SBpipe documentation

Release 4.10.0

Piero Dalle Pezze

Apr 11, 2018

Contents

1 Introduction **2**

2 Quick examples **2**
- 2.1 Model simulation . . . 3
- 2.2 Model parameter estimation . . . 5

3 Installation **8**
- 3.1 Requirements . . . 8
- 3.2 Installation on GNU/Linux . . . 9
  - 3.2.1 Installation of COPASI . . . 9
  - 3.2.2 Installation of LaTeX . . . 9
  - 3.2.3 Installation of SBpipe via Conda . . . 9
  - 3.2.4 Installation of SBpipe from source . . . 10
- 3.3 Installation on Windows . . . 11
- 3.4 Testing SBpipe . . . 11

4 How to use SBpipe **12**
- 4.1 Run SBpipe natively . . . 12
  - 4.1.1 Pipeline configuration files . . . 12
- 4.2 Run SBpipe via Snakemake . . . 16
- 4.3 Configuration for the mathematical models . . . 21
  - 4.3.1 COPASI models . . . 21
  - 4.3.2 Python wrapper executing models coded in any language . . . 21

5 How to report bugs or request new features **23**

6 Package structure **23**
- 6.1 docs . . . 24
- 6.2 sbpipe . . . 24
  - 6.2.1 pl . . . 24
  - 6.2.2 report . . . 25
  - 6.2.3 simul . . . 25
  - 6.2.4 tasks . . . 25
  - 6.2.5 utils . . . 25
- 6.3 scripts . . . 25
- 6.4 tests . . . 26

7 Development model **26**
- 7.1 Conventions . . . 26
- 7.2 Work flow . . . 26
- 7.3 New releases . . . 27

7.3.1 How to release a new tag . . . 27
7.3.2 How to release a new SBpipe conda package (Anaconda Cloud) . . . 27

8 Miscellaneous of useful commands **28**
8.1 Git . . . 28

9 License **29**

10 Change Log **32**

SBpipe allows mathematical modellers 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. SBpipe can run models implemented in COPASI, Python or coded in any other programming language using Python as a wrapper module. Pipelines can run on multicore computers, Sun Grid Engine (SGE), Load Sharing Facility (LSF) clusters, or via Snakemake (https://snakemake.readthedocs.io).

Project info

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

Mailing list: sbpipe@googlegroups.com

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

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

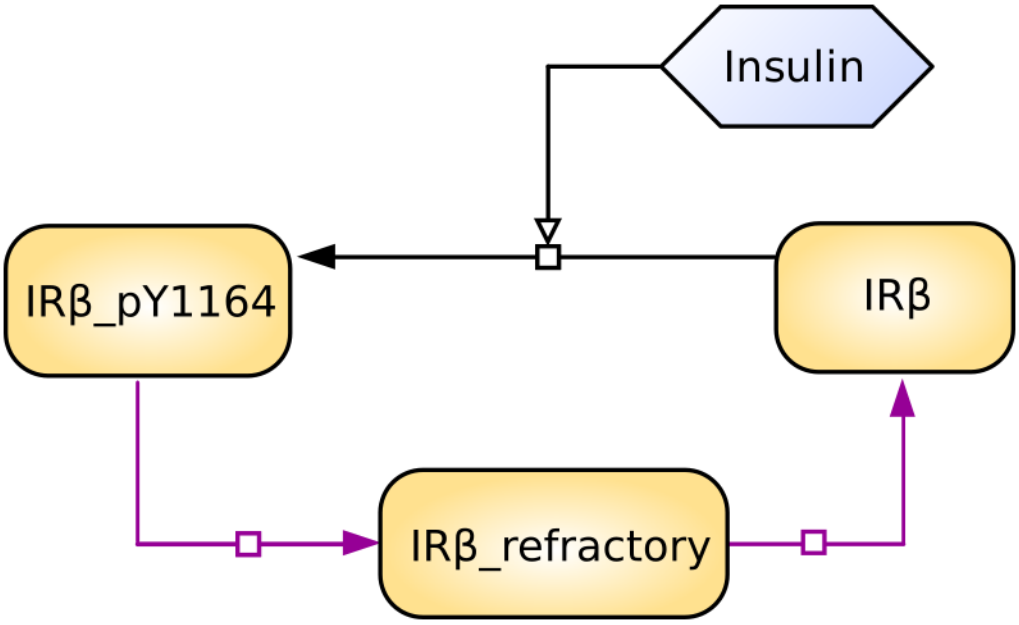
Citation: Dalle Pezze, P and Le Novère, N. (2017) *BMC Systems Biology* **11**:46. SBpipe: a collection of pipelines for automating repetitive simulation and analysis tasks. https://doi.org/10.1186/s12918-017-0423-3

1 Introduction

SBpipe is an open source software tool for automating repetitive tasks in model building and simulation. Using basic YAML configuration files, SBpipe builds a sequence of repeated model simulations or parameter estimations, performs analyses from this generated sequence, and finally generates a LaTeX/PDF report. The parameter estimation pipeline offers analyses of parameter profile likelihood and parameter correlation using samples from the computed estimates. Specific pipelines for scanning of one or two model parameters at the same time are also provided. Pipelines can run on multicore computers, Sun Grid Engine (SGE), or Load Sharing Facility (LSF) clusters, speeding up the processes of model building and simulation. If desired, pipelines can also be executed via Snakemake (https://snakemake.readthedocs.io), a powerful workflow management system. SBpipe can run models implemented in COPASI, Python or coded in any other programming language using Python as a wrapper module. Future support for other software simulators can be dynamically added without affecting the current implementation.

