PEtab

Release latest

Dec 14, 2019

Data format

CHAPTER 1

PEtab data format specification

This document explains the PEtab data format.

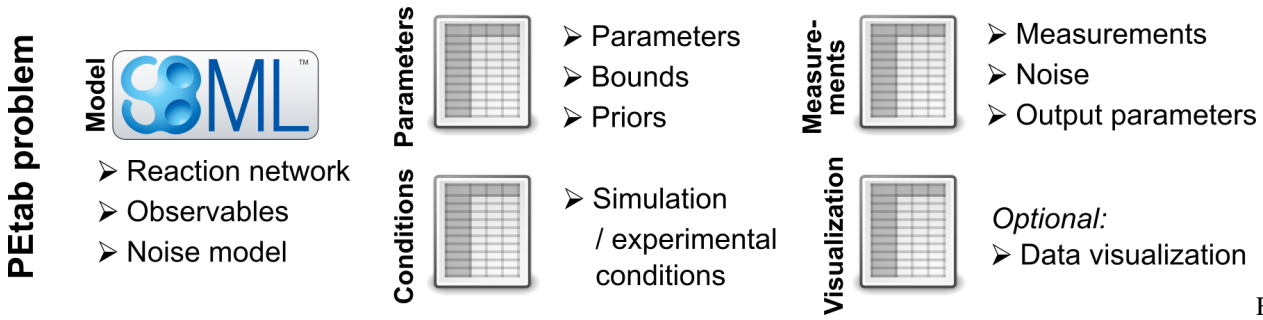
1.1 Purpose

Providing a standardized way for specifying parameter estimation problems in systems biology, especially for the case of Ordinary Differential Equation (ODE) models.

1.2 Overview

The PEtab data format specifies a parameter estimation problem using a number of text-based files (Systems Biology Markup Language (SBML) and Tab-Separated Values (TSV)), i.e.

- An SBML model [SBML]
- A measurement file to fit the model to [TSV]
- A condition file specifying model inputs and condition-specific parameters [TSV]
- A parameter file specifying optimization parameters and related information [TSV]
- (optional) A simulation file, which has the same format as the measurement file, but contains model simulations [TSV]
- (optional) A visualization file, which contains specifications how the data and/or simulations should be plotted by the visualization routines [TSV]



Files constituting a PEtab problem

The following sections will describe the minimum requirements of those components in the core standard, which should provide all information for defining the parameter estimation problem.

Extensions of this format (e.g. additional columns in the measurement table) are possible and intended. However, those columns should provide extra information for example for plotting, or for more efficient parameter estimation, but they should not affect the optimization problem as such. Some optional extensions are described in the last section, "Extensions", of this document.

General remarks

- All model entities column and row names are case-sensitive
- Fields in "[]" in the second row are optional and may be left empty.

1.3 SBML model definition

The model must be specified as valid SBML. Since parameter estimation is beyond the scope of SBML, there exists no standard way to specify observables (model outputs) and respective noise models. Therefore, we use the following convention.

1.3.1 Observables

In the SBML model, observables are specified as `AssignmentRules` assigning to parameters with `ids` starting with `observable_` followed by the `observableId` as in the corresponding column of the *measurement table* (see below).

E.g.

```
observable_pErk = observableParameter1_pErk + observableParameter2_pErk*pErk
```

where `observableParameter1_pErk` would be an offset, and `observableParameter2_pErk` a scaling parameter for the observable `pErk`. The observable parameter names have the structure: `observableParameter${indexOfObservableParameter}_${observableId}` to facilitate automatic recognition. The specific values or parameters are assigned in the *measurement table*.

1.3.2 Noise model

Measurement noise can be specified as a numerical value in the `noiseParameters` column of the *measurement table* (see below), which will default to a Gaussian noise model with standard deviation as provided in `noiseParameters`.

Alternatively, more complex noise models can be specified for each observable, using additional `AssignmentRules`. Those noise model rules assign to `sigma_${observableId}` parameters. A noise model which accounts for relative and absolute contributions could, e.g., be defined as

```
sigma_pErk = noiseParameter1_pErk + noiseParameter2_pErk*pErk
```

with `noiseParameter1_pErk` denoting the absolute and `noiseParameter2_pErk` the relative contribution for the observable `pErk`. The noise parameter names have the structure: `noiseParameter${indexOfNoiseParameter}_${observableId}` to facilitate automatic recognition. The specific values or parameters are assigned in the *measurement table*.

