows users need to install LaTeX MikTeX https://miktex.org/.

2.3.4 Install SBpipe via Miniconda3

See GNU/Linux.

2.3.5 Install SBpipe manually

Start `Git for Windows` and clone SBpipe from GitHub using the command:

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

We now need to set up the path to SBpipe:

```
wordpad .bashrc
```

The following lines must be appended to this file:

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

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
```

Python and R dependencies should be installed as explained in the corresponding section for GNU/Linux section.

2.4 Test SBpipe

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

```
# SBpipe version:
sbpipe -V
> sbpipe 4.6.0
```

```
# 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
```

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

3 How to use SBpipe

SBpipe pipelines can be executed natively or via Snakemake, a dedicated and more advanced tool for running computational pipelines.

3.1 Run SBpipe natively

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)
```

Mathematical odels 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
```

3.1.1 Pipeline configuration files

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

- generate_data: True
- analyse_data: True
- generate_report: True
- generate_tarball: False

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. The `generate_report` task generates a LaTeX report containing the computed plots and invokes the utility `pdflatex` to produce a PDF file. Finally, `generate_tarball` creates a tar.gz file of the results. By default, this is not executed. 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
# True if a zipped tarball should be generated, False otherwise
generate_tarball: False
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi, Python)
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 alpha level used for plotting the experimental dataset
exp_dataset_alpha: 1.0
# 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
# True if a zipped tarball should be generated, False otherwise
generate_tarball: False
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi, Python)
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
```

(continues on next page)

(continued from previous page)

```
# 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
# True if a zipped tarball should be generated, False otherwise
generate_tarball: False
# The relative path to the project directory
project_dir: "."
# The name of the configurator (e.g. Copasi, Python)
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: False
# The relative path to the project directory
project_dir: "."
```

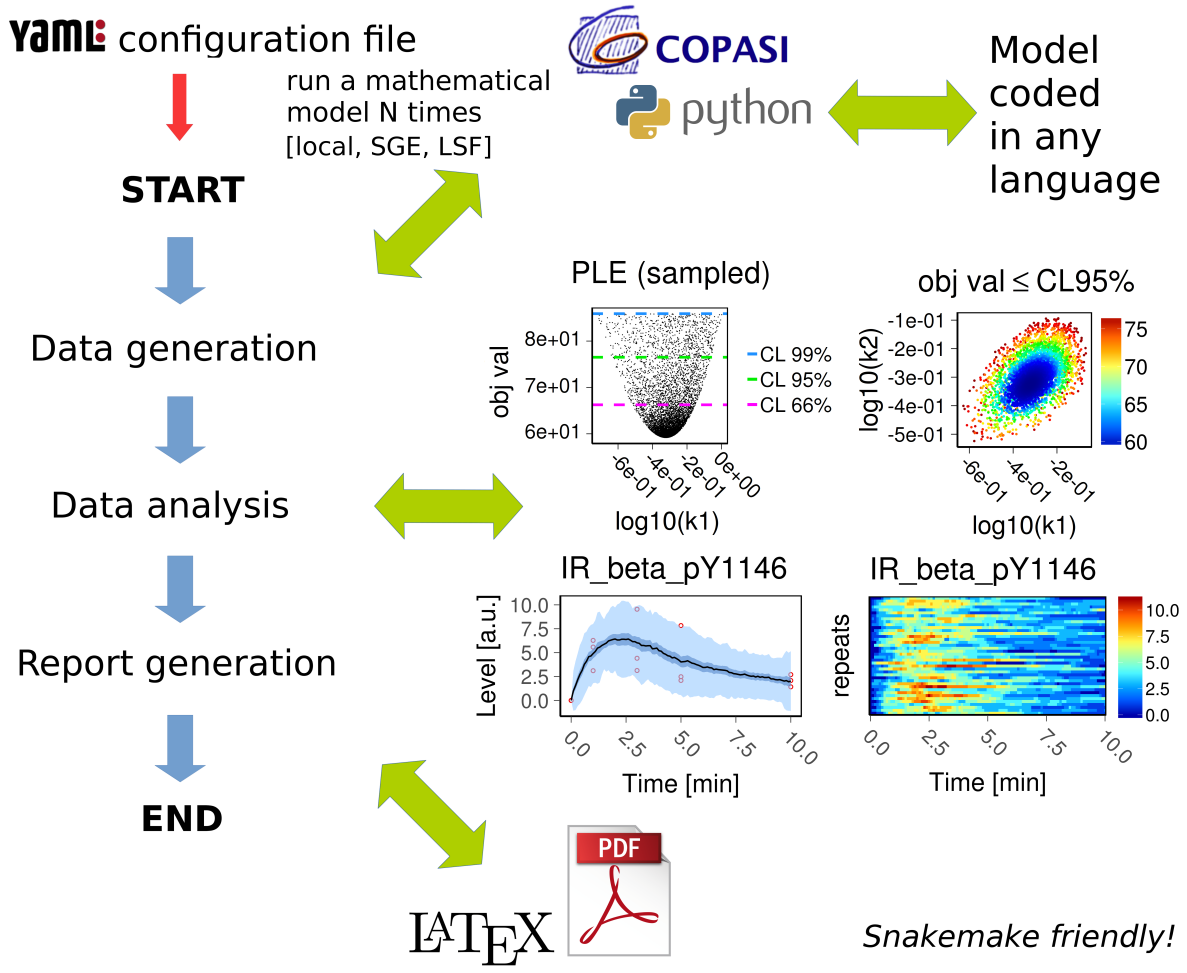
(continues on next page)