2 Quick examples

Here we illustrate how to use SBpipe to simulate and estimate the parameters of a minimal model of the insulin receptor.



To run this example Miniconda3 (https://conda.io/miniconda.html) must be installed. From a GNU/Linux shell, run the following commands:

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

# install LaTeX
conda install -c pkgw/label/superseded texlive-core=20160520 texlive-
↪selected=20160715

# install R dependencies (second example)
conda install r-desolve r-minpack.lm -c conda-forge

# create a project using the command:
sbpipe -c quick_example
```

2.1 Model simulation

For this example, the mathematical model is coded in Python. The following model file must be saved in `quick_example/Models/insulin_receptor.py`.

```
# insulin_receptor.py

import numpy as np
from scipy.integrate import odeint
import pandas as pd
import sys

# Retrieve the report file name (necessary for stochastic simulations)
report_filename = "insulin_receptor.csv"
if len(sys.argv) > 1:
    report_filename = sys.argv[1]


# Model definition
# ------------------------------------------------
def insulin_receptor(y, t, inp, p):
```

(continues on next page)

(continued from previous page)

```
    dy0 = - p[0] * y[0] * inp[0] + p[2] * y[2]
    dy1 = + p[0] * y[0] * inp[0] - p[1] * y[1]
    dy2 = + p[1] * y[1] - p[2] * y[2]
    return [dy0, dy1, dy2]

# input
inp = [1]
# Parameters
p = [0.475519, 0.471947, 0.0578119]
# a tuple for the arguments (see odeint syntax)
config = (inp, p)

# initial value
y0 = np.array([16.5607, 0, 0])

# vector of time steps
time = np.linspace(0.0, 20.0, 100)

# simulate the model
y = odeint(insulin_receptor, y0=y0, t=time, args=config)
# ----------------------------------------------


# Make the data frame
d = {'time': pd.Series(time),
     'IR_beta': pd.Series(y[:, 0]),
     'IR_beta_pY1146': pd.Series(y[:, 1]),
     'IR_beta_refractory': pd.Series(y[:, 2])}
df = pd.DataFrame(d)

# Write the output. The output file must be the model name with csv or txt␣
↪extension.
# Fields must be separated by TAB, and row indexes must be discarded.
df.to_csv(report_filename, sep='\t', index=False, encoding='utf-8')
```

We also add a data set file to overlap the model simulation with the experimental data. This file must be saved in `quick_example/Models/insulin_receptor_dataset.csv`. Fields can be separated by a TAB or a comma.

```
Time    IR_beta_pY1146
0   0
1   3.11
3   3.13
5   2.48
10  1.42
15  1.36
20  1.13
30  1.45
45  0.67
60  0.61
120 0.52
0   0
1   5.58
3   4.41
5   2.09
10  2.08
15  1.81
20  1.26
30  0.75
45  1.56
60  2.32
```

(continues on next page)

(continued from previous page)

```
120 1.94
0   0
1   6.28
3   9.54
5   7.83
10  2.7
15  3.23
20  2.05
30  2.34
45  2.32
60  1.51
120 2.23
```

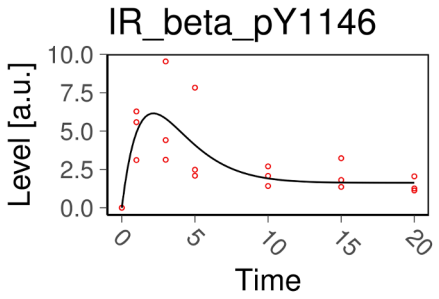
We then need a configuration file for SBpipe, which must be saved in `quick_example/insulin_receptor.yaml`

```
# insulin_receptor.yaml

generate_data: True
analyse_data: True
generate_report: True
project_dir: "."
simulator: "Python"
model: "insulin_receptor.py"
cluster: "local"
local_cpus: 4
runs: 1
exp_dataset: "insulin_receptor_dataset.csv"
plot_exp_dataset: True
exp_dataset_alpha: 1.0
xaxis_label: "Time"
yaxis_label: "Level [a.u.]"
```

Finally, SBpipe can execute the model as follows:

```
cd quick_example
sbpipe -s insulin_receptor.yaml
```

The folder `quick_example/Results/insulin_receptor` is now populated with the model simulation, plots, and a PDF report.



2.2 Model parameter estimation

For this example, the mathematical model is coded in R and a Python wrapper is used to invoke this model. The model and its wrapper file must be saved in `quick_example/Models/insulin_receptor_param_estim.R` and `quick_example/Models/insulin_receptor_param_estim.py`. This model uses the data set in the previous example.

```
# insulin_receptor_param_estim.R

library(reshape2)
library(deSolve)
library(minpack.lm)

# get the report file name
args <- commandArgs(trailingOnly=TRUE)
report_filename  <-  "insulin_receptor_param_estim.csv"
if(length(args) > 0) {
  report_filename <- args[1]
}

# retrieve the folder of this file to load the data set file name.
args <- commandArgs(trailingOnly=FALSE)
SBPIPE_R <- normalizePath(dirname(sub("^--file=", "", args[grep("^--file=",␣
↪args)])))

# load concentration data
df <- read.table(file.path(SBPIPE_R,'insulin_receptor_dataset.csv'), header=TRUE,␣
↪sep="\t")
colnames(df) <- c("time", "B")

