SB pipe documentation

Release 1.5.0

Piero Dalle Pezze and Nicolas Le Novère

October 12, 2016

CHAPTER

ONE

USER MANUAL

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

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

Mailing list: sb_pipe AT googlegroups.com

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

Introduction

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

Requirements

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

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

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

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

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

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

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

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

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

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

To install SB pipe Python dependencies, run:

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

To install SB pipe R dependencies, run:

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

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

Installation

Run the command inside the sb_pipe folder:

```
python setup.py install
```

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

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

How to use SB pipe

Preliminary configuration steps

Pipelines using Copasi

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

pipeline: simulate

- Tick the flag *executable* in the Time Course Task.
- Select a report template for the Time Course Task.
- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv.

pipeline: single or double parameter scan

- Tick the flag *executable* in the Parameter Scan Task.
- Select a report template for the Parameter Scan Task.
- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv.

pipeline: param-estim

- Tick the flag *executable* in the Parameter Estimation Task.

- Select the report template for the Parameter Estimation Task.
- Save the report in the same folder with the same name as the model but replacing the extension .cps with .csv.

Running SB pipe

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

As of Sep 2016 the output is as follows:

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

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

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

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

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

This generates the following structure:

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

Models must be stored in the Models/ folder. The folder Data/ is meant for collecting experimental data files and analyses in one place. Once the data files for Copasi (e.g. for parameter estimation) are generated, **it is advised** to move them into the Models/ folder so that the Copasi (.cps) file and its associated experimental data files are stored in the same folder. To run SB pipe, users need to create a configuration file for each pipeline they intend to run (see next section). These configuration files should be placed in the Working_Folder/. This folder will eventually contain all the results generated by SB pipe.

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

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

Pipeline configuration files

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

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

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

Example 1: configuration file for the pipeline *simulate*

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

Example 2: configuration file for the pipeline *single_param_scan*

```
[single_param_scan]
generate_data=True
analyse_data=True
generate_report=True
project_dir=..
model=insulin_receptor_inhib_scan_IR_beta.cps
# The variable to scan (as set in Copasi Parameter Scan Task)
scanned_par=IR_beta
# The number of intervals in the simulation
simulate__intervals=100
# The number of simulations to perform for each scan
single_param_scan_simulations_number=1
# True if the variable is only reduced (knock down), False otherwise.
single_param_scan_knock_down_only=True
# True if the scanning represents percent levels.
single_param_scan_percent_levels=True
# The minimum level (as set in Copasi Parameter Scan Task)
min_level=0
# The maximum level (as set in Copasi Parameter Scan Task)
max_level=100
# The number of scans (as set in Copasi Parameter Scan Task)
levels_number=10
# True if plot lines are the same between scans (e.g. full lines, same colour)
```

```
homogeneous_lines=False
# 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_param_scan*

```
[double_param_scan]
generate_data=True
analyse_data=True
generate_report=True
project_dir=..
model=insulin_receptor_inhib_dbl_scan_InsulinPercent__IRbetaPercent.cps
# The 1st variable to scan (as set in Copasi Parameter Scan Task)
scanned_par1=InsulinPercent
# The 2nd variable to scan (as set in Copasi Parameter Scan Task)
scanned_par2=IRbetaPercent
# The length of the simulation (as set in Copasi Time Course Task)
sim_length=10
```

Example 4: configuration file for the pipeline *param_estim*

```
[param_estim]
generate_data=True
analyse_data=True
generate_report=True
generate_tarball=True
project_dir=..
model=insulin_receptor_param_estim.cps
cluster=pp
pp_cpus=7
# The parameter estimation round which is used to distinguish
# phases of parameter estimations when parameters cannot be
# estimated at the same time
round=1
# The number of parameter estimations
# (the length of the fit sequence)
runs=40
# The threshold percentage of the best fits to consider
best_fits_percent=75
# The number of available data points
data_point_num=33
# True if 2D all fits plots for 66% and 95% confidence levels
# should be plotted. This is computationally expensive.
plot_2d_66_95cl_corr=True
# True if parameter values should 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:

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

Reporting bugs or requesting new features

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

```
sb_pipe AT googlegroups.com
```

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

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

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

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

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

CHAPTER

TWO

DEVELOPER MANUAL

Mailing list: sb_pipe AT googlegroups.com

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

Introduction

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

Development model

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

Conventions

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

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

Work flow

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

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

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

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

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

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

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

Finally delete the branch:

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

New releases

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

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

To transfer the tag to the remote server:

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

To see all the releases:

```
git show
```

Package structure

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

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

The project is structured as follows:

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

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

docs

The folder *docs/* contains the documentation for this project. The user and developer manuals are contained inside the subfolder *source*. In order to generate the complete documentation for SB pipe, the following packages must

be installed:

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

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

To generate the source code documentation:

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

To clean the documentation:

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

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

sb_pipe

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

pipelines

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

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

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

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

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

utils

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

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

tests

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

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

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

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

Miscellaneous of useful commands

Git

Startup

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

Update

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

File system

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

Information

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

Maintenance

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

Rename a branch locally and remotely

```
git branch -m old_branch new_branch         # Rename branch locally
git push origin :old_branch                 # Delete the old branch
git push --set-upstream origin new_branch   # Push the new branch, set
local branch to track the new remote
```

Reset

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

Syncing a fork (assuming upstreams are set)

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

CHAPTER

THREE

SOURCE CODE

Python modules

basic_sync_counter module

class basic_sync_counter.**BasicSyncCounter**

This is a monitor. It is a callback class for collecting information about finished processes. It is used by Parallel Python (pp).

add(*pid*, *value*)

The callback function

Parameters

- **pid** – this is callbackargs passed to parallel python *submit()* method
- **value** – the return value of the parallelised function. It is the callback value.

get_count()

Return the counter

Returns the number of running processes.

get_value()

Return the internal status.

Returns True if the counter is empty.

collect_results module

collect_results.**get_parameter_names_list**(*filein*)

Return the list of parameter names from filein

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

Returns the list of parameter names

collect_results.**retrieve_all_estimates**(*path_in='.'*, *path_out='.'*, *filename_out='all_estimates.csv'*)

Collect all the parameter estimates from the Copasi parameter estimation report. 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** – the filename to store the final estimates

collect_results.**retrieve_final_estimates**(*path_in='.'*, *path_out='.'*, *filename_out='final_estimates.csv'*)
Collect the final parameter estimates from the Copasi parameter estimation report. 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** – the filename to store the final estimates

collect_results.**retrieve_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

collect_results.**write_all_estimates**(*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

collect_results.**write_final_estimates**(*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

collect_results.**write_parameter_names**(*colNames*, *path_out*, *filename_out*)
Write the list of parameter names to filename_out

Parameters

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

copasi_parser module

class copasi_parser.**CopasiParser**
Retrieve information from a Copasi file.

retrieve_param_estim_values(*file_in*)
Parse a Copasi file and retrieve information on the parameters to estimate.

Parameters **file_in** – the Copasi file including absolute path to parse

Returns a tuple containing the report file name, the parameter lower bounds, names, starting values, and upper bounds

copasi_utils module

copasi_utils.**replace_str_copasi_sim_report**(*report*)
Replace a group of annotation strings from a generated copasi report file

Parameters **report** – The report file with absolute path

create_project module

class create_project.**CreateProject**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*)
Bases: *pipeline.Pipeline* (page 20)

This module initialises the folder tree for a new project.

Parameters

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

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

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

Returns 0

double_param_scan module

class double_param_scan.**DoubleParamScan**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*, *sim_data_folder='double_param_scan_data'*, *sim_plots_folder='double_param_scan_plots'*)
Bases: *pipeline.Pipeline* (page 20)

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

static analyse_data(*model*, *scanned_par1*, *scanned_par2*, *inputdir*, *outputdir*)
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

static generate_data(*model*, *sim_length*, *inputdir*, *outputdir*)
The first pipeline step: data generation.