Any parameters named `noiseParameter${1..n}` *must* be overwritten in the `noiseParameters` column of the measurement file (see below).

1.4 Condition table

The condition table specifies parameters or *constant* species for specific simulation condition (generally corresponding to different experimental conditions).

This is specified as tab-separated value file with condition-specific species/parameters in the following way:

Row names are condition names as referenced in the measurement table below. Column names are global parameter IDs or IDs of constant species as given in the SBML model. These parameters will override any parameter values specified in the model. `parameterOrStateId`s and `conditionId`s must be unique.

Row- and column-ordering are arbitrary, although specifying `parameterId` first may improve human readability. The `conditionName` column is optional. Additional columns are *not* allowed.

Note 1: Instead of adding additional columns to the condition table, they can easily be added to a separate file, since every row of the condition table has `parameterId` as unique key.

1.5 Measurement table

A tab-separated values files containing all measurements to be used for model training or validation.

Expected to have the following named columns in any (but preferably this) order:

(wrapped for readability)

Additional (non-standard) columns may be added. If the additional plotting functionality of PEtab should be used, such columns could be

where `datasetId` is a necessary column to use particular plotting functionality, and `replicateId` is optional, which can be used to group replicates and plot error bars.

1.5.1 Detailed field description

- `observableId` [STRING, NOT NULL, REFERENCES(sbml.observableID)]

  Observable ID with a matching parameter in the SBML model with ID `observable_${observableId}`

- `preequilibrationConditionId` [STRING OR NULL, REFERENCES(conditionsTable.conditionID)]

  The `conditionId` to be used for preequilibration. E.g. for drug treatments the model would be preequilibrated with the no-drug condition. Empty for no preequlibration.

- `simulationConditionId` [STRING, NOT NULL, REFERENCES(conditionsTable.conditionID)]

  `conditionId` as provided in the condition table, specifying the condition-specific parameters used for simulation.

- `measurement` [NUMERIC, NOT NULL]

  The measured value in the same units/scale as the model output.

- `time` [NUMERIC OR STRING, NOT NULL]

  Time point of the measurement in the time unit specified in the SBML model, numeric value or `inf` (lower-case) for steady-state measurements.

- `observableParameters` [STRING OR NULL]

  This field allows overriding or introducing condition-specific versions of parameters defined in the model. The model can define observables (see above) containing place-holder parameters which can be replaced by condition-specific dynamic or constant parameters. Placeholder parameters must be named `observableParameter${n}_${observableId}` with `n` ranging from 1 (not 0) to the number of placeholders for the given observable, without gaps. If the observable specified under `observableId` contains no placeholders, this field must be empty. If it contains `n > 0` placeholders, this field must hold `n` semicolon-separated numeric values or parameter names. No trailing semicolon must be added.

  Different lines for the same `observableId` may specify different parameters. This may be used to account for condition-specific or batch-specific parameters. This will translate into an extended optimization parameter vector.

  All placeholders defined in the model must be overwritten here. If there are not placeholders in the model, this column may be omitted.

- `noiseParameters` [STRING]

  The measurement standard deviation or `NaN` if the corresponding sigma is a model parameter.

  Numeric values or parameter names are allowed. Same rules apply as for `observableParameters` in the previous point.

- `observableTransformation` [STRING]

  Transformation of the observable. `lin`, `log` or `log10`. Defaults to 'lin'.

- `noiseDistribution` [STRING: 'normal' or 'laplace']

  Assumed Noise distribution for the given measurement. Only normally or Laplace distributed noise is currently allowed. Defaults to `normal`. If `normal`, the specified `noiseParameters` will be interpreted as standard deviation (*not* variance).

- `datasetId` [STRING, optional]

  The datasetId is used to group certain measurements to datasets. This is typically the case for data points which belong to the same observable, the same simulation and preequilibration condition, the same noise model, the same observable tranformation and the same observable parameters. This grouping makes it possible to use the plotting routines which are provided in the PEtab repository.

- `replicateId` [STRING, optional]

  The replicateId can be used to discern replicates with the same datasetId, which is helpful for plotting e.g. error bars.

