# SBpipe documentation

***Release 4.21.0***

**Piero Dalle Pezze**

**Jun 26, 2018**

## Contents

---

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.

**Project info**



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



Home Page: http://sbpipe.readthedocs.io

Anaconda Cloud: https://anaconda.org/bioconda/sbpipe

PyPI: https://pypi.org/project/sbpipe

sbpiper (R dependency for data analysis): https://cran.r-project.org/package=sbpiper

sbpipe_snake (Snakemake workflows for SBpipe): https://github.com/pdp10/sbpipe_snake

Mailing list: sbpipe@googlegroups.com

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

GitHub (dev): https://github.com/pdp10/sbpipe

Travis-CI (dev): https://travis-ci.org/pdp10/sbpipe

Issues / Feature requests (dev): https://github.com/pdp10/sbpipe/issues

Citation: Dalle Pezze P, Le Novère N. SBpipe: a collection of pipelines for automating repetitive simulation and analysis tasks. *BMC Systems Biology*. 2017 Apr;11:46. 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.

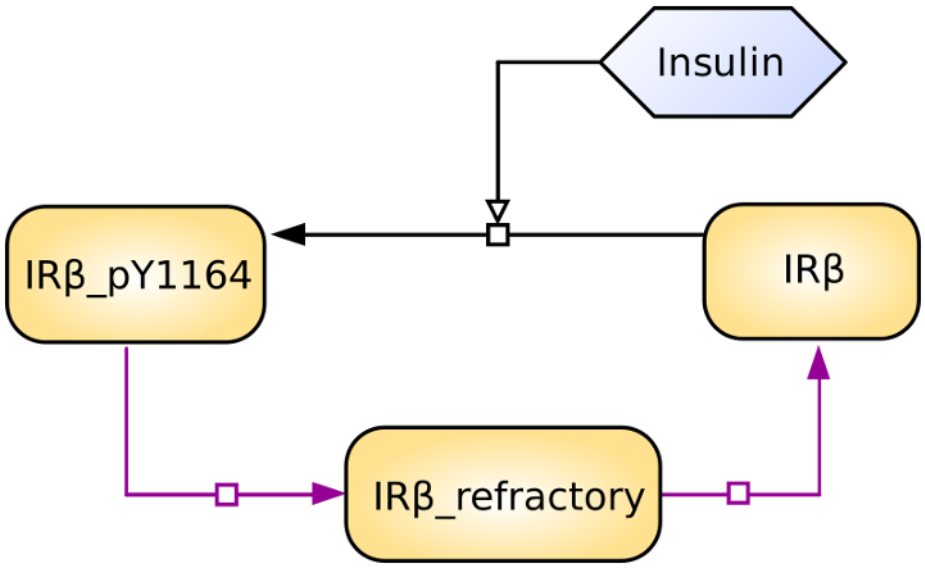



Fig. 1: Minimal model of the insulin receptor

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

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

(continues on next page)

(continued from previous page)

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

# install sbpipe and its dependencies (including sbpiper)
conda install -c bioconda sbpipe

# 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

This example should complete within 1 minute. 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):
    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)
# -----------------------------------------------
```

(continues on next page)

(continued from previous page)

```
# 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
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
```

(continues on next page)

(continued from previous page)

```
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.

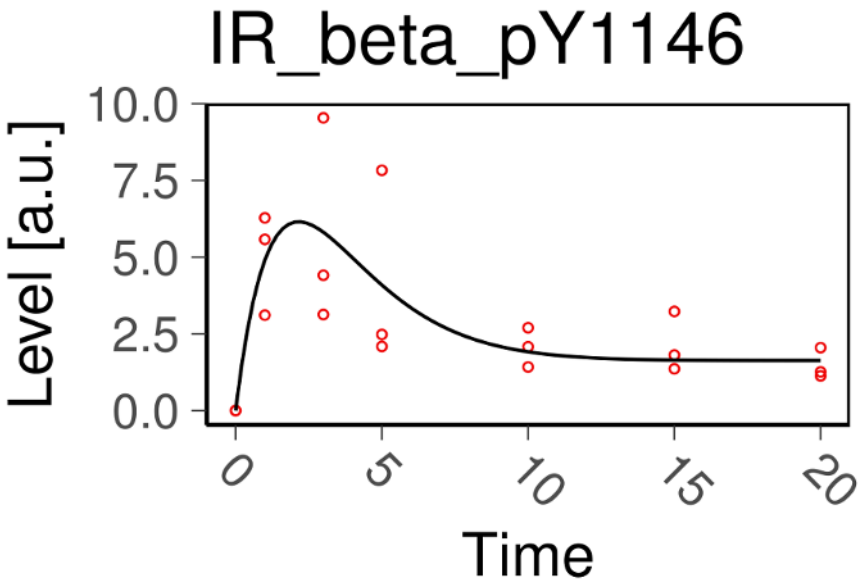



Fig. 2: model simulation

## 2.2 Model parameter estimation

This example should complete within 5 minutes. 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)
```

(continues on next page)

(continued from previous page)

```
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=',')
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)

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

(continues on next page)

(continued from previous page)

```
}

# 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
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: 50
best_fits_percent: 75
data_point_num: 33
plot_2d_66cl_corr: True
plot_2d_95cl_corr: True
plot_2d_99cl_corr: True
logspace: False
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/

Please, make sure that *Python pip* and *Rscript* work fine. 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:

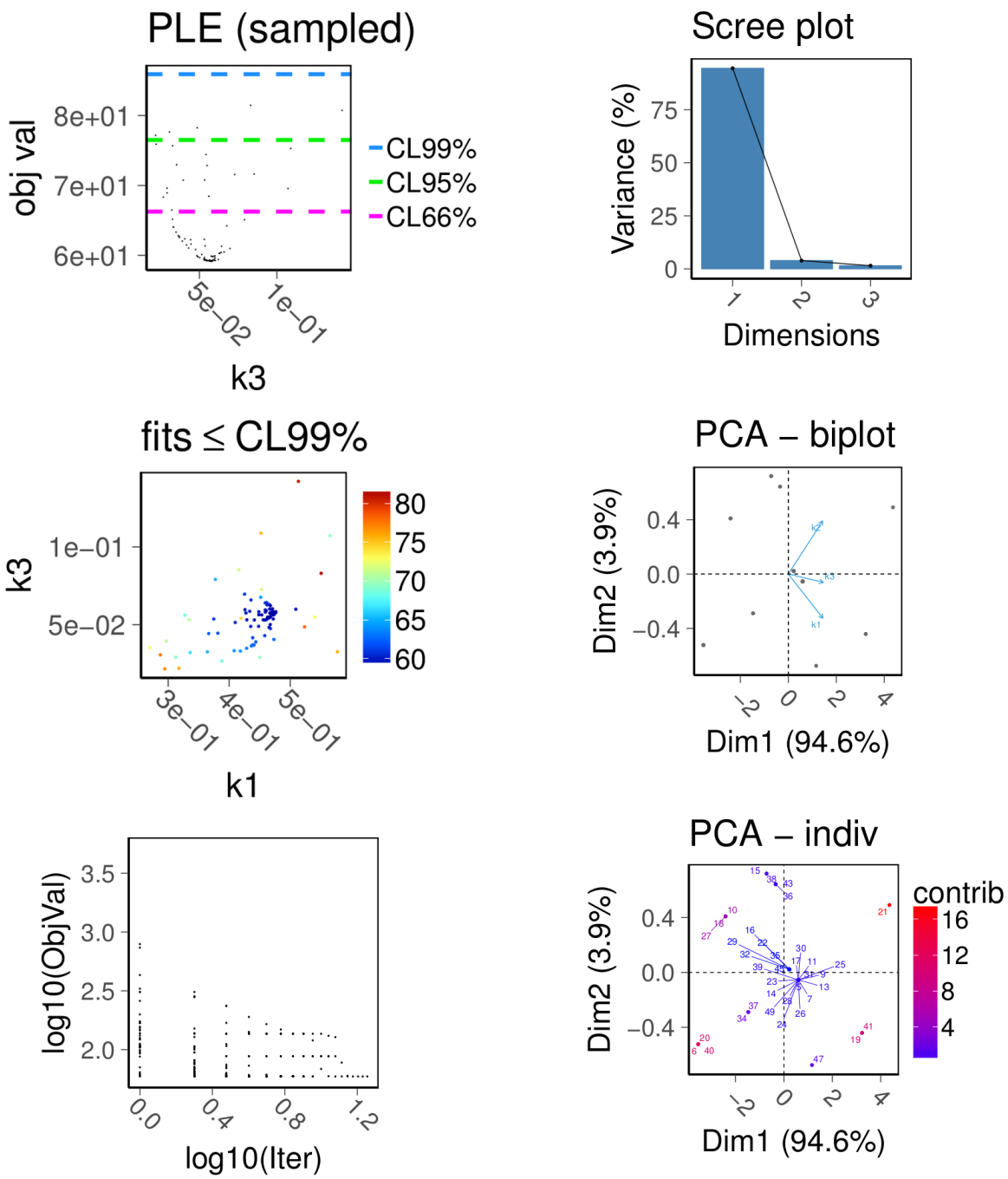



Fig. 3: model parameter estimation