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

static 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.

read_configuration (*lines*)

run (*config_file*)

io_util_functions module

io_util_functions.**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.

io_util_functions.**get_pattern_position** (*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

io_util_functions.**refresh_directory** (*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

io_util_functions.**replace_string_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

io_util_functions.**write_matrix_on_file** (*path*, *filename_out*, *data*)
Write the matrix results stored in data to filename_out

Parameters

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

latex_reports module

latex_reports.**get_latex_header**(*pdftitle='SB pipe report'*, *title='SB pipe 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

latex_reports.**latex_report**(*outputdir*, *sim_plots_folder*, *model_noext*, *filename_prefix*)

Generate a generic report.

Parameters

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

latex_reports.**latex_report_double_param_scan**(*outputdir*, *sim_plots_folder*, *filename_prefix*, *model_noext*, *scanned_par1*, *scanned_par2*)

Generate a report for a double parameter scan task.

Parameters

- **outputdir** – the output directory
- **sim_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_par1** – the 2nd scanned parameter

latex_reports.**latex_report_simulate**(*outputdir*, *sim_plots_folder*, *model_noext*, *filename_prefix*)

Generate a report for a time course task.

Parameters

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

latex_reports.**latex_report_single_param_scan**(*outputdir*, *sim_plots_folder*, *filename_prefix*, *model_noext*, *scanned_par*)

Generate a report for a single parameter scan task.

Parameters

- **outputdir** – the output directory
- **sim_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

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

parallel_computation module

parallel_computation.**parallel_computation**(*command*, *command_iter_substr*, *cluster_type*, *runs*, *output_dir*, *pp_cpus=1*)
Generic funcion to run a command in parallel

Parameters

- **command** – the command string to run in parallel
- **command_iter_substr** – the substring of the iteration number. This will be replaced in a number automatically
- **cluster_type** – the cluster type among pp (multithreading), sge, or lsf
- **runs** – the number of runs
- **output_dir** – the output directory
- **pp_cpus** – the number of cpus that pp should use at most

parallel_computation.**run_command_instance**(*command*)
Run a command using Python subprocess.

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

parallel_computation.**run_command_pp**(*command*, *command_iter_substr*, *runs*, *server*, *syncCounter=<basic_sync_counter.BasicSyncCounter instance>*)
Run instances of a command in multithreading using parallel python (pp).

Parameters

- **command** – the command string to run in parallel
- **command_iter_substr** – the substring of the iteration number. This will be replaced in a number automatically
- **runs** – the number of runs
- **server** – the server that pp should use
- **syncCounter** – the mutex object to count the jobs

parallel_computation.**run_jobs_lsf**(*command*, *command_iter_substr*, *outDir*, *errDir*, *runs*)
Run jobs using a Load Sharing Facility (LSF) cluster.

Parameters

- **command** – the full command to run as a job
- **command_iter_substr** – the substring in command to be replaced with a number
- **outDir** – the directory containing the standard output from bsub
- **errDir** – the directory containing the standard error from bsub
- **runs** – the number of runs to execute

parallel_computation.**run_jobs_pp**(*command*, *command_iter_substr*, *runs*, *pp_cpus=1*)
Run jobs using parallel python (pp) locally.

Parameters

- **command** – the full command to run as a job
- **command_iter_substr** – the substring in command to be replaced with a number
- **runs** – the number of runs to execute
- **pp_cpus** – The number of available cpus. If pp_cpus <=0, all the available cores will be used.

parallel_computation.**run_jobs_sge**(*command*, *command_iter_substr*, *outDir*, *errDir*, *runs*)
Run jobs using a Sun Grid Engine (SGE) cluster.