# mathematical model
insulin_receptor <- function(t,x,parms){
  # t: time
  # x: initial concentrations
  # parms: kinetic rate constants and the insulin input
  insulin <- 1
  with(as.list(c(parms, x)), {
      dA <- -k1*A*insulin + k3*C
      dB <- k1*A*insulin - k2*B
      dC <- k2*B - k3*C
      res <- c(dA, dB, dC)
      list(res)
  })
}


# residual function
rf <- function(parms){
  # inital concentration
  cinit <- c(A=16.5607,B=0,C=0)
  # time points
  t <- seq(0,120,1)
  # parameters from the parameter estimation routine
  k1 <- parms[1]
  k2 <- parms[2]
  k3 <- parms[3]
  # solve ODE for a given set of parameters
  out <- ode(y=cinit,times=t,func=insulin_receptor,
             parms=list(k1=k1,k2=k2,k3=k3),method="ode45")

  outdf <- data.frame(out)
  # filter the column we have data for
  outdf <- outdf[ , c("time", "B")]
  # Filter data that contains time points where data is available
  outdf <- outdf[outdf$time %in% df$time,]
  # Evaluate predicted vs experimental residual
  preddf <- melt(outdf,id.var="time",variable.name="species",value.name="conc")
  expdf <- melt(df,id.var="time",variable.name="species",value.name="conc")
  ssqres <- sqrt((expdf$conc-preddf$conc)^2)
```

(continues on next page)

(continued from previous page)

```
  # return predicted vs experimental residual
  return(ssqres)
}

# parameter fitting using Levenberg-Marquardt nonlinear least squares algorithm
# initial guess for parameters
parms <- runif(3, 0.001, 1)
names(parms) <- c("k1", "k2", "k3")
tc <- textConnection("eval_functs","w")
sink(tc)
fitval <- nls.lm(par=parms,
                 lower=rep(0.001,3), upper=rep(1,3),
                 fn=rf,
                 control=nls.lm.control(nprint=1, maxiter=100))
sink()
close(tc)

# create the report containing the evaluated functions
report <- NULL;
for (eval_fun in eval_functs) {
  items <- strsplit(eval_fun, ",")[[1]]
  rss <- items[2]
  rss <- gsub("[[:space:]]", "", rss)
  rss <- strsplit(rss, "=")[[1]]
  rss <- rss[2]
  estim.parms <- items[3]
  estim.parms <- strsplit(estim.parms, "=")[[1]]
  estim.parms <- strsplit(trimws(estim.parms[[2]]), "\\s+")[[1]]
  rbind(report, c(rss, estim.parms)) -> report
}
report <- data.frame(report)
names(report) <- c("rss", names(parms))

# write the output
write.table(report, file=report_filename, sep="\t", row.names=FALSE, quote=FALSE)
```

```
# insulin_receptor_param_estim.py

# 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.

import os
import sys
import subprocess
import shlex

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

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

# we replace \\ with / otherwise subprocess complains on windows systems.
command = command.replace('\\', '\\\\')

# Block until command is finished
```

(continues on next page)

(continued from previous page)

```
subprocess.call(shlex.split(command))
```

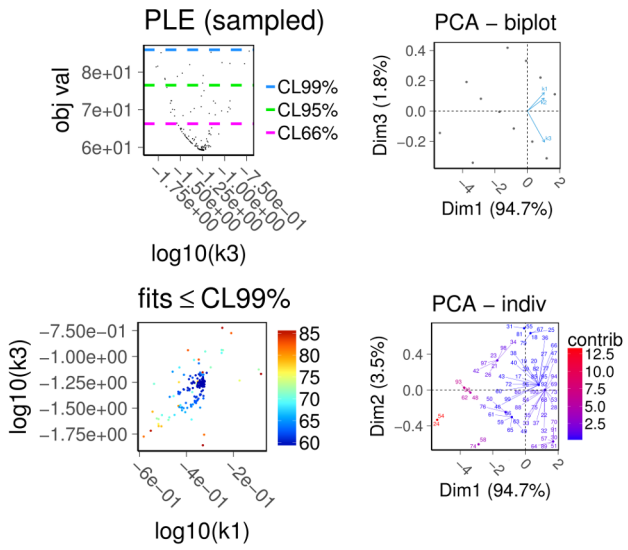
We then need a configuration file for SBpipe, which must be saved in `quick_example/insulin_receptor_param_estim.yaml`

```
# insulin_receptor_param_estim.yaml

generate_data: True
analyse_data: True
generate_report: True
project_dir: "."
simulator: "Python"
model: "insulin_receptor_param_estim.py"
cluster: "local"
local_cpus: 7
round: 1
runs: 100
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
```

Finally, SBpipe can execute the model as follows:

```
cd quick_example
sbpipe -e insulin_receptor_param_estim.yaml
```

The folder `quick_example/Results/insulin_receptor_param_estim` is now populated with the model simulation, plots, and a PDF report.



3 Installation

3.1 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+

3.2 Installation on GNU/Linux

3.2.1 Installation of 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)
```

3.2.2 Installation of 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).
```

3.2.3 Installation of SBpipe via Conda

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
```

3.2.4 Installation of SBpipe from source

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
```

(continues on next page)

(continued from previous page)

```
> 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.

3.3 Installation on Windows

See installation on GNU/Linux and install SBpipe via Conda. Windows users need to install LaTeX MikTeX https://miktex.org/.