```
# 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 Python pip

SBpipe and its Python dependencies can simply be installed via Python pip using the command:

```
# install sbpipe from pypi.org via pip
pip install sbpipe
```

In order to analyse the data, SBpipe requires the R package `sbpiper`, which can be installed as follows:

```
# install sbpiper from r-cran
Rscript -e "install.packages('sbpiper', dep=TRUE, repos='http://cran.r-project.org
↪')"
```

### 3.2.4 Installation of SBpipe via Conda

Users need to download and install Miniconda3 (https://conda.io/miniconda.html). SBpipe will be installed in a dedicated conda environment:

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

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

# install sbpipe and its dependencies (including sbpiper)
conda install -c bioconda sbpipe
```

### 3.2.5 Installation of SBpipe from source

Users need to install `git`.

```
# clone SBpipe from GitHub
git clone https://github.com/pdp10/sbpipe.git
# move to sbpipe folder
```

(continues on next page)

(continued from previous page)

```
cd sbpipe
# install SBpipe dependencies on GNU/Linux, run:
make
```

Finally, to run sbpipe from any shell, users need to add 'sbpipe/scripts' to the `PATH` environment variable by adding the following lines to the `$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
```

## 3.3 Installation on Windows

See installation on GNU/Linux and install SBpipe via PIP or 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 4.13.0
```

Unless SBpipe was installed from source, users need to download the source code at the page https://github.com/pdp10/sbpipe/releases to run the test suites.

```
# unzip and change path
unzip sbpipe-X.Y.Z.zip
cd sbpipe-X.Y.Z/tests
```

```
# run model simulation using COPASI (see results in tests/copasi_models):
nosetests test_copasi_sim.py --nocapture
```

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

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

# 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.
```

(continues on next page)

(continued from previous page)

```
# 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"
# 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
```

(continues on next page)

(continued from previous page)

```
# 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 Snakemake workflows for SBpipe

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 via

```
# Python pip
pip install snakemake
# or via conda:
conda install -c bioconda snakemake
```

The Snakemake workflows for SBpipe can be retrieved as follows:

```
# clone workflow into working directory
git clone https://github.com/pdp10/sbpipe_snake.git path/to/workdir
cd path/to/workdir
```

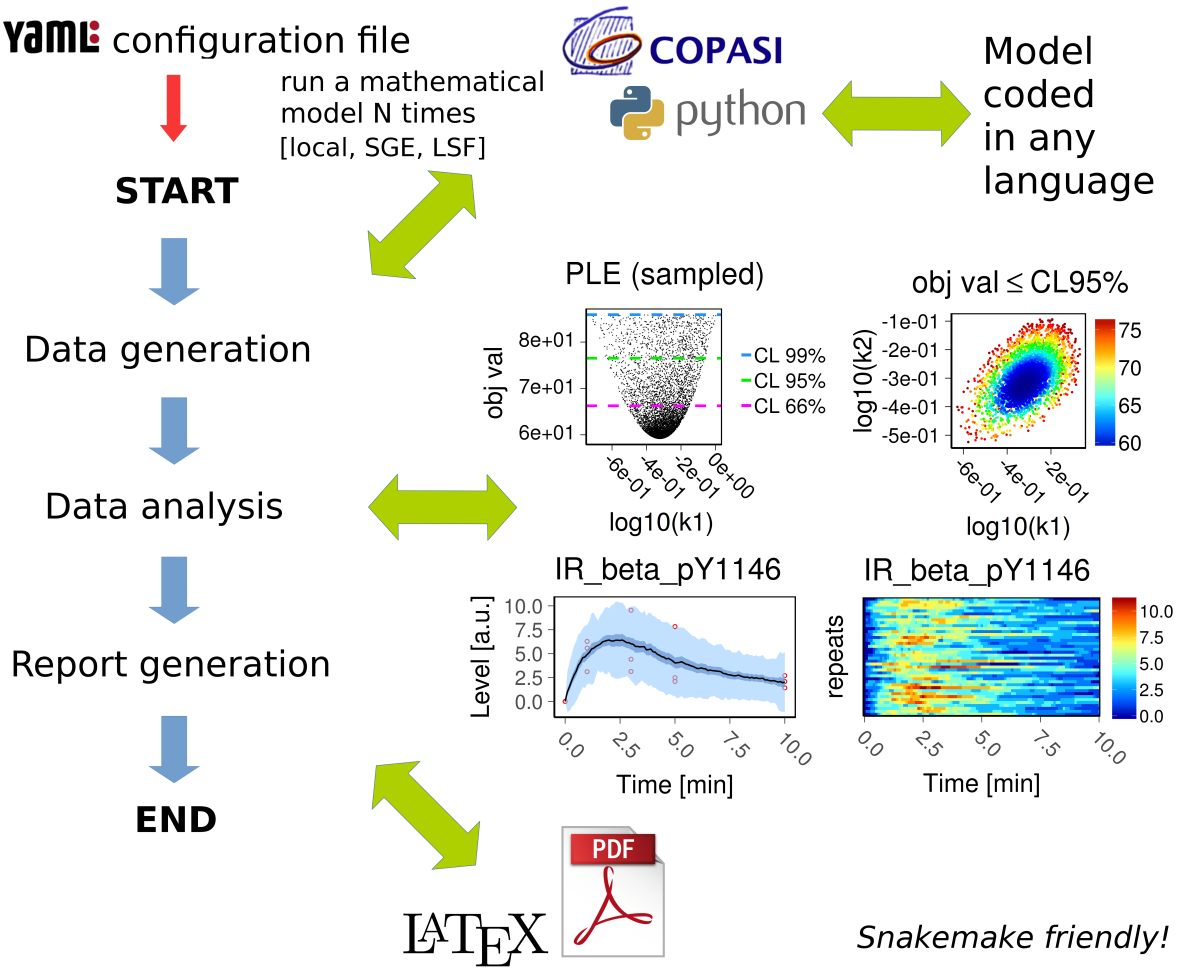



Fig. 4: SBpipe native workflow

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

An advantage of using snakemake as pipeline infrastructure is that it offers an extended command sets compared to the ones 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 using Snakemake, the configuration files described above require the additional field:

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

which contains 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 file 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 the Snakemake workflow for SBpipe are in `tests/snakemake`. For testing those workflows, both SBpipe and Snakemake must be installed. The workflows must be retrieved from the github repository as explained previously, and placed in the folder containing the Snakemake configuration files. Both the generic and the Snakemake test suites retrieve these workflows automatically, and store them in `tests/snakemake`.

Examples of commands running SBpipe pipelines using Snakemake are:

```
# run model simulation
snakemake -s 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 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 sbpipe_ps1.snake --configfile SBPIPE_CONFIG_FILE.yaml -j 5 --cluster
↪"bsub"

# run model parameter parameter scan using 5 jobs
snakemake -s 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 SBPIPE_CONFIG_FILE.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.

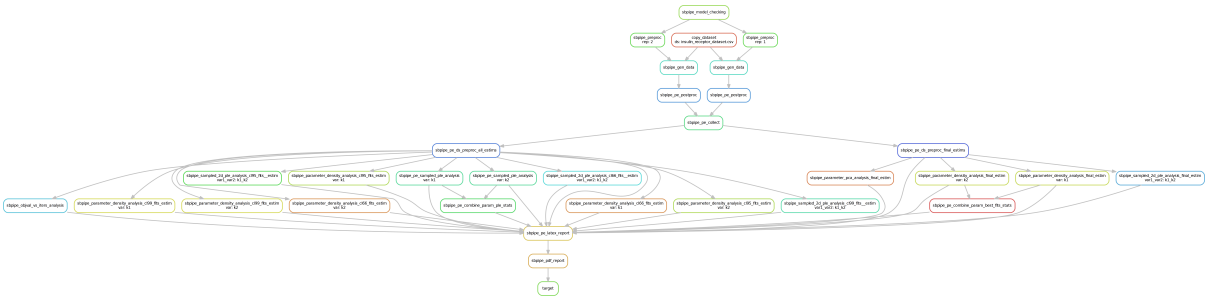


Fig. 5: Snakemake workflow for SBpipe pipeline `parameter estimation`


Fig. 6: Snakemake workflow for SBpipe pipeline `simulation`

## 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).

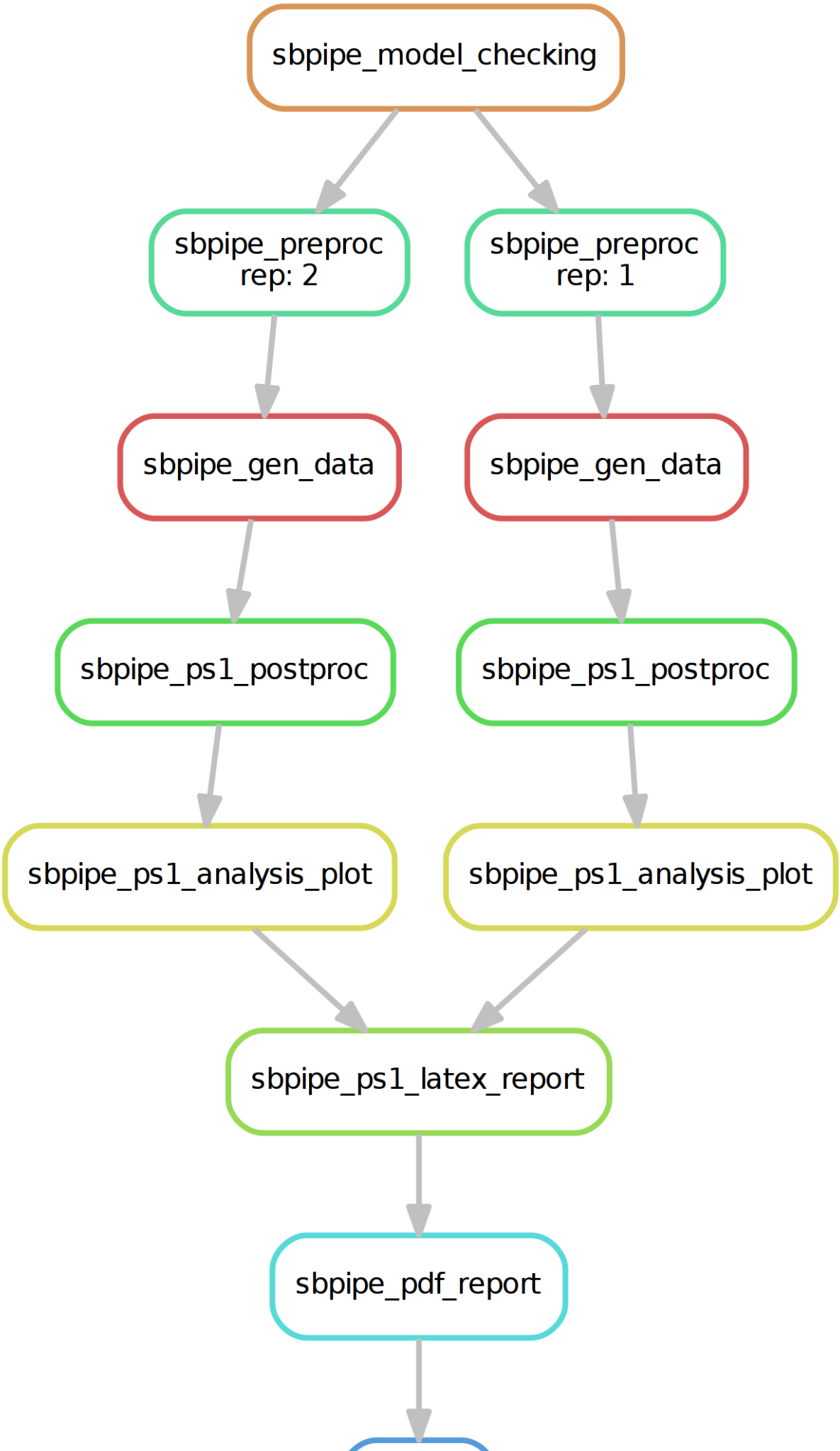
**pipelines: single or double parameter scan**