1.6 Parameter table

A tab-separated value text file containing information on model parameters.

This table must include the following parameters:

- Named parameter overrides introduced in the *conditions table*
- Named parameter overrides introduced in the *measurement table*

and must not include

- placeholder parameters (see `observableParameters` and `noiseParameters` above)
- parameters included as column names in the *condition table*

One row per parameter with arbitrary order of rows and columns:

Additional columns may be added.

1.6.1 Detailed field description:

- `parameterId` [STRING, NOT NULL, REFERENCES(sbml.parameterId)]

  The `parameterId` of the parameter described in this row. This has be identical to the parameter IDs specified in the SBML model or in the `observableParameters` or `noiseParameters` column of the measurement table (see above).

  There must exist one line for each parameterId specified in the SBML model (except for placeholder parameter, see above) or the `observableParameters` or `noiseParameters` column of the measurement table.

- `parameterName` [STRING, OPTIONAL]

  Parameter name to be used e.g. for plotting etc. Can be chosen freely. May or may not coincide with the SBML parameter name.

- `parameterScale` [lin|log|log10]

  Scale of the parameter. The parameters and boundaries and the nominal parameter value in the following fields are expected to be given in this scale.

- `lowerBound` [NUMERIC]

  Lower bound of the parameter used for optimization. Optional, if `estimate==0`.

- `upperBound` [NUMERIC]

  Upper bound of the parameter used for optimization. Optional, if `estimate==0`.

- `nominalValue` [NUMERIC]

  Some parameter value (scale as specified in `parameterScale`) to be used if the parameter is not subject to estimation (see `estimate` below). Optional, unless `estimate==0`.

- `estimate` [BOOL 0|1]

  1 or 0, depending on, if the parameter is estimated (1) or set to a fixed value(0) (see `nominalValue`).

- `priorType`

  Type of prior, which is used for sampling of initial points for a possible optimization and for the objective function. Priors which are only used for sampling of initial starting points or only for optimization should be specified in the additional columns `initializationPriorType` or `objectivePriorType`, respectivly. Possible prior types are (see Extensions):

  - uniform: flat prior on linear parameters
  - normal: Gaussian prior on linear parameters
  - laplace: Laplace prior on linear parameters
  - logNormal: exponentiated Gaussian prior on linear parameters
  - logLaplace: exponentiated Laplace prior on linear parameters
  - parameterScaleUniform (default): Flat prior on original parameter scale (equivalent to "no prior")
  - parameterScaleNormal: Gaussian prior on original parameter scale
  - parameterScaleLaplace: Laplace prior on original parameter scale

- `priorParameters`

  Parameters for prior specified in `priorType`, separated by a semicolon. Accordingly, there are optional columns for priors which should be used for initial point sampling or optimization only. (i.e., `initializationPriorParameters` and `objectivePriorParameters`, respectively) So far, only numeric values will be supported, no parameter names. Parameters for the different prior types are:

  - uniform: lower bound; upper bound
  - normal: mean; standard deviation (**not** variance)
  - laplace: location; scale
  - logNormal: parameters of corresp. normal distribution (see: normal)
  - logLaplace: parameters of corresp. Laplace distribution (see: laplace)
  - parameterScaleUniform: lower bound; upper bound
  - parameterScaleNormal: mean; standard deviation (**not** variance)
  - parameterScaleLaplace: location; scale

1.7 Parameter estimation problems combining multiple models

Issue #49

1.8 Visualization table

A tab-separated value file containing the specification of the visualization routines which come with the PEtab repository. Plots are in general collections of different datasets as specified using their `datasetId` (if provided) inside the measurement table.

Expected to have the following columns in any (but preferably this) order:

(wrapped for readability)

(wrapped for readability)

1.8.1 Detailed field description:

- `plotId` [STRING, NOT NULL]

An ID which corresponds to a specific plot. All datasets with the same plotId will be plotted into the same axes object.

- `plotName` [STRING]

A name for the specific plot.

- `plotTypeSimulation` [STRING]

The type of the corresponding plot, can be `LinePlot` or `BarPlot`. Default is `LinePlot`.

- `plotTypeData`

The type how replicates should be handled, can be `MeanAndSD`, `MeanAndSEM`, `replicate` (for plotting all replicates separately), or `provided` (if numeric values for the noise level are provided in the measurement table). Default is `MeanAndSD`.