(continued from previous page)

```
# The name of the configurator (e.g. Copasi, Python)
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/
```



3.2 Run SBpipe via Snakemake

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

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

```
# Install snakemake (note: it requires python 3+ to run)
conda install -c bioconda snakemake
```

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 on an SGE cluster.
# snakemake waits for output files for 100 s.
snakemake -s path/to/sbpipe/sbpipe_pe.snake --configfile SBPIPE_CONFIG_FILE.yaml --
↪latency-wait 100 -j 40 --cluster "qsub -cwd -V -S /bin/sh"

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

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

If the grid engine supports DRMAA, it can be convenient to use Snakemake with the option `--drmaa`.

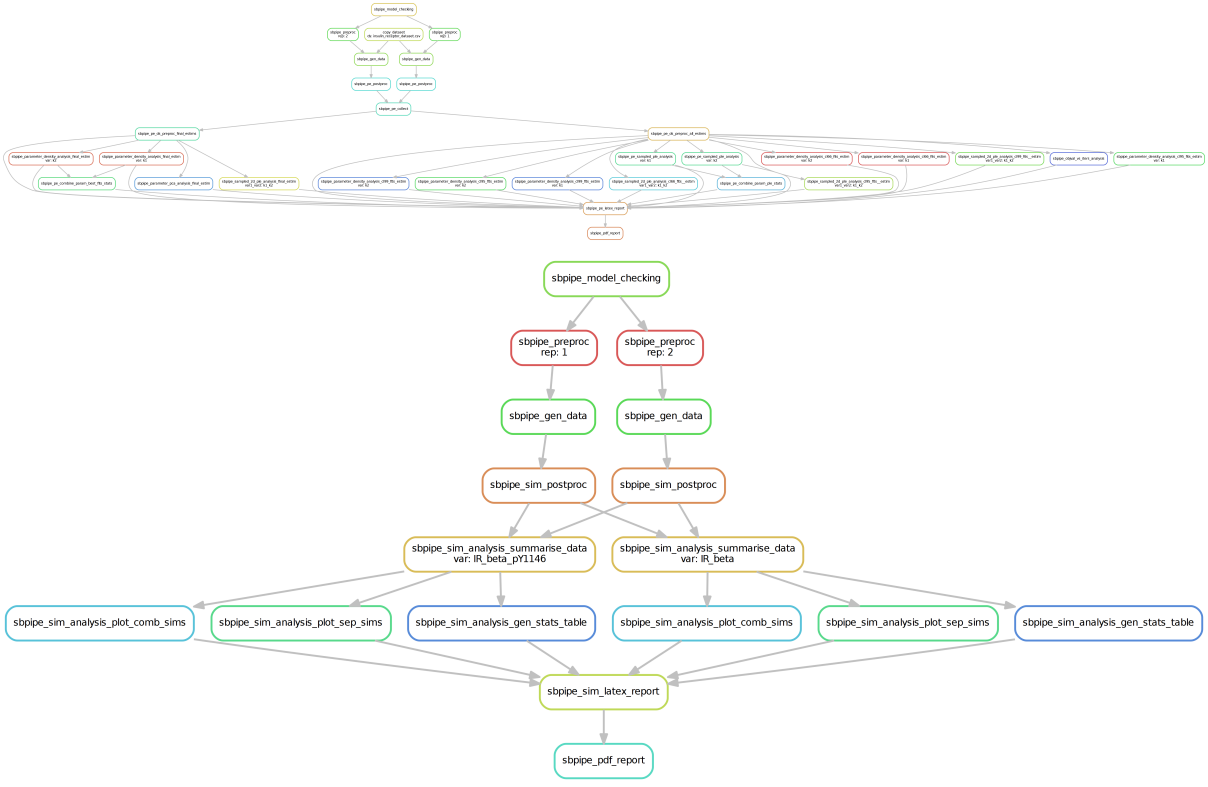
```
# See the DRMAA Python bindings for a preliminary documentation: https://pypi.
↪python.org/pypi/drmaa
# The following is an example of configuration for DRMAA for the grid engine␣
↪installed at the Babraham Institute
# (Cambridge, UK).

# load Python 3
module load python3/3.5.1
alias python=python3
# install python drmaa locally
easy_install-3.5 --user drmaa

# Update accordingly and add the following line to your ~/.bashrc file:
export SGE_ROOT=/opt/gridengine
export SGE_CELL=default
export DRMAA_LIBRARY_PATH=/opt/gridengine/lib/lx26-amd64/libdrmaa.so.1.0
```