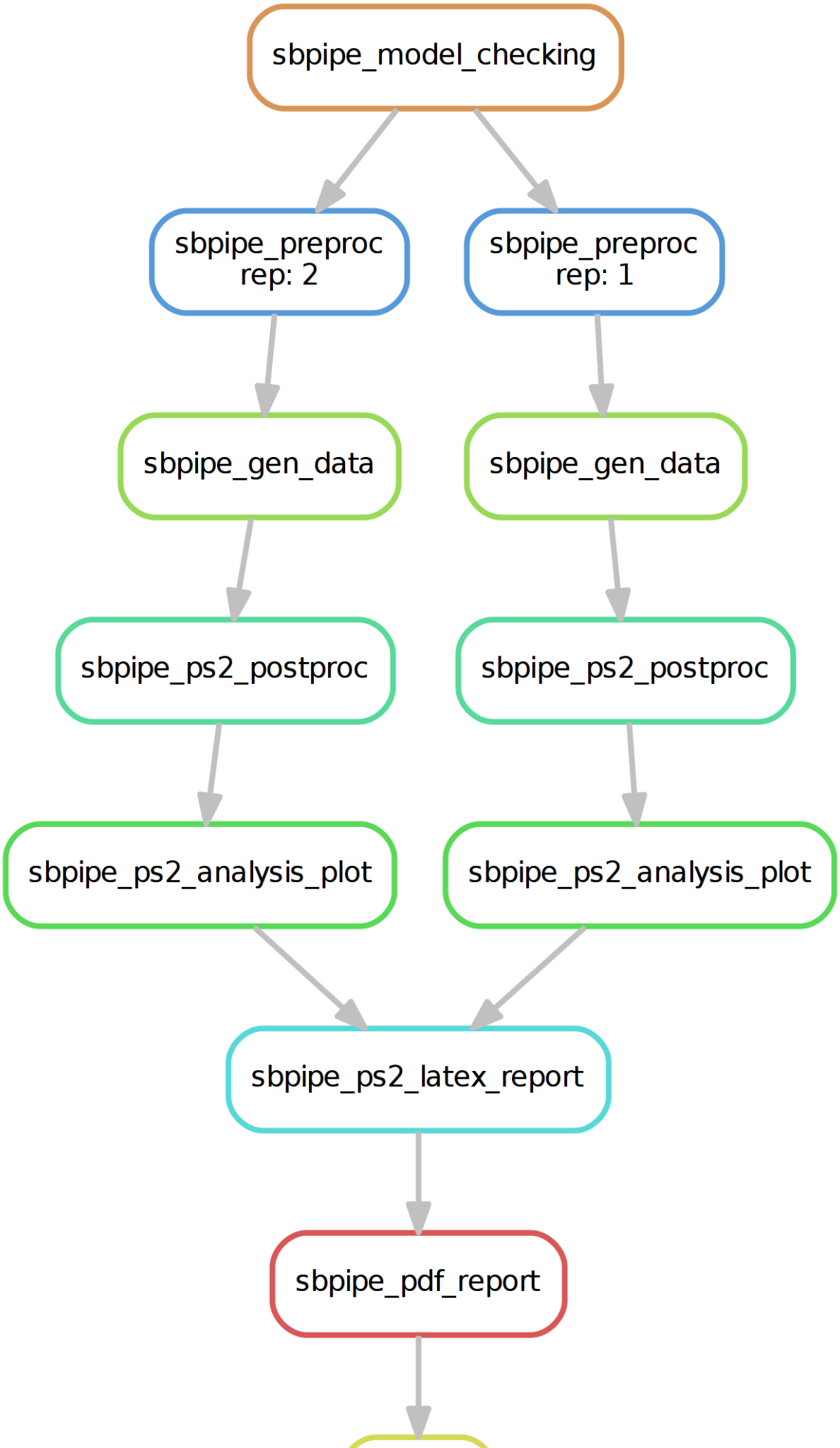
- Tick the flag *executable* in the Parameter Scan Task.
- Select a report template for the Parameter Scan Task.
- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv (extensions .txt, .tsv, or .dat are also accepted by SBpipe)

**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.


sbpipe_model_checking
sbpipe_preproc
rep: 2
sbpipe_preproc
rep: 1
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

### 4.3.2 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"
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
```

(continues on next page)

(continued from previous page)

```
# 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
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 Issues / Feature requests

SBpipe is a relatively young project and there is a chance that some error occurs. Issues and feature requests can be notified using the github issue tracking system for SBpipe at the web page: https://github.com/pdp10/sbpipe/issues. 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.

Please, use the following mailing list for generic questions: sbpipe@googlegroups.com. The topics discussed in this mailing list are also available at the website: https://groups.google.com/forum/#!forum/sbpipe.

# 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
        | - snakemake
        | - 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. 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 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 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 snakemake

The subpackage `sbpipe.snakemake` contains the Python scripts to invoke the single SBpipe tasks. These are invoked by the rules in the snakemake files. These snakemake workflows for SBpipe are stored in https://github.com/pdp10/sbpipe_snake.git .

### 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 GitHub 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 is used to release a new version of SBpipe on:

- github
- pypi
- anaconda cloud (channel: pdp10)

## 7.4 Conda releases

This is a short guide for building a conda package for SBpipe. Miniconda3 and the conda package `conda-build` must be installed:

```
conda install conda-build
```

SBpipe is stored in two Anaconda Cloud channels:

- `bioconda` (official)
- `pdp10` (for testing purposes **only**)

### 7.4.1 How to release the conda package of SBpipe on the bioconda channel (Anaconda Cloud)

This conda repository is used for storing the stable releases of SBpipe and sbpiper. More documentation can be found here:

- https://bioconda.github.io/
- https://bioconda.github.io/contribute-a-recipe.html
- https://bioconda.github.io/guidelines.html#guidelines

The first step is to setup a repository forked from bioconda-recipes:

```
# fork the GitHub repository: https://github.com/bioconda/bioconda-recipes.git

# clone your forked bioconda-recipes repository
git clone https://github.com/YOUR_REPOSITORY/bioconda-recipes.git

# move to the repository
```

(continues on next page)

(continued from previous page)

```
cd bioconda-recipes

# create and checkout new branch `sbpipe`. This branch is used for both sbpipe and␣
↪sbpiper.
git checkout -b sbpipe

# set a new remote upstream repository that will be synced with the fork.
git remote add upstream https://github.com/bioconda/bioconda-recipes.git

# synchronise the remote upstream repository with your local forked repository.
git fetch upstream
```

Create the recipes for SBpipe and sbpiper:

```
# assuming your current location is bioconda-recipes/, move to recipes
cd recipes

# use conda skeleton to create a recipe for sbpipe.
# This will create a folder called sbpipe.
conda skeleton pypi sbpipe

# use conda skeleton to create a recipe for sbpiper.
# This will create a folder called r-sbpiper
conda skeleton cran sbpiper

################
### At this stage, follow the instructions provided in the above three links. ###
################
```

Finally, the recipes should be committed and pushed. A pull request including these edits should be created in the repository *bioconda/bioconda-recipes*

```
git add -u
git commit -m 'added recipes'
git push origin sbpipe
```

### 7.4.2 How to test the conda package of SBpipe on the pdp10 channel (Anaconda Cloud)

This channel is used for storing the latest release of SBpipe and sbpiper. It is also used by Travis-CI for continuous integration tests.

```
# 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
```

To install the package from Anaconda Cloud:

```
conda install sbpipe -c pdp10 -c conda-forge -c fbergmann -c defaults
```

# 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

```
MIT License

Copyright (c) 2018 Piero Dalle Pezze

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

# 10 Change Log

```
sbpiper (MIT License)
Copyright 2010-2018 Piero Dalle Pezze
```

(continues on next page)

(continued from previous page)

```
### CHANGELOG

v4.21.0 (Beyond the Kuiper Belt)

- added support for Copasi Optimisation task. This also uses the -e option.
- bugfix: added is_package_installed.r to MANIFEST.ini.
- SBpipe v4.18.0, sbpiper v1.8.0, sbpipe_snake v1.0.0 and above are released under␣
↪MIT License.
Previous versions of these packages were released under GNU GPL v3.
- Improved project consistency and added warnings.
- Snakemake files moved to a separate repository (https://github.com/pdp10/sbpipe_
↪snake.git).
- Added script for moving data sets and update indexes.
- Project and documentation clean-up.
- SBpipe is now available on pypi.org.
- Improved setup.py file for python packaging
- SBpipe tests no longer require CopasiSE.
- 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.
- 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.
```

(continues on next page)

(continued from previous page)