- `datasetId` [STRING, NOT NULL, REFERENCES(measurementTable.datasetId)]

The datasets, which should be grouped into one plot.

- `xValues` [STRING]

The independent variable, which will be plotted on the x-axis. Can be `time` (default, for time resolved data), or it can be `parameterOrStateId` for dose-response plots. The coresponding numeric values will be shown on the x-axis.

- `xOffset` [NUMERIC]

Possible data-offsets for the independent variable (default is `0`).

- `xLabel` [STRING]

Label for the x-axis.

- `xScale` [STRING]

Scale of the independent variable, can be `lin`, `log`, or `log10`.

- `yValues` [observableId, REFERENCES(measurementTable.observableId)]

The observable which should be plotted on the y-axis.

- `yOffset` [NUMERIC]

Possible data-offsets for the observable (default is `0`).

- `yLabel` [STRING]

Label for the y-axis.

- `yScale` [STRING]

Scale of the observable, can be `lin`, `log`, or `log10`.

- `legendEntry` [STRING]

The name that should be displayed for the corresponding dataset in the legend and which defaults to `datasetId`.

1.9 Extensions

Additional columns, such as `Color`, etc. may be specified.

1.9.1 Parameter table

Extra columns

- `hierarchicalOptimization` (optional)

  hierarchicalOptimization: 1 if parameter is optimized using hierarchical optimization approach. 0 otherwise.

- `initializationPriorType` (optional)

  Prior types used for sampling of initial point for optimization. Uses the entries from `priorType` as default, but will overwrite those, if something else is specified here. For more detailed documentation, see `priorType`.

- `initializationPriorParameters` (optional)

  Prior parameters used for sampling of initial point for optimization. Uses the entries from `priorParameters` as default, but will overwrite those, if something else is specified here. For more detailed documentation, see `priorParameters`.

- `objectivePriorType` (optional)

  Prior types used for the objective function during optimization. Uses the entries from `priorType` as default, but will overwrite those, if something else is specified here. For more detailed documentation, see `priorType`.

- `objectivePriorParameters` (optional)

  Prior parameters used for the objective function during optimization. Uses the entries from `priorParameters` as default, but will overwrite those, if something else is specified here. For more detailed documentation, see `priorParameters`.

CHAPTER 2

API Reference

petab	PEtab exports
petab.core	PEtab core functions
petab.lint	Integrity checks and tests for specific features used
petab.parameter_mapping	Functions related to mapping parameter from model to parameter estimation problem
petab.petablint	Command line tool to check for correct format
petab.sbml	Functions for direct access of SBML models

2.1 petab

PEtab exports

2.2 petab.core

PEtab core functions

Functions

create_condition_df(parameter_ids, condition_ids)	Create empty condition dataframe
create_measurement_df()	Create empty measurement dataframe
create_parameter_df(sbml_model, ...)	Create a new PEtab parameter table
flatten_timepoint_specific_output_overrides()	If the PEtab problem definition has timepoint-specific observableParameters or noiseParameters for the same observable, replace those by replicating the respective observable.