Parameters

- **command** – the full command to run as a job
- **command_iter_substr** – the substring in command to be replaced with a number
- **outDir** – the directory containing the standard output from qsub
- **errDir** – the directory containing the standard error from qsub
- **runs** – the number of runs to execute

param_estim module

class param_estim.**ParamEstim**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*, *sim_data_folder='param_estim_data'*, *sim_plots_folder='param_estim_plots'*)
Bases: *pipeline.Pipeline* (page 20)

This module provides the user with a complete pipeline of scripts for running a model parameter estimation using copasi

static analyse_data(*model*, *inputdir*, *outputdir*, *fileout_final_estims*, *fileout_all_estims*, *fileout_approx_ple_stats*, *fileout_conf_levels*, *sim_plots_dir*, *best_fits_percent*, *data_point_num*, *plot_2d_66_95cl_corr=False*, *logspace=True*, *scientific_notation=True*)
The second pipeline step: data analysis.

Parameters

- **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 Chi^2
- **fileout_all_estims** – the name of the file containing all the parameter sets with Chi^2
- **fileout_approx_ple_stats** – the file name of the PLE results
- **fileout_conf_levels** – the file name of the confidence levels results
- **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

- **plot_2d_66_95cl_corr** – True if 2 dim plots for the parameter sets within 66% and 95% 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

static generate_data(*model*, *inputdir*, *cluster_type*, *pp_cpus*, *nfits*, *outputdir*, *sim_data_dir*, *updated_models_dir*)
The first pipeline step: data generation.

Parameters

- **model** – the model to process
- **inputdir** – the directory containing the model
- **cluster_type** – pp for parallel python, lsf for load sharing facility, sge for sun grid engine
- **pp_cpus** – the number of cpu for parallel python
- **nfits** – the number of fits to perform
- **outputdir** – the directory to store the results
- **sim_data_dir** – the directory containing the simulation data sets
- **updated_models_dir** – the directory containing the Copasi models with updated parameters for each estimation

static 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

read_configuration(*lines*)

run(*config_file*)

pipeline module

class pipeline.**Pipeline**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*, *sim_data_folder='sim_data'*, *sim_plots_folder='sim_plots'*)
Generic pipeline.

Parameters

- **data_folder** – the folder containing the experimental (wet) data sets
- **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

config_parser(*config_file*, *section*)
Return the configuration for the parsed section in the config_file

Parameters

- **config_file** – the configuration file to parse

- **section** – the section in the configuration file to parse

Returns the configuration for the parsed section in the config_file

get_data_folder()
Return the folder containing the experimental (wet) data sets.

Returns the experimental data sets folder.

get_models_folder()
Return the folder containing the models.

Returns the models folder.

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

Returns the folder of the simulated data sets.

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

Returns the folder of the simulated plots.

get_working_folder()
Return the folder containing the results.

Returns the working folder.

read_common_configuration(*lines*)
Parse the common parameters from the configuration file

Returns return a tuple containing the common parameters

read_configuration(*lines*)
Read the section lines from the configuration file. This method is abstract.

Returns a tuple containing the configuration

run(*config_file*)
Run the pipeline.

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

Returns 0 if the pipeline was executed correctly, 1 if the pipeline executed but some output was skipped, 2 if the pipeline did not execute correctly.

random_functions module

random_functions.**get_rand_alphanum_str**(*length*)
Return a random alphanumeric string

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

Returns the generated string

random_functions.**get_rand_num_str**(*length*)
Return a random numeric string

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

Returns the generated string

randomise_parameters module

class randomise_parameters.**RandomiseParameters**(*path*, *filename_in*)
This class generates multiple copies of a Copasi file configured for parameter estimation task, and randomises the starting values of the parameters to estimate.

Parameters

- **path** – the path to filename_in
- **filename_in** – the Copasi file to process.

generate_instances_from_template(*num_files*, *idstr*)
Generate num_files files and randomise the starting values for the parameter to estimate.