```
- 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.
- 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.
```

(continues on next page)

(continued from previous page)

```
- 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).


v1.0.0 (Mercury)

- completed source code documentation
- completed user and developer manuals.
- configured Python Sphinx for documenting SBpipe
- bug fixes.
```

(continues on next page)

(continued from previous page)

```
- 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).
- 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.
```

(continues on next page)

(continued from previous page)

```
- 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.
```

# 11 Sphinx AutoAPI Index

This page is the top-level of your generated API documentation. Below is a list of all items that are documented here.

## 11.1 sbpipe_move_datasets

### 11.1.1 Module Contents

sbpipe_move_datasets.**get_index**(*name=""*, *output_path="."*)
Get the largest index of the reports in output_path :param name: the name of the report :param output_path: the output path storing reports :return: the largest index or -1 if no file was found

sbpipe_move_datasets.**move_dataset**(*name=""*, *input_path=""*, *output_path=""*)
Move data sets from one path to another and update the sequence number.

**Parameters**

- **name** – the model name without extension
- **input_path** – the path containing the input files
- **output_path** – the path to store the output files

sbpipe_move_datasets.**main**(*argv=None*)
Move data sets from one Path to another and update the sequence number.

## 11.2 sbpipe_cleanup

### 11.2.1 Module Contents

sbpipe_cleanup.**cleanup**()
Clean-up the package including the tests.

sbpipe_cleanup.**main**(*argv=None*)

## 11.3 __init__

### 11.3.1 Package Contents

**class** __init__.**NullHandler**

**emit**(*record*)

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

**Returns** sbpipe logo

__init__.**sbpipe_license**()
Return sbpipe license.

**Returns** sbpipe license

__init__.**sbpipe_version**()
Return sbpipe version.

**Returns** sbpipe version

__init__.**set_basic_logger**(*level="INFO"*)
Set a basic StreamHandler logger. :param level: the level for this console logger

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

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

**Parameters**

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

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

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

**Parameters**

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

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

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

**Returns** 0 if OK, 1 if trouble

## 11.4 utils

### 11.4.1 Submodules

#### utils.dependencies

#### Module Contents

utils.dependencies.**which**(*cmd_name*)
Utility equivalent to *which* in GNU/Linux OS.

**Parameters** **cmd_name** – a command name

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

utils.dependencies.**is_py_package_installed**(*package*)
Utility checking whether a Python package is installed.

**Parameters** **package** – a Python package name

**Returns** True if it is installed, false otherwise.

utils.dependencies.**is_r_package_installed**(*package*)
Utility checking whether a R package is installed.

**Parameters** **package** – an R package name

**Returns** True if it is installed, false otherwise.

#### utils.io

#### Module Contents

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

**Parameters**

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

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

**Parameters**

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

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

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

**Parameters**

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

**Returns** the files containing the pattern.

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

**Parameters**

- **fileout** – the output file
- **data** – the data to store in a file

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

**Parameters**

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

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

**Parameters** **report** – the report

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

**Parameters** **filename** – the file to remove

utils.io.**git_pull**(*repo_name*)
Pull a git repository.

**Parameters** **repo_name** – the repository to pull

utils.io.**git_clone**(*repo*)
Clone a git repository.

**Parameters** **repo** – the repository to clone

utils.io.**git_retrieve**(*repo*)
Clone or pull a git repository.

**Parameters** **repo** – the repository to clone or pull

## utils.parcomp

### Module Contents

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

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

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

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

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

**Parameters**

- **cmd** – the command string to run in parallel
- **cmd_iter_substr** – the substring of the iteration number. This will be replaced in a number automatically
- **output_dir** – the output directory

- **cluster** – the cluster type among local (Python multiprocessing), sge, or lsf
- **runs** – the number of runs. Ignored if colnames is not empty
- **local_cpus** – the number of cpus to use at most
- **output_msg** – print the output messages on screen (available for cluster='local' only)
- **colnames** – the name of the columns to process

**Returns** True if the computation succeeded.

utils.parcomp.**progress_bar**(*it*, *total*)
A minimal CLI progress bar.

**Parameters**

- **it** – current iteration starting from 1
- **total** – total iterations

utils.parcomp.**progress_bar2**(*it*, *total*)
A CLI progress bar.

**Parameters**

- **it** – current iteration starting from 1
- **total** – total iterations

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

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

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

**Parameters**

- **cmd** – the full command to run as a job
- **cmd_iter_substr** – the substring in command to be replaced with a number
- **runs** – the number of runs. Ignored if colnames is not empty
- **local_cpus** – The number of available cpus. If local_cpus <=0, only one core will be used.
- **output_msg** – print the output messages on screen (available for cluster_type='local' only)
- **colnames** – the name of the columns to process

**Returns** True

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

**Parameters**

- **cmd** – the full command to run as a job
- **cmd_iter_substr** – the substring in command to be replaced with a number
- **out_dir** – the directory containing the standard output from qsub
- **err_dir** – the directory containing the standard error from qsub
- **runs** – the number of runs. Ignored if colnames is not empty
- **colnames** – the name of the columns to process

**Returns** True if the computation succeeded.

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

**Parameters**

- **cmd** – the full command to run as a job
- **cmd_iter_substr** – the substring in command to be replaced with a number
- **out_dir** – the directory containing the standard output from bsub
- **err_dir** – the directory containing the standard error from bsub
- **runs** – the number of runs. Ignored if colnames is not empty
- **colnames** – the name of the columns to process

**Returns** True if the computation succeeded.

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

**Parameters**

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

**Returns** True

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

**Parameters**

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

**Returns** True

## utils.rand

### Module Contents

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

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

**Returns** the generated string

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

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

**Returns** the generated string

### utils.re_utils

#### Module Contents

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

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

**Returns** the key to sort strings alphanumerically

utils.re_utils.**escape_special_chars**(*text*)
Escape ^,%, ,[,],(,),{,} from text

**Parameters** **text** – the command to escape special characters inside

**Returns** the command with escaped special characters

## 11.5 pl

### 11.5.1 Subpackages

#### pl.create

#### Submodules

#### pl.create.newproj

#### Module Contents

**class** pl.create.newproj.**NewProj**(*models_folder="Models"*, *working_folder="Results"*)
This module initialises the folder tree for a new project.

**Parameters**

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

**__init__**(*models_folder="Models"*, *working_folder="Results"*)
Constructor.

**run**(*project_name*)
Create a project directory tree.

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

**Returns** 0

## pl.pe

### Submodules

## pl.pe.parest

### Module Contents

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

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

**__init__**(*models_folder="Models"*, *working_folder="Results"*, *sim_data_folder="param_estim_data"*, *sim_plots_folder="param_estim_plots"*)

**run**(*config_file*)

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

The first pipeline step: data generation.

**Parameters**

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

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

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

The second pipeline step: data analysis.