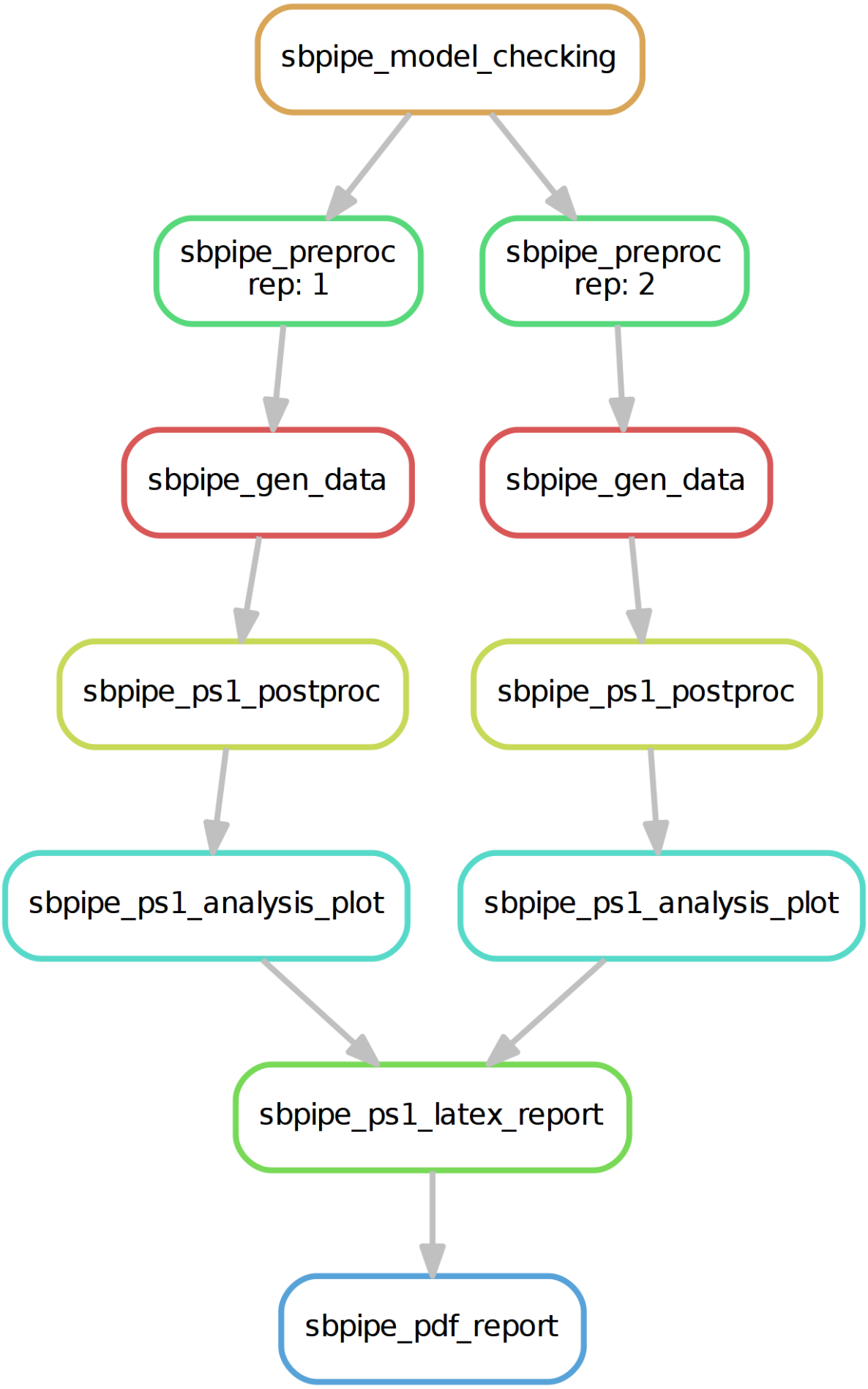
Snakemake can now be executed using drmaa as follows:

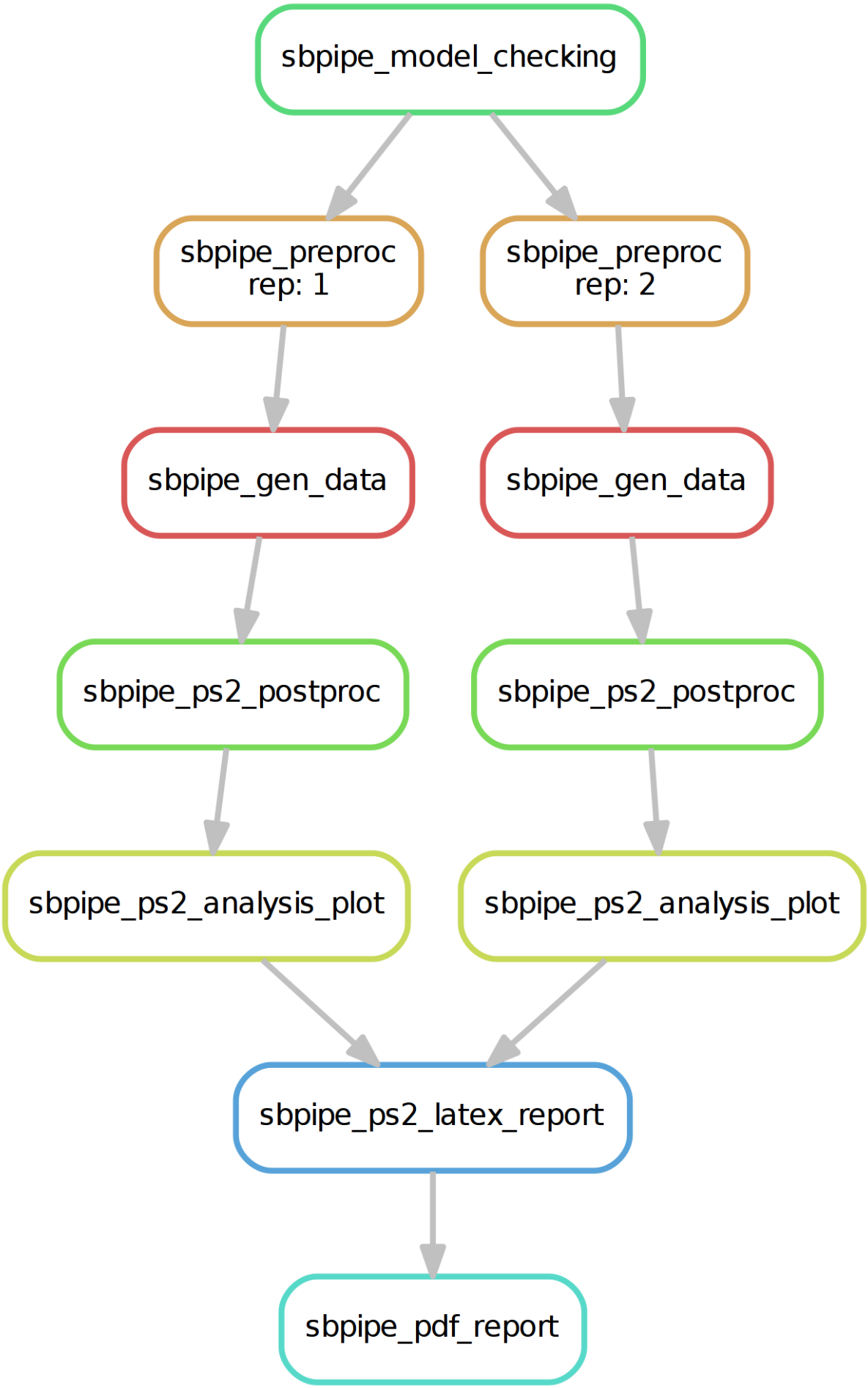
```
snakemake -s ../../sbpipe_sim.snake --configfile ir_model_stoch_simul.yaml -j 200 -
↪-latency-wait 100 --drmaa " -cwd -V -S /bin/sh"
```

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

The implementation of SBpipe pipelines for Snakemake is more scalable and allows for additional controls and resiliance.




sbpipe_model_checking
sbpipe_preproc
rep: 1
sbpipe_preproc
rep: 2
sbpipe_gen_data
sbpipe_gen_data
sbpipe_ps1_postproc
sbpipe_ps1_postproc
sbpipe_ps1_analysis_plot
sbpipe_ps1_analysis_plot
sbpipe_ps1_latex_report
sbpipe_pdf_report


sbpipe_model_checking
sbpipe_preproc
rep: 1
sbpipe_preproc
rep: 2
sbpipe_gen_data
sbpipe_gen_data
sbpipe_ps2_postproc
sbpipe_ps2_postproc
sbpipe_ps2_analysis_plot
sbpipe_ps2_analysis_plot
sbpipe_ps2_latex_report
sbpipe_pdf_report

3.3 Configuration for the mathematical models

SBpipe can run COPASI models or models coded in any programming language using a Python wrapper to invoke them.

3.3.1 COPASI models

A COPASI model must be configured as follow using the command `CopasiUI`:

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.

3.3.2 Python wrapper executing models coded in any language

Users can use Python as a wrapper to execute models (programs) coded in any programming language. The model must be functional and a Python wrapper should be able to run it via the command `python`. The program must receive the report file name as input argument (see examples in sbpipe/tests/). If the program generates a model simulation, a report file must be generated including the column `Time`. Report fields must be separated by TAB, and row names must be discarded. If the program runs a parameter estimation, a report file must be generated including 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.