Continued on next page

Table 2 – continued from previous page

get_condition_df(condition_file_name)	Read the provided condition file into a *pandas.Dataframe*
get_default_condition_file_name(model_name, ...)	Get file name according to proposed convention
get_default_measurement_file_name(...)	Get file name according to proposed convention
get_default_parameter_file_name(model_name, ...)	Get file name according to proposed convention
get_default_sbml_file_name(model_name, folder)	Get file name according to proposed convention
get_measurement_df(measurement_file_name)	Read the provided measurement file into a *pandas.Dataframe*.
get_measurement_parameter_ids(measurement_df)	Return list of ID of parameters which occur in measurement table as observable or noise parameter overrides.
get_model_parameters(sbml_model)	Return list of SBML model parameter IDs which are not AssignmentRule targets for observables or sigmas
get_noise_distributions(measurement_df)	Returns dictionary of cost definitions per observable, if specified.
get_notnull_columns(df, candidates)	Return list of df-columns in candidates which are not all null/nan.
get_observable_id(parameter_id)	Get observable id from sigma or observable parameter_id
get_observables(sbml_model, remove)	Returns dictionary of observable definitions.
get_optimization_parameters(parameter_df)	Get list of optimization parameter ids from parameter dataframe.
get_parameter_df(parameter_file_name)	Read the provided parameter file into a *pandas.Dataframe*.
get_placeholders(formula_string, ...)	Get placeholder variables in noise or observable definition for the given observable ID.
get_priors_from_df(parameter_df)	Create list with information about the parameter priors
get_required_parameters_for_parameter_table()	Get set of parameters which need to go into the parameter table
get_rows_for_condition(measurement_df, ...)	Extract rows in *measurement_df* for *condition* according to 'preequilibrationConditionId' and 'simulationConditionId' in *condition*.
get_sigmas(sbml_model, remove)	Returns dictionary of sigma definitions.
get_simulation_conditions(measurement_df)	Create a table of separate simulation conditions.
measurements_have_replicates(measurement_df)	Tests whether the measurements come with replicates
parameter_is_offset_parameter(parameter, formula)	Returns true if parameter *parameter* is an offset parameter with positive sign in formula *formula*.
parameter_is_scaling_parameter(parameter, ...)	Returns true if parameter *parameter* is a scaling parameter in formula *formula*.
sample_from_prior(prior, n_starts)	Creates samples based on prior
sample_parameter_startpoints(parameter_df, ...)	Create numpy.array with starting points for an optimization
sbml_parameter_is_observable(sbml_parameter)	Returns whether the *libsbml.Parameter* *sbml_parameter* matches the defined observable format.
sbml_parameter_is_sigma(sbml_parameter)	Returns whether the *libsbml.Parameter* *sbml_parameter* matches the defined sigma format.
split_parameter_replacement_list(...)	Split values in observableParameters and noiseParameters in measurement table.

Classes

Problem(sbml_model, sbml_reader, ...)	PEtab parameter estimation problem as defined by - SBML model - condition table - measurement table - parameter table [optional]

class petab.core.**Problem**(*sbml_model: libsbml.Model = None*, *sbml_reader: libsbml.SBMLReader = None*, *sbml_document: libsbml.SBMLDocument = None*, *condition_df: pandas.core.frame.DataFrame = None*, *measurement_df: pandas.core.frame.DataFrame = None*, *parameter_df: pandas.core.frame.DataFrame = None*)

Bases: object

PEtab parameter estimation problem as defined by - SBML model - condition table - measurement table - parameter table [optional]

condition_df
@type pandas.DataFrame

measurement_df
@type pandas.DataFrame

parameter_df
@type pandas.DataFrame

sbml_reader
@type libsbml.SBMLReader Stored to keep object alive.

sbml_document
@type libsbml.Document Stored to keep object alive.

sbml_model
@type libsbml.Model

create_parameter_df(**args*, ***kwargs*)
Create a new PEtab parameter table

See create_parameter_df

static from_files(*sbml_file: str = None*, *condition_file: str = None*, *measurement_file: str = None*, *parameter_file: str = None*) → petab.core.Problem
Factory method to load model and tables from files.

Parameters

- **sbml_file** – PEtab SBML model
- **condition_file** – PEtab condition table
- **measurement_file** – PEtab measurement table
- **parameter_file** – PEtab parameter table

static from_folder(*folder: str*, *model_name: str = None*) → petab.core.Problem
Factory method to use the standard folder structure and file names, i.e.

**${model_name}/** +– experimentalCondition_${model_name}.tsv +– measurementData_${model_name}.tsv +– model_${model_name}.xml +– parameters_${model_name}.tsv

Parameters

- **folder** – Path to the directory in which the files are located.
- **model_name** – If specified, overrides the model component in the file names. Defaults to the last component of *folder*.

get_constant_parameters()
Provide list of IDs of parameters which are fixed (i.e. not subject to optimization, no sensitivities w.r.t. these parameters are required).

get_dynamic_simulation_parameters()
See *get_model_parameters*

get_noise_distributions()
See *get_noise_distributions*.

get_observables(*remove: bool = False*)
Returns dictionary of observables definitions See *assignment_rules_to_dict* for details.

get_optimization_parameters()
Return list of optimization parameter IDs.