**Parameters**

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model name
- **inputdir** – the directory containing the simulation data
- **outputdir** – the directory to store the results
- **fileout_final_estims** – the name of the file containing final parameter sets with the objective value
- **fileout_all_estims** – the name of the file containing all the parameter sets with the objective value
- **fileout_param_estim_best_fits_details** – the file containing the statistics for the best fits analysis
- **fileout_param_estim_details** – the file containing the statistics for the PLE analysis

- **fileout_param_estim_summary** – the name of the file containing the summary for the parameter estimation
- **sim_plots_dir** – the directory of the simulation plots
- **best_fits_percent** – the percent to consider for the best fits
- **data_point_num** – the number of data points
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **plot_2d_66cl_corr** – True if 2 dim plots for the parameter sets within 66% should be plotted
- **plot_2d_95cl_corr** – True if 2 dim plots for the parameter sets within 95% should be plotted
- **plot_2d_99cl_corr** – True if 2 dim plots for the parameter sets within 99% should be plotted
- **logspace** – True if parameters should be plotted in log space
- **scientific_notation** – True if axis labels should be plotted in scientific notation

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

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

The third pipeline step: report generation.

**Parameters**

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

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

**parse**(*my_dict*)

## pl.ps1

### Submodules

## pl.ps1.parscan1

### Module Contents

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

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

**__init__**(*models_folder="Models"*, *working_folder="Results"*, *sim_data_folder="single_param_scan_data"*, *sim_plots_folder="single_param_scan_plots"*)

**run**(*config_file*)

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

The first pipeline step: data generation.

**Parameters**

- **simulator** – the name of the simulator (e.g. Copasi)

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

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

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

The second pipeline step: data analysis.

**Parameters**

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

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

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

The third pipeline step: report generation.

**Parameters**

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

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

**parse**(*my_dict*)

## pl.ps2

### Submodules

## pl.ps2.parscan2

### Module Contents

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

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

**__init__**(*models_folder="Models"*, *working_folder="Results"*, *sim_data_folder="double_param_scan_data"*, *sim_plots_folder="double_param_scan_plots"*)

**run**(*config_file*)

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

The first pipeline step: data generation.

**Parameters**

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

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

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

The second pipeline step: data analysis.

**Parameters**

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

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

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

The third pipeline step: report generation.

**Parameters**

- **model** – the model name

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

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

**parse**(*my_dict*)

### pl.sim

## Submodules

### pl.sim.sim

## Module Contents

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

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

**__init__**(*models_folder="Models"*, *working_folder="Results"*, *sim_data_folder="simulate_data"*, *sim_plots_folder="simulate_plots"*)

**run**(*config_file*)

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

The first pipeline step: data generation.

**Parameters**

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

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

**analyse_data**(*simulator*, *model*, *inputdir*, *outputdir*, *sim_plots_dir*, *exp_dataset*, *plot_exp_dataset*, *exp_dataset_alpha=1.0*, *cluster="local"*, *local_cpus=2*, *xaxis_label=""*, *yaxis_label=""*)

The second pipeline step: data analysis.

**Parameters**

- **simulator** – the name of the simulator (e.g. Copasi)
- **model** – the model name
- **inputdir** – the directory containing the data to analyse
- **outputdir** – the output directory containing the results
- **sim_plots_dir** – the directory to save the plots
- **exp_dataset** – the full path of the experimental data set
- **plot_exp_dataset** – True if the experimental data set should also be plotted

- **exp_dataset_alpha** – the alpha level for the data set
- **cluster** – local, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **local_cpus** – the number of CPUs.
- **xaxis_label** – the label for the x axis (e.g. Time [min])
- **yaxis_label** – the label for the y axis (e.g. Level [a.u.])

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

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

**Parameters**

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

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

**parse**(*my_dict*)

### 11.5.2 Submodules

**pl.pipeline**

### Module Contents

**class** pl.pipeline.**Pipeline**(*models_folder="Models"*, *working_folder="Results"*, *sim_data_folder="sim_data"*, *sim_plots_folder="sim_plots"*)
Generic pipeline.

**Parameters**

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

**__init__**(*models_folder="Models"*, *working_folder="Results"*, *sim_data_folder="sim_data"*, *sim_plots_folder="sim_plots"*)

**run**(*config_file*)
Run the pipeline.

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

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

**get_models_folder**()
Return the folder containing the models.

**Returns** the models folder.

**get_working_folder**()
Return the folder containing the results.

**Returns** the working folder.

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

**Returns** the folder of the simulated data sets.

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

**Returns** the folder of the simulated plots.

**generate_tarball**(*output_folder*)
Create a gz tarball.

**Parameters**

- **working_dir** – the working directory
- **output_folder** – the name of the folder to store the tar.gz file

**Returns** True if the generation of the tarball succeeded.

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

**Parameters** **simulator** – the simulator name

**Returns** the simulator object.

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

**Parameters** **config** – a YAML configuration file

**Returns** the dictionary structure of the configuration file

**Raise** yaml.YAMLError if the config cannot be loaded.

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

**Returns** a tuple containing the configuration

## 11.6 snakemake

### 11.6.1 Submodules

#### snakemake.data_generation

### Module Contents

snakemake.data_generation.**run_copasi_model**(*infile*)
Run a Copasi model

**Parameters** **infile** – the input file

snakemake.data_generation.**run_generic_model**(*infile*)
Run a generic model

**Parameters** **infile** – the input file

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

**Parameters**

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

## snakemake.model_checking

### Module Contents

snakemake.model_checking.**model_checking**(*infile*, *fileout*, *task_name*)

Check the consistency for the COPASI file.

**Parameters**

- **infile** – the input file
- **fileout** – the output file
- **task_name** – the name of the task (Copasi models)

**Returns** False if model checking can be executed and fails or if the COPASI simulator is not found.

## snakemake.pe_analysis

### Module Contents

snakemake.pe_analysis.**pe_combine_param_best_fits_stats**(*plots_dir*, *fileout_param_estim_best_fits_details*)

Combine the statistics for the parameter estimation details

**Parameters**

- **plots_dir** – the directory to save the generated plots
- **fileout_param_estim_best_fits_details** – the name of the file containing the detailed statistics for the estimated parameters

snakemake.pe_analysis.**pe_combine_param_ple_stats**(*plots_dir*, *fileout_param_estim_details*)

Combine the statistics for the parameter estimation details

**Parameters**

- **plots_dir** – the directory to save the generated plots
- **fileout_param_estim_details** – the name of the file containing the detailed statistics for the estimated parameters

snakemake.pe_analysis.**pe_ds_preproc**(*filename*, *param_names*, *logspace=True*, *all_fits=False*, *data_point_num=0*, *fileout_param_estim_summary="param_estim_summary.csv"*)

Parameter estimation pre-processing. It renames the data set columns, and applies a log10 transformation if logspace is TRUE. If all.fits is true, it also computes the confidence levels.

**Parameters**

- **filename** – the dataset filename containing the fits sequence
- **param_names** – the list of estimated parameter names
- **logspace** – true if the data set shoud be log10-transformed.
- **all_fits** – true if filename contains all fits, false otherwise
- **data_point_num** – the number of data points used for parameterise the model. Ignored if all.fits is false
- **fileout_param_estim_summary** – the name of the file containing the summary for the parameter estimation. Ignored if all.fits is false