Parameters

- **num_files** – the number of files (instances) to generate
- **idstr** – an ID string to label the generated files (e.g. a timestamp)

get_copasi_obj()
Return the Copasi parser object

Returns the Copasi parser object

get_lower_bounds_list()
Return the list of parameter lower bounds

Returns the list of parameter lower bounds

get_param_names_list()
Return the list of parameter names

Returns the list of parameter names

get_path()
Return the path containing the template Copasi file

Returns the path to the Copasi file

get_report_filename_template_str()
Return the name of the template parameter estimation report

Returns the name of the report file name for parameter estimation

get_start_values_list()
Return the list of parameter starting values

Returns the list of parameter starting values

get_template_copasi_file()
Return the name of the template Copasi file

Returns the name of the Copasi file

get_upper_bounds_list()
Return the list of parameter upper bounds

Returns the list of parameter upper bounds

print_parameters_to_estimate()
Print the parameter names, lower/upper bounds, and starting value, as extracted from COPASI template file

re_utils module

re_utils.**natural_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

sensitivity module

class sensitivity.**Sensitivity**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*, *sim_data_folder='sensitivity_data'*, *sim_plots_folder='sensitivity_plots'*)

Bases: *pipeline.Pipeline* (page 20)

This module provides the user with a complete pipeline of scripts for computing model sensitivity analysis using Copasi

static analyse_data(*outputdir*)

The second pipeline step: data analysis.

Parameters **outputdir** – the directory to store the performed analysis

static generate_data(*model*, *inputdir*, *outputdir*)

The first pipeline step: data generation.

Parameters

- **model** – the model to process
- **inputdir** – the directory containing the model
- **outputdir** – the directory to store the results

static 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 directory containing the time courses results combined with experimental data

read_configuration(*lines*)

run(*config_file*)

simulate module

class simulate.**Simulate**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*, *sim_data_folder='simulate_data'*, *sim_plots_folder='simulate_plots'*)

Bases: *pipeline.Pipeline* (page 20)

This module provides the user with a complete pipeline of scripts for running a model simulation using copasi

static analyse_data(*model*, *inputdir*, *outputdir*, *sim_plots_dir*, *xaxis_label*, *yaxis_label*)

The second pipeline step: data analysis.

Parameters

- **model** – the model name
- **inputdir** – the directory containing the data to analyse
- **outputdir** – the output directory containing the results
- **sim_plots_dir** – the directory to save the plots
- **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.])

static generate_data(*model*, *inputdir*, *outputdir*, *cluster_type='pp'*, *pp_cpus=2*, *runs=1*)
The first pipeline step: data generation.

Parameters

- **model** – the model to process
- **inputdir** – the directory containing the model
- **outputdir** – the directory containing the output files
- **cluster_type** – pp for local Parallel Python, lsf for Load Sharing Facility, sge for Sun Grid Engine.
- **pp_cpus** – the number of CPU used by Parallel Python.
- **runs** – the number of model simulation

static 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

read_configuration(*lines*)

run(*config_file*)

single_param_scan module

class single_param_scan.**SingleParamScan**(*data_folder='Data'*, *models_folder='Models'*, *working_folder='Working_Folder'*, *sim_data_folder='single_param_scan_data'*, *sim_plots_folder='single_param_scan_plots'*)
Bases: *pipeline.Pipeline* (page 20)

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

static analyse_data(*model*, *scanned_par*, *knock_down_only*, *outputdir*, *sim_data_folder*, *sim_plots_folder*, *simulations_number*, *percent_levels*, *min_level*, *max_level*, *levels_number*, *homogeneous_lines*, *xaxis_label*, *yaxis_label*)
The second pipeline step: data analysis.

Parameters

- **model** – the model name
- **scanned_par** – the scanned parameter
- **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
- **simulations_number** – the number of simulations
- **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
- **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.])

static generate_data(*model*, *scanned_par*, *sim_number*, *simulate_intervals*, *single_param_scan_intervals*, *inputdir*, *outputdir*)

The first pipeline step: data generation.

Parameters

- **model** – the model to process
- **scanned_par** – the scanned parameter
- **sim_number** – the number of simulations (for det sim: 1, for stoch sim: n>1)
- **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

static 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

read_configuration(*lines*)

run(*config_file*)

CHAPTER

FOUR

META INFORMATION

Copyright

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

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

CHAPTER

FIVE

INDICES

- genindex
- modindex
- search

b

c

d

i

l

p

r

s

A

B

C

D

F

G