3.4 Testing 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
```

4 How to use SBpipe

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

4.1 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
```

4.1.1 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
# 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"
```

(continues on next page)

(continued from previous page)

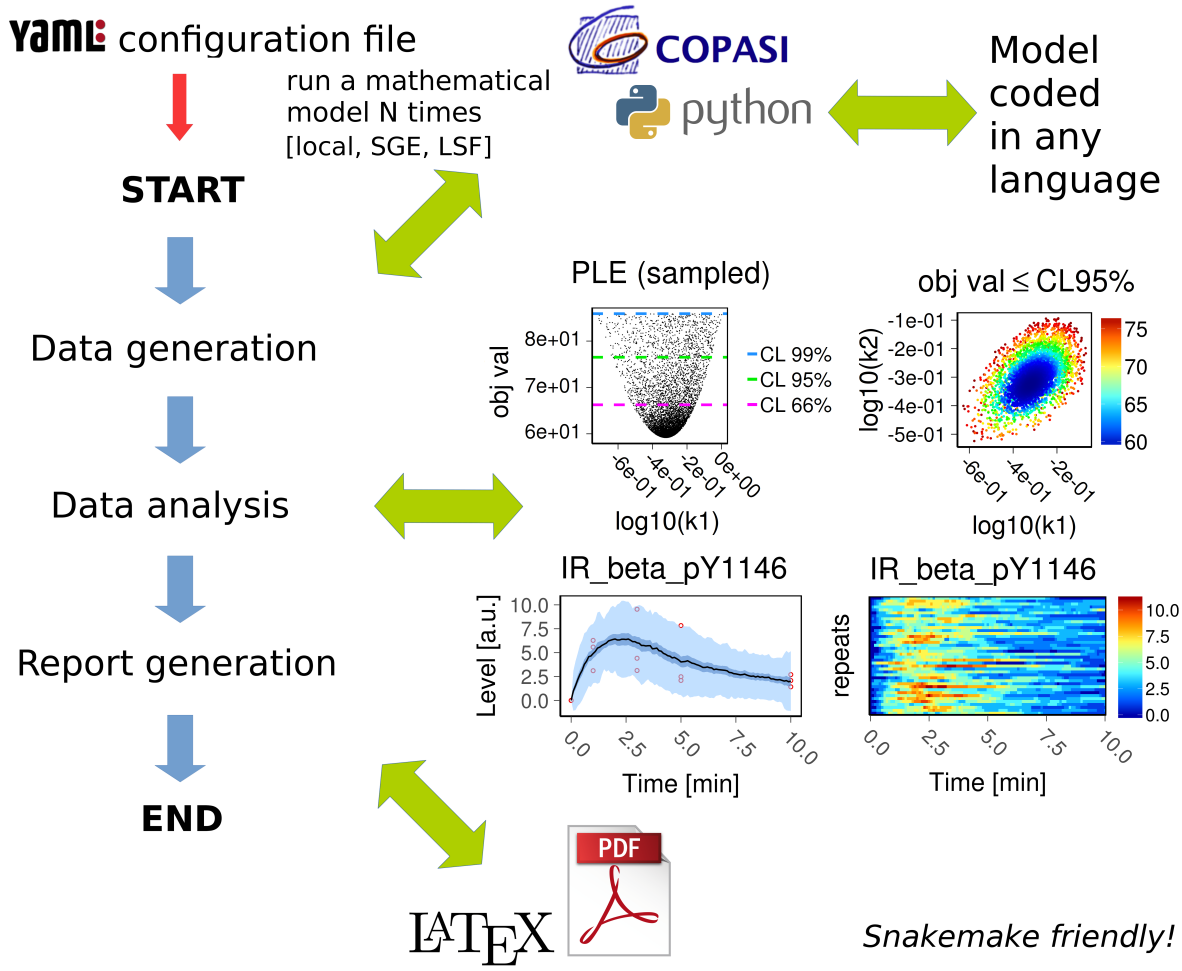
```
# The 2nd variable to scan (as set in Copasi Parameter Scan Task)
scanned_par2: "IRbetaPercent"
# The cluster type. local if the model is run locally,
# sge/lsf if run on cluster.
cluster: "local"
# The number of CPU if local is used, ignored otherwise
local_cpus: 7
# The number of simulations to 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: "."
# 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/
```



4.2 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`.

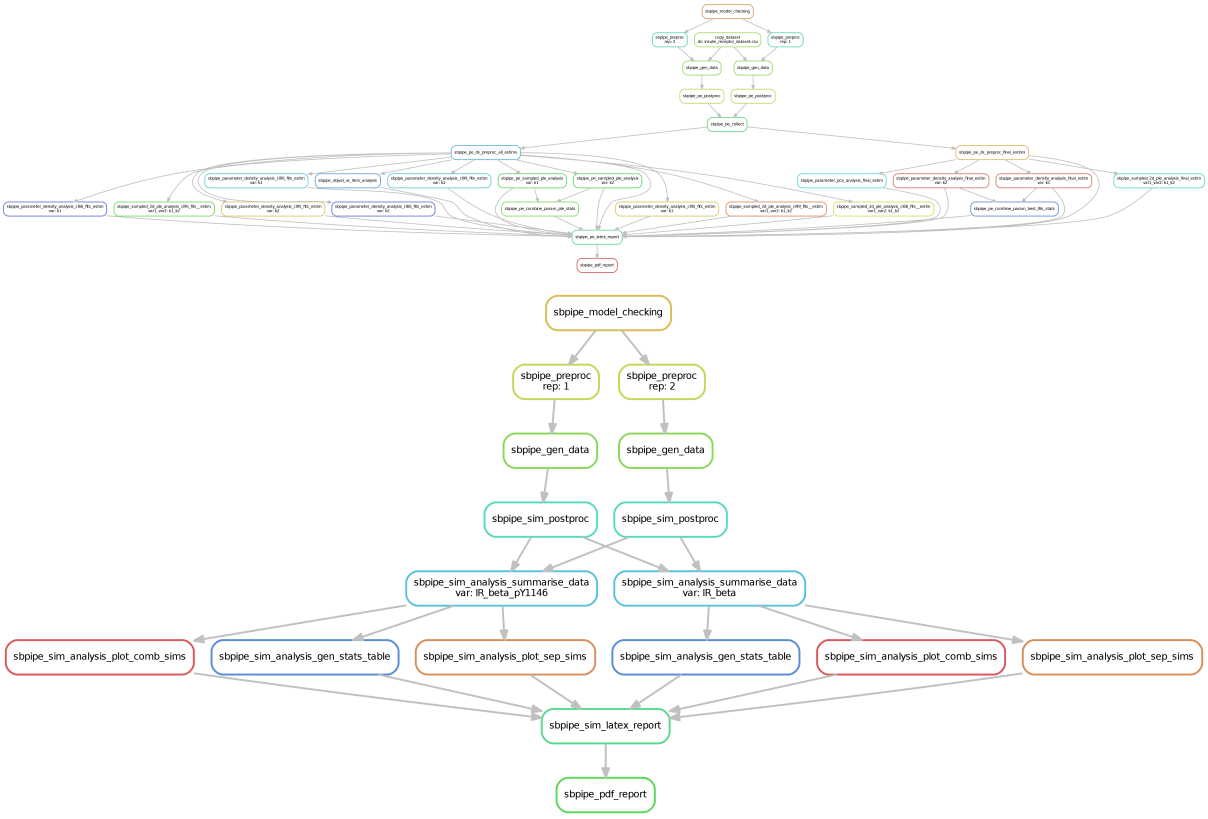
```
# See the DRMAA Python bindings for a preliminary documentation: https://pypi.
↪python.org/pypi/drmaa
# The following is an example of configuration for DRMAA for the grid engine␣
↪installed at the Babraham Institute
# (Cambridge, UK).