snakemake.pe_analysis.**pe_objval_vs_iters_analysis**(*model_name*, *filename*, *plots_dir*)

Analysis of the Objective values vs Iterations.

**Parameters**

- **model_name** – the model name without extension
- **filename** – the filename containing the fits sequence
- **plots_dir** – the directory for storing the plots

snakemake.pe_analysis.**pe_parameter_density_analysis**(*model_name*, *filename*, *parameter*, *plots_dir*, *thres="BestFits"*, *best_fits_percent=100*, *fileout_param_estim_summary=""*, *logspace=True*, *scientific_notation=True*)

Parameter density analysis.

**Parameters**

- **model_name** – the model name without extension
- **filename** – the filename containing the fits sequence
- **parameter** – the name of the parameter to plot the density
- **plots_dir** – the directory for storing the plots
- **thres** – the threshold used to filter the dataset. Values: “BestFits”, “CL66”, “CL95”, “CL99”, “All”.
- **best_fits_percent** – the percent of best fits to analyse. Only used if thres=”BestFits”.
- **fileout_param_estim_summary** – the name of the file containing the summary for the parameter estimation. Only used if thres!=”BestFits”.
- **logspace** – true if the parameters should be plotted in logspace
- **scientific_notation** – true if the axis labels should be plotted in scientific notation

snakemake.pe_analysis.**pe_parameter_pca_analysis**(*model_name*, *filename*, *plots_dir*, *best_fits_percent=100*)

PCA for the best fits of the estimated parameters.

**Parameters**

- **model_name** – the model name without extension
- **filename** – the filename containing the fits sequence
- **plots_dir** – the directory for storing the plots
- **best_fits_percent** – the percent of best fits to analyse. Only used if thres=”BestFits”.

snakemake.pe_analysis.**pe_sampled_2d_ple_analysis**(*model_name*, *filename*, *parameter1*, *parameter2*, *plots_dir*, *thres="BestFits"*, *best_fits_percent=100*, *fileout_param_estim_summary=""*, *logspace=True*, *scientific_notation=True*)

2D profile likelihood estimation analysis.

**Parameters**

- **model_name** – the model name without extension
- **filename** – the filename containing the fits sequence
- **parameter1** – the name of the first parameter
- **parameter2** – the name of the second parameter
- **plots_dir** – the directory for storing the plots
- **thres** – the threshold used to filter the dataset. Values: “BestFits”, “CL66”, “CL95”, “CL99”, “All”.
- **best_fits_percent** – the percent of best fits to analyse. Only used if thres=”BestFits”.
- **fileout_param_estim_summary** – the name of the file containing the summary for the parameter estimation. Only used if thres!=”BestFits”.
- **logspace** – true if the parameters should be plotted in logspace
- **scientific_notation** – true if the axis labels should be plotted in scientific notation

snakemake.pe_analysis.**pe_sampled_ple_analysis**(*model_name*, *filename*, *parameter*, *plots_dir*, *fileout_param_estim_summary*, *logspace=True*, *scientific_notation=True*)

Run the profile likelihood estimation analysis.

**Parameters**

- **model_name** – the model name without extension
- **filename** – the filename containing the fits sequence
- **parameter** – the parameter to compute the PLE analysis
- **plots_dir** – the directory to save the generated plots
- **fileout_param_estim_summary** – the name of the file containing the summary for the parameter estimation
- **logspace** – true if parameters should be plotted in logspace
- **scientific_notation** – true if the axis labels should be plotted in scientific notation

## snakemake.pe_collection

### Module Contents

snakemake.pe_collection.**pe_collect**(*inputdir*, *outputdir*, *fileout_final_estims*, *fileout_all_estims*, *copasi=True*)

Collect the results so that they can be processed.

**Parameters**

- **inputdir** – the input folder containing the data
- **outputdir** – the output folder to stored the collected results
- **fileout_final_estims** – the name of the file containing the best estimations
- **fileout_all_estims** – the name of the file containing all the estimations
- **copasi** – True if COPASI was used to generate the data.

## snakemake.pe_postproc

### Module Contents

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

**Parameters**

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

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

**Parameters**

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

## snakemake.preproc

### Module Contents

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

**Parameters**

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

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

**Parameters**

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

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

**Parameters**

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

## snakemake.ps1_analysis

### Module Contents

snakemake.ps1_analysis.**ps1_analyse_plot**(*model_name*, *inhibition_only*, *inputdir*, *outputdir*, *repeat*, *percent_levels*, *min_level*, *max_level*, *levels_number*, *xaxis_label*, *yaxis_label*)

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

**Parameters**

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

snakemake.ps1_analysis.**ps1_analyse_plot_homogen**(*model_name*, *inputdir*, *outputdir*, *repeat*, *xaxis_label*, *yaxis_label*)

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

**Parameters**

- **model_name** – the model name without extension
- **inputdir** – the input directory containing the simulated data
- **outputdir** – the output directory that will contain the simulated plots
- **repeat** – the simulation number
- **xaxis_label** – the label for the x axis (e.g. Time [min])
- **yaxis_label** – the label for the y axis (e.g. Level [a.u.])

## snakemake.ps1_postproc

### Module Contents

snakemake.ps1_postproc.**ps1_header_init**(*report*, *scanned_par*)

Header report initialisation for single parameter scan pipeline.

**Parameters**

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

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

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

Perform post processing organisation to single parameter scan report files.

**Parameters**

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

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

Perform post processing organisation to single parameter scan report files.

**Parameters**

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

## snakemake.ps2_analysis

### Module Contents

snakemake.ps2_analysis.**ps2_analyse_plot**(*model*, *scanned_par1*, *scanned_par2*, *inputdir*, *outputdir*, *id*)

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

**Parameters**

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

## snakemake.ps2_postproc

### Module Contents

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

Perform post processing organisation to double parameter scan report files.

**Parameters**

- **infile** – the model to process

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

snakemake.ps2_postproc.**ps2_postproc**(*infile*, *outfile*, *sim_length*, *copasi=True*)
Perform post processing organisation to double parameter scan report files.

**Parameters**

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

## snakemake.sim_analysis

### Module Contents

snakemake.sim_analysis.**sim_analyse_gen_stats_table**(*inputfile*, *outputfile*, *variable*)
Plot model simulation time courses (Python wrapper).

**Parameters**

- **inputfile** – the file containing the repeats
- **outputfile** – the output file containing the statistics
- **variable** – the model variable to analyse

snakemake.sim_analysis.**sim_analyse_summarise_data**(*inputdir*, *model*, *outputfile_repeats*, *variable*)
Plot model simulation time courses (Python wrapper).

**Parameters**

- **inputdir** – the directory containing the data to analyse
- **model** – the model name
- **outputfile_repeats** – the output file containing the model simulation repeats
- **variable** – the model variable to analyse

snakemake.sim_analysis.**sim_analyse_plot_sep_sims**(*inputdir*, *outputdir*, *model*, *exp_dataset*, *plot_exp_dataset*, *exp_dataset_alpha*, *xaxis_label*, *yaxis_label*, *variable*)
Plot model simulation time courses (Python wrapper).

**Parameters**

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

- **variable** – the model variable to analyse

snakemake.sim_analysis.**sim_analyse_plot_comb_sims**(*inputdir*, *outputdir*, *model*, *exp_dataset*, *plot_exp_dataset*, *exp_dataset_alpha*, *xaxis_label*, *yaxis_label*, *variable*)

Plot model simulation time courses (Python wrapper).

**Parameters**

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

### snakemake.sim_postproc

#### Module Contents

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

Perform post processing file editing for the *simulate* pipeline

**Parameters**

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

snakemake.sim_postproc.**sim_postproc**(*infile*, *outfile*, *copasi=True*)

Perform post processing file editing for the *simulate* pipeline

**Parameters**

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

## 11.7 report

### 11.7.1 Submodules

### report.latex_reports

#### Module Contents

report.latex_reports.**get_latex_header**(*pdftitle="SBpipe report"*, *title="SBpipe report"*, *abstract="Generic report."*)

Initialize a Latex header with a title and an abstract.

**Parameters**

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

**Returns** the LaTeX header

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

Generate a report for a single parameter scan task.

**Parameters**

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

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

Generate a report for a double parameter scan task.