See get_optimization_parameters.

get_optimization_to_simulation_parameter_mapping(*warn_unmapped: bool = True*)
See get_simulation_to_optimization_parameter_mapping.

get_sigmas(*remove: bool = False*)
Return dictionary of observableId => sigma as defined in the SBML model. This does not include parameter mappings defined in the measurement table.

get_simulation_conditions_from_measurement_df()
See petab.get_simulation_conditions

lb
Parameter table lower bounds

sample_parameter_startpoints(*n_starts: int = 100*)
Create starting points for optimization

See sample_parameter_startpoints

ub
Parameter table upper bounds

x_fixed_indices
Parameter table non-estimated parameter indices

x_fixed_vals
Nominal values for parameter table non-estimated parameters

x_ids
Parameter table parameter IDs

x_nominal
Parameter table nominal values

petab.core.**create_condition_df**(*parameter_ids: Iterable[str], condition_ids: Iterable[str] = None*) → pandas.core.frame.DataFrame
Create empty condition dataframe

Parameters

- **parameter_ids** – the columns
- **condition_ids** – the rows

Returns An pandas.DataFrame with empty given rows and columns and all nan values

petab.core.**create_measurement_df**() → pandas.core.frame.DataFrame
Create empty measurement dataframe

petab.core.**create_parameter_df**(*sbml_model: libsbml.Model*, *condition_df: pandas.core.frame.DataFrame*, *measurement_df: pandas.core.frame.DataFrame*, *parameter_scale: str = 'log10'*, *lower_bound: Iterable[T_co] = None*, *upper_bound: Iterable[T_co] = None*) → pandas.core.frame.DataFrame
Create a new PEtab parameter table

All table entries can be provided as string or list-like with length matching the number of parameters

Parameters

- **sbml_model** – @type libsbml.Model
- **condition_df** – @type pandas.DataFrame
- **measurement_df** – @type pandas.DataFrame
- **parameter_scale** – parameter scaling
- **lower_bound** – lower bound for parameter value
- **upper_bound** – upper bound for parameter value

petab.core.**flatten_timepoint_specific_output_overrides**(*petab_problem: petab.core.Problem*) → None
If the PEtab problem definition has timepoint-specific observableParameters or noiseParameters for the same observable, replace those by replicating the respective observable.

This is a helper function for some tools which may not support such timepoint-specific mappings.

Parameters **petab_problem** – PEtab problem to work on

petab.core.**get_condition_df**(*condition_file_name: str*) → pandas.core.frame.DataFrame
Read the provided condition file into a *pandas.Dataframe*

Conditions are rows, parameters are columns, conditionId is index.

petab.core.**get_default_condition_file_name**(*model_name: str*, *folder: str = ''*)
Get file name according to proposed convention

petab.core.**get_default_measurement_file_name**(*model_name: str*, *folder: str = ''*)
Get file name according to proposed convention

petab.core.**get_default_parameter_file_name**(*model_name: str*, *folder: str = ''*)
Get file name according to proposed convention

petab.core.**get_default_sbml_file_name**(*model_name: str*, *folder: str = ''*)
Get file name according to proposed convention

petab.core.**get_measurement_df**(*measurement_file_name: str*) → pandas.core.frame.DataFrame
Read the provided measurement file into a *pandas.Dataframe*.

petab.core.**get_measurement_parameter_ids**(*measurement_df: pandas.core.frame.DataFrame*) → list
Return list of ID of parameters which occur in measurement table as observable or noise parameter overrides.

petab.core.**get_model_parameters**(*sbml_model: libsbml.Model*) → List[str]
Return list of SBML model parameter IDs which are not AssignmentRule targets for observables or sigmas

petab.core.**get_noise_distributions**(*measurement_df: pandas.core.frame.DataFrame*) → dict
Returns dictionary of cost definitions per observable, if specified.

Looks through all parameters satisfying *sbml_parameter_is_cost* and return as dictionary.