# 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
```

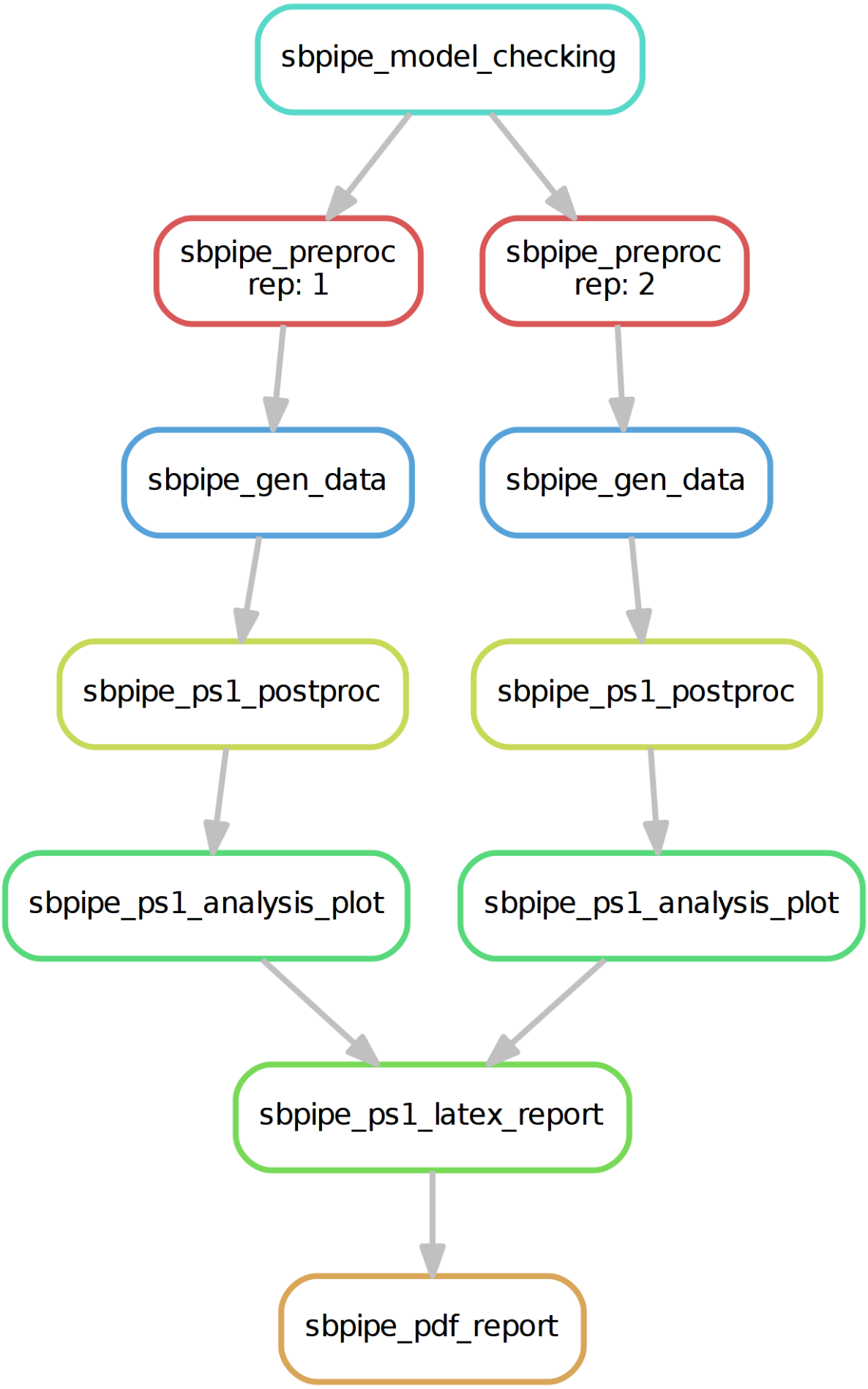
Snakemake can now be executed using drmaa as follows:

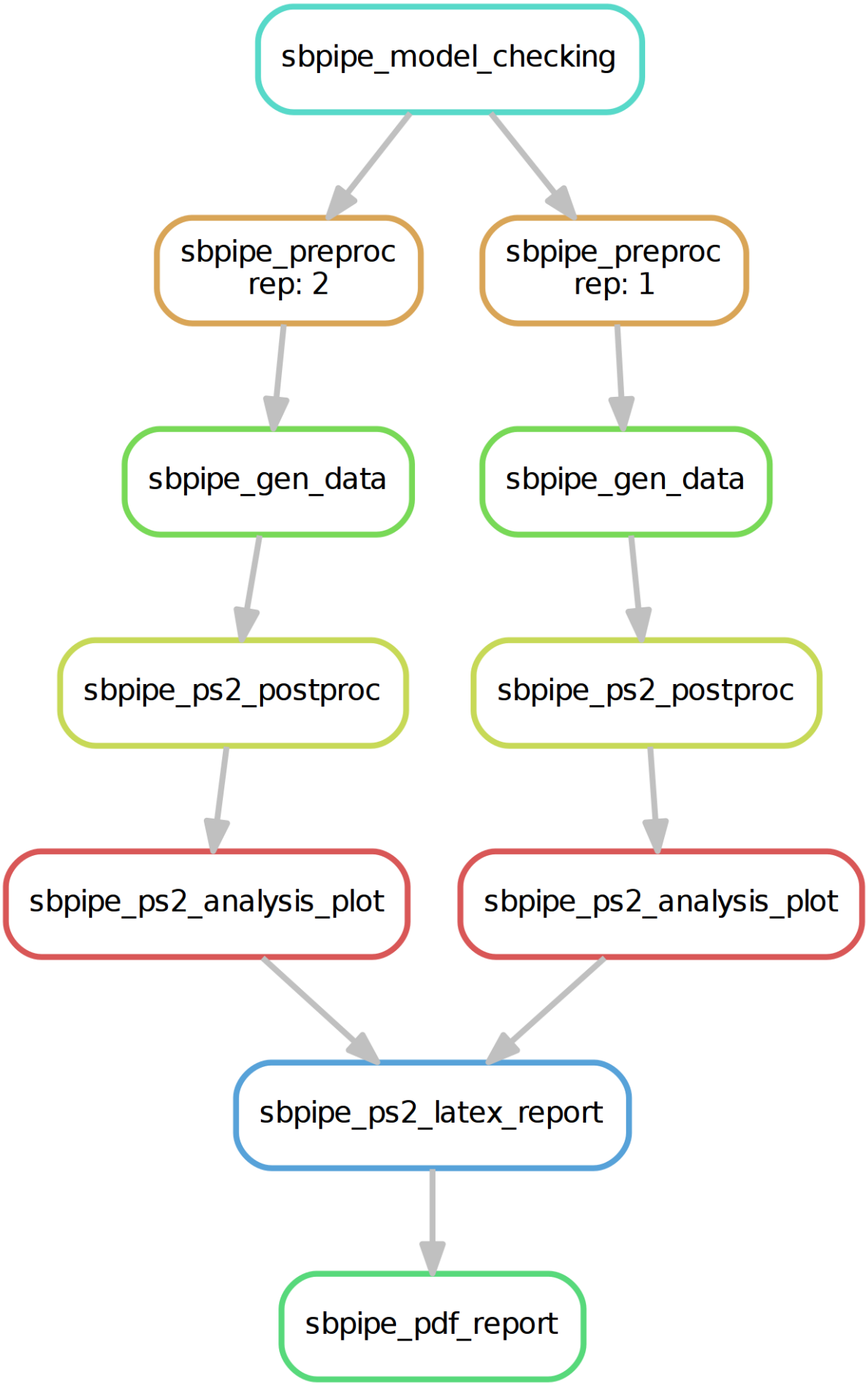
```
snakemake -s ../../sbpipe_sim.snake --configfile ir_model_stoch_simul.yaml -j 200 -
↪-latency-wait 100 --drmaa " -cwd -V -S /bin/sh"
```

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: 2
sbpipe_preproc
rep: 1
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

4.3 Configuration for the mathematical models

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

4.3.1 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.

4.3.2 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)
```