**Parameters**

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

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

Generate a report for a time course task.

**Parameters**

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

report.latex_reports.**latex_report_pe**(*outputdir*, *plots_folder*, *model_noext*, *filename_prefix*)

Generate a report for a parameter estimation task.

**Parameters**

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

report.latex_reports.**latex_report**(*outputdir*, *plots_folder*, *model_noext*, *filename_prefix*, *caption=False*)

Generate a generic report.

**Parameters**

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

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

Generate a PDF report from LaTeX report using pdflatex.

**Parameters**

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

## 11.8 simul

### 11.8.1 Subpackages

**simul.copasi**

### Submodules

**simul.copasi.copasi**

### Module Contents

**class** simul.copasi.copasi.**Copasi**

Copasi simulator.

**__init__**()

**model_checking**(*model_filename*, *fileout*, *task_name=""*)

Check whether the Copasi model can be loaded and executed correctly.

**Parameters**

- **model_filename** – the COPASI filename
- **fileout** – the file containing the model checking results
- **task_name** – the task to check

**Returns** boolean indicating whether the model could be loaded and executed successfully

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

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

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

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

**_run_par_comput**(*inputdir*, *model*, *outputdir*, *cluster="local"*, *local_cpus=1*, *runs=1*, *output_msg=False*)

**_get_params_list**(*filein*)

Return the list of parameter names from filein

**Parameters** **filein** – a Copasi parameter estimation report file

**Returns** the list of parameter names

**_write_best_fits** (*files*, *path_out*, *filename_out*)
Write the final estimates to filename_out

**Parameters**

- **files** – the list of Copasi parameter estimation reports
- **path_out** – the path to store the file combining the final (best) estimates (filename_out)
- **filename_out** – the file containing the final (best) estimates

**_write_all_fits** (*files*, *path_out*, *filename_out*)
Write all the estimates to filename_out

**Parameters**

- **files** – the list of Copasi parameter estimation reports
- **path_out** – the path to store the file combining all the estimates
- **filename_out** – the file containing all the estimates

## simul.copasi.model_checking

### Module Contents

simul.copasi.model_checking.**copasi_model_checking** (*model_filename*, *fileout*, *task_name=""*)
Perform a basic model checking for a COPASI model file.

**Parameters**

- **model_filename** – the filename to a COPASI file
- **fileout** – the file containing the model checking results
- **task_name** – the task to check

**Returns** a boolean indicating whether the model could be loaded successfully

simul.copasi.model_checking.**severity2string** (*severity*)
Return a string representing the severity of the error message :param severity: an integer representing severity :return: a string of the error message

simul.copasi.model_checking.**check_model_loading** (*model_filename*, *data_model*)
Check whether the COPASI model can be loaded

**Parameters**

- **model_filename** – the filename to a COPASI file
- **data_model** – the COPASI data model structure

**Returns** a boolean indicating whether the model could be loaded successfully

simul.copasi.model_checking.**check_task_selection** (*model_filename*, *task_name*, *data_model*)
Check whether the COPASI model task can be executed

**Parameters**

- **model_filename** – the filename to a COPASI file
- **task_name** – the task to check
- **data_model** – the COPASI data model structure.

**Returns** a boolean indicating whether the model task can be executed correctly

simul.copasi.model_checking.**check_task_report**(*model_filename*, *task_name*, *data_model*, *task*)

Check whether the COPASI model task can be executed

**Parameters**

- **model_filename** – the filename to a COPASI file
- **task_name** – the task to check
- **data_model** – the COPASI data model structure
- **task** – the COPASI task data structure

**Returns** a boolean indicating whether the model task can be executed correctly

**simul.python**

### Submodules

**simul.python.python**

### Module Contents

**class** simul.python.python.**Python**

Python Simulator.

**__init__**()

### 11.8.2 Submodules

**simul.pl_simul**

### Module Contents

**class** simul.pl_simul.**PLSimul**(*lang=None*, *lang_err_msg="No programming language is set!"*, *options=""*)

A generic simulator for models coded in a programming language.

**__init__**(*lang=None*, *lang_err_msg="No programming language is set!"*, *options=""*)

A constructor for a simulator of models coded in a programming language :param lang: the programming language name (e.g. python, Copasi) :param lang_err_msg: the message to print if lang is not found. :param options: the options to use when invoking the command (e.g. “” for python).

**get_lang**()

Return the programming language name :return: the name

**get_lang_err_msg**()

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

**get_lang_options**()

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

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

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

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

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

**_run_par_comput**(*model*, *inputdir*, *outputdir*, *cluster="local"*, *local_cpus=1*, *runs=1*, *output_msg=False*)

**replace_str_in_report**(*report*)

**simul.simul**

## Module Contents

**class** simul.simul.**Simul**

Generic simulator.

**__init__**()

Default constructor.

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

Time course simulator.

**Parameters**

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

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

Single parameter scan.

**Parameters**

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

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

Double paramter scan.

**Parameters**

- **model** – the model to process
- **sim_length** – the length of the simulation

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

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

**Parameters**

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

**get_sim_columns** (*path_in="."*)
Return the columns to analyse (sim task)

**Parameters** **path_in** – the path to the input files

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

**Parameters**

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

**Returns** the number of retrieved files

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

**Parameters**

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

**Returns** the number of retrieved files

**_run_par_comput** (*model*, *inputdir*, *outputdir*, *cluster="local"*, *local_cpus=1*, *runs=1*, *output_msg=False*)
Run generic parallel computation.

**Parameters**

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

**Returns** (groupid, group_model)

**_get_model_group** (*model*)
Return the model without extension concatenated with the groupid string :param model: the model name :return: the concatenated string

**_move_reports** (*inputdir*, *outputdir*, *model*, *groupid*)
Move the report files

**Parameters**

- **inputdir** – the directory containing the model
- **outputdir** – the directory containing the output files
- **model** – the model to process
- **groupid** – a string identifier in the file name characterising the batch of simulated models.

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

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

**_get_input_files** (*path*)
Retrieve the input files in a path.

**Parameters** **path** – the path containing the input files to retrieve

**Returns** the list of input files

**_get_params_list** (*filein*)
Return the list of parameter names from filein

**Parameters** **filein** – a report file

**Returns** the list of parameter names

**_write_params** (*col_names*, *path_out*, *filename_out*)
Write the list of parameter names to filename_out

**Parameters**

- **col_names** – the list of parameter names
- **path_out** – the path to store filename_out
- **filename_out** – the output file to store the parameter names

**_write_best_fits** (*files*, *path_out*, *filename_out*)
Write the final estimates to filename_out

**Parameters**

- **files** – the list of parameter estimation reports
- **path_out** – the path to store the file combining the final (best) estimates (filename_out)

- **filename_out** – the file containing the final (best) estimates

**_write_all_fits**(*files*, *path_out*, *filename_out*)
Write all the estimates to filename_out

**Parameters**

- **files** – the list of parameter estimation reports
- **path_out** – the path to store the file combining all the estimates
- **filename_out** – the file containing all the estimates

**_ps1_header_init**(*report*, *scanned_par*)
Header report initialisation for single parameter scan pipeline.

**Parameters**

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

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

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

**Parameters**

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

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

**Parameters**

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

# Python Module Index

# Index

## Symbols



## A



## C



## E



## F



## G



## I

## L



## M



## N



## P



## Q



## R

## S



## U



## W