Parameters **measurement_df** – PEtab measurement table

Returns cost definition}

Return type {observableId

petab.core.**get_notnull_columns**(*df: pandas.core.frame.DataFrame, candidates: Iterable[T_co]*)
Return list of df-columns in candidates which are not all null/nan. The output can e.g. be used as input for pandas.DataFrame.groupby.

petab.core.**get_observable_id**(*parameter_id: str*) → str
Get observable id from sigma or observable parameter_id

e.g. for observable_obs1 -> obs1 sigma_obs1 -> obs1

petab.core.**get_observables**(*sbml_model: libsbml.Model*, *remove: bool = False*) → dict
Returns dictionary of observable definitions. See *assignment_rules_to_dict* for details.

petab.core.**get_optimization_parameters**(*parameter_df: pandas.core.frame.DataFrame*) → List[str]
Get list of optimization parameter ids from parameter dataframe.

petab.core.**get_parameter_df**(*parameter_file_name: str*) → pandas.core.frame.DataFrame
Read the provided parameter file into a *pandas.Dataframe*.

petab.core.**get_placeholders**(*formula_string: str*, *observable_id: str*, *override_type: str*) → set
Get placeholder variables in noise or observable definition for the given observable ID.

Parameters

- **formula_string** – observable formula (typically from SBML model)
- **observable_id** – ID of current observable
- **override_type** – 'observable' or 'noise', depending on whether *formula* is for observable or for noise model

Returns (Un-ordered) set of placeholder parameter IDs

petab.core.**get_priors_from_df**(*parameter_df: pandas.core.frame.DataFrame*)
Create list with information about the parameter priors

Parameters **parameter_df** – @type pandas.DataFrame

petab.core.**get_required_parameters_for_parameter_table**(*sbml_model: libsbml.Model*, *condition_df: pandas.core.frame.DataFrame*, *measurement_df: pandas.core.frame.DataFrame*) → Set[str]
Get set of parameters which need to go into the parameter table

petab.core.**get_rows_for_condition**(*measurement_df: pandas.core.frame.DataFrame, condition: Union[pandas.core.frame.DataFrame, dict]*) → pandas.core.frame.DataFrame
Extract rows in *measurement_df* for *condition* according to 'preequilibrationConditionId' and 'simulationConditionId' in *condition*.

Returns **cur_measurement_df** – The subselection of rows in *measurement_df* for the condition *condition*.

Return type pd.DataFrame

petab.core.**get_sigmas**(*sbml_model: libsbml.Model*, *remove: bool = False*) → dict
Returns dictionary of sigma definitions.

Keys are observable IDs, for values see *assignment_rules_to_dict* for details.

petab.core.**get_simulation_conditions**(*measurement_df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Create a table of separate simulation conditions. A simulation condition is a specific combination of simulationConditionId and preequilibrationConditionId.

Parameters **measurement_df** – PEtab measurement table

Returns Dataframe with columns 'simulationConditionId' and 'preequilibrationConditionId'. All-NULL columns will be omitted.

petab.core.**measurements_have_replicates**(*measurement_df: pandas.core.frame.DataFrame*) → bool
Tests whether the measurements come with replicates

Parameters **measurement_df** – Measurement table

Returns True if there are replicates, False otherwise

petab.core.**parameter_is_offset_parameter**(*parameter: str*, *formula: str*) → bool
Returns true if parameter *parameter* is an offset parameter with positive sign in formula *formula*.

petab.core.**parameter_is_scaling_parameter**(*parameter: str*, *formula: str*) → bool
Returns true if parameter *parameter* is a scaling parameter in formula *formula*.

petab.core.**sample_from_prior**(*prior: tuple*, *n_starts: int*)
Creates samples based on prior

Parameters

- **prior** – @type tuple
- **n_starts** – @type int

petab.core.**sample_parameter_startpoints**(*parameter_df: pandas.core.frame.DataFrame*, *n_starts: int = 100*)
Create numpy.array with starting points for an optimization

Dimension of output: n_optimization_parameters x n_startpoints

Parameters

- **parameter_df** – @type pandas.DataFrame
- **n_starts** – @type int

petab.core.**sbml_parameter_is_observable**(*sbml_parameter: libsbml.Parameter*) → bool
Returns whether the *libsbml.Parameter sbml_parameter* matches the defined observable format.

petab.core.**sbml_parameter_is_sigma**(*sbml_parameter: libsbml.Parameter*) → bool
Returns whether the *libsbml.Parameter sbml_parameter* matches the defined sigma format.

petab.core.**split_parameter_replacement_list**(*list_string: Union[str, numbers.Number]*, *delim: str = ';'*) → List[T]
Split values in observableParameters and noiseParameters in measurement table. Convert numeric values to float.