5 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.

6 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.

6.1 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.

6.2 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.

6.2.1 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.

6.2.2 report

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

6.2.3 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`.

6.2.4 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.

6.2.5 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.

6.3 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.

6.4 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`.

7 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.

7.1 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*.

7.2 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)
```

7.3 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.

7.3.1 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
```

7.3.2 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
```

8 Miscellaneous of useful commands

8.1 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 origin tagName

# 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
```

9 License

SBpipe is licensed under the GNU LGPLv3:

```
                   GNU LESSER GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


  This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

  0. Additional Definitions.

  As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

  "The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

  An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

  A "Combined Work" is a work produced by combining or linking an
Application with the Library.  The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.
```

(continues on next page)

(continued from previous page)

```
  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

   a) Give prominent notice with each copy of the Combined Work that
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
   document.

   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
```

(continues on next page)

(continued from previous page)

copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser

(continues on next page)

(continued from previous page)

```
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

  If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```

10 Change Log

```
sbpipe (GNU Lesser General Public License v3)
Copyright 2010-2018 Piero Dalle Pezze

Website: https://pdp10.github.io/sbpipe/


### CHANGELOG

v4.10.0 (Beyond the Kuiper Belt)

- Documentation update.
- Added generate_tarball option to all the remaining pipelines in native SBpipe.
- Improved output messages.
- Added progress information for native SBpipe.
- Added PCA analysis for the best parameter estimates. Replaced conda channel "r"␣
↪with "conda-forge".
- Improved data analysis scalability for parameter estimation (using Snakemake).
- Added checks whether a COPASI model can be loaded and executed correctly. This␣
↪is based on Python bindings for COPASI.
- Optimisation of snakemake pipelines. Improved efficiency of rules for analyses.
- Bugfix - SGE and LSF job names now include a random string, avoiding potential␣
↪interactions among multiple
SBpipe executions. Whilst this does not affect the results, it was still a␣
↪performance-related bug.
- SBpipe R code is now an independent R package called sbpiper. This is imported␣
↪by SBpipe as
an external dependency. Users can invoke SBpipe functions for data analysis␣
↪directly from their R code.

v4.0.0 (Mars)

- added option `exp_dataset_alpha` to `sim` pipeline. This option allows to plot␣
↪experimental
data with an alpha level.
- data analysis for `sim` pipeline is scalable.
- improved yaml files for installing SBpipe using conda. SBpipe is now tested on␣
↪Python 2.7 and 3.6.
- added transparencies and improved simulation plots combined with data set.
- added release.sh script for releasing SBpipe versions automatically.
- if `data_point_num` is [0, est_param_number], the analysis task for parameter␣
↪estimation will
continue BUT the thresholds will be discarded.
- bug fix - conda build package after conda was upgraded to v3.x.x
- bug fix - constraints in parameter estimation using Copasi
- added scripts
- code optimisation for parameter estimation pipeline.
- Improved conda packaging.
```

(continues on next page)

(continued from previous page)

```
- Improved import of parameter names for parameter estimation pipeline.
- Improved SBpipe packaging (snakemake is not a requirement)
- SBpipe pipelines are also available as snake files. Therefore, SBpipe can be run␣
↪using Snakemake.
- SBpipe is now also available as conda package (installation+dependencies: conda␣
↪install -c pdp10 sbpipe)
- The environment variable SBPIPE is no longer necessary.
- Anaconda can be used for installing SBpipe dependencies. This improves␣
↪portability on Linux
and Windows OS.
- subprocess.Popen() and logging fileConfig() use `with .. as ..:` construct with␣
↪Python3+.
- changed `chi^2` label to `obj val` in parameter estimation plots
- Added additional arguments to sbpipe
- Output is now coloured.
- Improved logging messages. Added log.debug() calls.
- Added sbpipe() function in main.py to facilitate programmatic use of sbpipe.
- Replaced Python getopt with argparse.
- Improved unit tests and nosetests with Travis-CI.
- Replaced INI configuration files with YAML configuration files.
- Skip heatmap and multiple time course plot if only one simulation is run. These␣
↪plots are
just redundant.
- removed support for running R, Octave, and Java models directly as these can be␣
↪run via
a Python model wrapper.
- bug fixes

v3.0.0 (Earth)

- added prints to r plotting functions
- pipeline analyses are executed on cluster (local, sge, lsf) using sbpipe parcomp␣
↪module.
- renamed option `pp_cpus` to `local_cpus`, after removal of parallel python.
- renamed value `pp` to `local` for option `cluster` after removal of parallel␣
↪python.
- added support for Python 3. The code is now expected to work for Python 2.7+, 3.
↪2, and 3.6.
- replaced parallel python with python multiprocessing package. This should␣
↪facilitate
the transition to Python 3.
- removed deprecated source code for manually randomising parameters before␣
↪parameter
estimation in Copasi files.
- Copasi and PL-based simulators now share a large amount of code.
- adapted programming language-based simulators to use ps1 and ps2 post-processing␣
↪code.
All simulators support all the pipelines.
- moved post-processing code for ps1 and ps2 from Copasi to Simul.
- improved code cohesion by moving utility code into Simul class().
- improved output name consistency for report and plot files
- improved sorting of plots in latex/pdf report for ps1 pipeline
- added test case for stochastic double parameter scan.
- double parameter scans can be executed in parallel as repeats.
- added support for stochastic double parameter scans.
- added test case for stochastic single parameter scan.
- single parameter scans can be executed in parallel as repeats.
- added support for stochastic single parameter scans.
- modularised parallel computation within Copasi simulator
- added heatmap plot representing stochastic repeats for the time course␣
↪simulation.
```

(continues on next page)

(continued from previous page)

```
- moved R code from pipelines to R/
- redesign of simulation plots. Improvements plus based on melt function.
- removed remaining old gplots dependent code. Sbpipe only uses ggplot2 now.
- added two new plots useful for stochastic simulation
- improved reuse for all R plots.
- added plots reproducing all the single simulations per species.
- changed main script name from run_sbpipe.py to sbpipe.
- Added support for parameter estimation using non-Copasi models. Test using R␣
↪model.
- Skip Java, Python, and R model tests if their dependencies are not satisfied.
- Optimised Java, Python, and R simulators. Report file names are passed as input␣
↪argument.
Models do not need to be replicated.
- Java models can be used for model simulation in addition to Copasi.
- Python models can be used for model simulation in addition to Copasi.
- R models can be used for model simulation in addition to Copasi.