The following example illustrates how SBpipe can simulate a model called `sde_periodic_drift.r` and coded in R, using a Python wrapper called `sde_periodic_drift.py`. Both the Python wrapper and R model are stored in the folder `Models/`. The idea is that the configuration file tells SBpipe to run the Python wrapper which receives the report file name as input argument and forwards it to the R model. After executing, the results are stored in this report, enabling SBpipe to analyse the results. The full example is stored in: `sbpipe/tests/r_models/`.

```
# Configuration file invoking the Python wrapper `sde_periodic_drift.py`
# Note that simulator must be set to "Python"
generate_data: True
analyse_data: True
generate_report: True
project_dir: "."
simulator: "Python"
model: "sde_periodic_drift.py"
```

(continues on next page)

(continued from previous page)

```
cluster: "local"
local_cpus: 7
runs: 14
exp_dataset: ""
plot_exp_dataset: False
exp_dataset_alpha: 1.0
xaxis_label: "Time"
yaxis_label: "#"
```

```
# Python wrapper: `sde_periodic_drift.py`.

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))
```

```
# R model `sde_periodic_drift.r`

# Model from https://cran.r-project.org/web/packages/sde/sde.pdf

# import sde package
# sde and its dependencies must be installed.
if(!require(sde)){
    install.packages('sde')
    library(sde)
}

# Retrieve the report file name (necessary for stochastic simulations)
args <- commandArgs(trailingOnly=TRUE)
report_filename = "sde_periodic_drift.csv"
if(length(args) > 0) {
    report_filename <- args[1]
}


# Model definition
# ----------------------------------------------
# set.seed()
d <- expression(sin(x))
d.x <- expression(cos(x))
A <- function(x) 1-cos(x)

X0 <- 0
delta <- 1/20
```

(continues on next page)

(continued from previous page)

```
N <- 500
time <- seq(X0, N*delta, by=delta)

# EA = exact method
periodic_drift <- sde.sim(method="EA", delta=delta, X0=X0, N=N, drift=d, drift.x=d.
↪x, A=A)

out <- data.frame(time, periodic_drift)
# ------------------------------------------

# Write the output. The output file must be the model name with csv or txt␣
↪extension.
# Fields must be separated by TAB, and row names must be discarded.
write.table(out, file=report_filename, sep="\t", row.names=FALSE)
```

4 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.

5 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/
        | - pl
        | - report
        | - simul
        | - tasks
        | - utils
  | - scripts/
  | - tests/
```

These folders will be discussed in the next sections. In SBpipe, Python is the project main language, whereas R is used for computing statistics 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. The R code for SBpipe is distributed as a separate R package and installed as a dependency using the provided script (install_rdeps.r) or conda. The source code for this package can be found here: https://github.com/pdp10/sbpiper and on CRAN https://cran.r-project.org/package=sbpiper.

5.1 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
- texlive-fonts-recommended
- texlive-latex-extra

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

To generate the source code documentation:

```
cd path/to/sbpipe/docs
./create_doc.sh
```

GitHub and ReadTheDocs.io are automatically configured to build the documentation in HTML and PDF format at every commit. These are available at: http://sbpipe.readthedocs.io.

5.2 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.

5.2.1 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.

5.2.2 report

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

5.2.3 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`.

5.2.4 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.

5.2.5 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.

5.3 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.

5.4 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`.

6 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.

6.1 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*.

6.2 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)
```

6.3 New releases

The script `release.sh` at the root of the package allows to release a new version of SBpipe or update the last github tag. This script also creates and uploads a new SBpipe package for Anaconda Cloud.

The following two sections describe how to release a new version for SBpipe, manually.

6.3.1 How to release a new tag

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.

```
# record the release add a tag:
git tag -a v1.3 -m "SBpipe v1.3"

# transfer the tag to the remote server:
git push origin v1.3   # Note: this goes to a separate 'branch'

# see all the releases:
git show
```

6.3.2 How to release a new SBpipe conda package (Anaconda Cloud)

This is a short guide for building SBpipe as a conda package. 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 -c pdp10 -c conda-forge -c fbergmann -c defaults
```

To test this package locally:

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

# uninstall
conda remove sbpipe
```

To upload the package to Anaconda Cloud repository:

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

7 Miscellaneous of useful commands

7.1 Git

Startup

```
# clone master
git clone https://github.com/pdp10/sbpipe.git
# get develop branch
git checkout -b develop origin/develop
# 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
```