v2.0.0 (Venus)

- improved threshold levels for Sampled PLE plots.
- added 20 tests including wrong configuration file settings.
- extensive refactory of unit tests.
- source code uses PEP8 standard
- source code cleaning and reformatting.
- improved source code by eliminating some warning highlighted by PyCharm.
- moved script core functions within sbpipe package.
- the copasi package is now a dynamically loaded simulator. Users can choose the␣
↪simulator
to use in the configuration file.
- simulators are loaded dynamically. Uncoupling between simulators and pipelines.
- separation of code for generating data from pipeline package.
- improved source code modularisation for the whole program.
- extracted scripts (run_sbpipe and cleanup_sbpipe) from sbpipe/.
- sbpipe supports execution as a Python module (__main__.py).
- all Python imports are now absolute (in agreement with Python 3).
- improvements to program prints
- project renamed sbpipe
- added AIC, AICc, BIC to the parameter estimation summary table.
- randomisation of initial parameter values for parameter estimation is now only
performed by Copasi.
- added plots comparing model simulation vs experimental data in simulate pipeline.
- improving plot margins for simulate and single parameter scan pipelines.
- source code refractoring in parameter estimation analysis task.
- fixed a bug in parameter estimation pipeline related to the filtering of␣
↪confidence
intervals from the complete data set.
- added ratios in parameter estimation summaries to investigate the distance␣
↪between the
estimated parameter and its confidence intervals.
- plot polishing.
- separated options for plotting 2d correlations within 66%, 95%, or 99%␣
↪confidence intervals.
- added 99% confidence intervals parameter estimation plots.
- added option to plot parameter estimation plots using the scientific notation.
- improved plots layout (fonts, legends).
- added option for y axis label to simulate and single parameter scan pipelines.
- Copasi models are fully consistent.
- table of estimated parameter and confidence values is in normal scale (not␣
↪log10).
```

(continues on next page)

(continued from previous page)

```
v1.0.0 (Mercury)

- completed source code documentation
- completed user and developer manuals.
- configured Python Sphinx for documenting SBpipe
- bug fixes.
- separation of pdf report code from pipelines.
- configuration sessions integrated in pipeline classes.
- pipelines converted to classes.
- added option for plotting parameter estimation results in log10 parameter space␣
↪(default).
- improved heat palette for double parameter scan and coloured scatterplots.
- added test files for double parameter scan
- ported all Matlab code to Python / R
- added pipeline for double parameter scan (parsing, plots, report)
- further removal of deprecated files
- generated copasi files for parameter estimation now moved to Working_Folder/xx/
- improved insulin receptor model for testing.
- Copasi report files now in Models/ .
- Copasi experimental data files now in Models/ .
- added scripts for automatically installing Python and R package dependencies.
- use of sections in configuration
- separation of configuration file parsing from program logic.
- restructuring dataset parsing for simulate and single_param_scan.
- added parameter scan plot with homogeneous lines (useful for plotting param conf.
↪ interv.).
- replaced all prints with Python logging.
- improved LaTeX reports
- tested parameter estimation using Gillespie algorithm for model simulation.
- configured Travis-CI for continous integration tests.
- pipeline renaming.
- added computation for parameter confidence intervals.
- added plot for fit history.
- added 2D parameter correlations using 66% or 95% confidence levels from␣
↪calculated PLE.
- added profile likelihood estimation based on intermediate estimations.
- cleaned pipeline output.
- added documentation for configuring Copasi.
- removed part of the deprecated code.
- internalised code for each pipeline; run_sbpipe.py is the main executor for␣
↪sbpipe.
- bug fixes.
- models can now be simulated in parallel using PP, SGE, or LSF.
- separation of parallel code from param_estim__copasi pipeline. It is generic now.
- sbpipe should now be platform independent (untested yet).
- removed unused dependencies.
- better separation of test cases.
- pipeline steps can be executed separately.
- pipeline restructuring (separation of the steps: generate data, analyse data,␣
↪and generate report).
- model parameters can now be estimated in parallel using PP, SGE, or LSF.
- removed old deprecated code.
- restructuring source code in the lib/ folder (now sbpipe/pipelines and sbpipe/
↪utils).
- finalised skeleton for sb_param_estim pipeline.
- added parameter correlation plots for sb_param_estim pipeline.
- ported R gplots code to ggplot in sb_param_scan__single_perturb pipeline.
- ported R gplots code to ggplot in sb_simulate pipeline.
- sbpipe is now a Python package.
- added documentation (readme, developer_guide).
```

(continues on next page)

(continued from previous page)

```
- added unit tests and setup.py.
- ported Bash / sed / grep and cut code to Python in sb_param_estim pipeline.
- ported Bash / sed / grep and cut code to Python in sb_param_scan__single_perturb␣
↪pipeline.
- ported Bash / sed / grep and cut code to Python in sb_simulate pipeline.
- added param_estim__copasi.sh.
- improved configuration file.
- simulation time start, end, xaxis label and time step now replace the parameter␣
↪`team`.
- adjusted sb_simulate.sh, sb_param_scan__single_perturb.sh, sb_sensitivity.sh.
- packaging of sb_modules in /bin.
- added test scripts.
```