Parameters

- **delim** – delimiter

- **list_string** – delim-separated stringified list

2.3 petab.lint

Integrity checks and tests for specific features used

Functions

assert_all_parameters_present_in_parameter_df(...)	Ensure all required parameters are contained in the parameter table with no additional ones
assert_measured_observables_present_in_observable_df(...)	Check if all observables in measurement files have been specified in the model
assert_model_parameters_in_condition_or_parameter_table(...)	Model parameters that are targets of AssignmentRule must not be present in parameter table or in condition table columns.
assert_no_leading_trailing_whitespace(...)	
assert_noise_distributions_valid(measurement_df)	Check whether there are not multiple noise distributions for an observable, and that the names are correct.
assert_overrides_match_parameter_count(...)	Ensure that number of parameters in the observable definition matches the number of overrides in *measurement_df*
assert_parameter_bounds_are_numeric(parameter_df)	Check if all entries in the lowerBound and upperBound columns of the parameter table are numeric.
assert_parameter_estimate_is_boolean(...)	Check if all entries in the estimate column of the parameter table are 0 or 1.
assert_parameter_id_is_string(parameter_df)	Check if all entries in the parameterId column of the parameter table are string and not empty.
assert_parameter_id_is_unique(parameter_df)	Check if the parameterId column of the parameter table is unique.
assert_parameter_scale_is_valid(parameter_df)	Check if all entries in the parameterScale column of the parameter table are 'lin' for linear, 'log' for natural logarithm or 'log10' for base 10 logarithm.
check_condition_df(df, sbml_model)	
check_measurement_df(df)	
check_parameter_bounds(parameter_df)	Check if all entries in the lowerBound are smaller than upperBound column in the parameter table.
check_parameter_df(df, sbml_model, ...)	
condition_table_is_parameter_free(condition_df)	Check if all entries in the condition table are numeric (no parameter IDs)
lint_problem(problem)	Run PEtab validation on problem
measurement_table_has_observable_parameter_numeric_overrides(...)	Are there any numbers to override observable parameters?
measurement_table_has_timepoint_specific_mappings(...)	Are there time-point or replicate specific parameter assignments in the measurement table.

petab.lint.**assert_all_parameters_present_in_parameter_df**(*parameter_df: pandas.core.frame.DataFrame*, *sbml_model: libsbml.Model*, *measurement_df: pandas.core.frame.DataFrame*, *condition_df: pandas.core.frame.DataFrame*)

Ensure all required parameters are contained in the parameter table with no additional ones

petab.lint.**assert_measured_observables_present_in_model**(*measurement_df*, *sbml_model*)

Check if all observables in measurement files have been specified in the model

petab.lint.**assert_model_parameters_in_condition_or_parameter_table**(*sbml_model: libsbml.Model*, *condition_df: pandas.core.frame.DataFrame*, *parameter_df: pandas.core.frame.DataFrame*)

Model parameters that are targets of AssignmentRule must not be present in parameter table or in condition table columns. Other parameters must only be present in either in parameter table or condition table columns. Check that.

petab.lint.**assert_noise_distributions_valid**(*measurement_df*)

Check whether there are not multiple noise distributions for an observable, and that the names are correct.

petab.lint.**assert_overrides_match_parameter_count**(*measurement_df*, *observables*, *noise*)

Ensure that number of parameters in the observable definition matches the number of overrides in *measurement_df*

:param : param measurement_df: :param : param observables: dict: obsId => {obsFormula} :param : param noise: dict: obsId => {obsFormula}

petab.lint.**assert_parameter_bounds_are_numeric**(*parameter_df*)

Check if all entries in the lowerBound and upperBound columns of the parameter table are numeric.

petab.lint.**assert_parameter_estimate_is_boolean**(*parameter_df*)

Check if all entries in the estimate column of the parameter table are 0 or 1.

petab.lint.**assert_parameter_id_is_string**(*parameter_df*)

Check if all entries in the parameterId column of the parameter table are string and not empty.

petab.lint.**assert_parameter_id_is_unique**(*parameter_df*)

Check if the parameterId column of the parameter table is unique.

petab.lint.**assert_parameter_scale_is_valid**(*parameter_df*)

Check if all entries in the parameterScale column of the parameter table are 'lin' for linear, 'log' for natural logarithm or 'log10' for base 10 logarithm.

petab.lint.**check_parameter_bounds**(*parameter_df*)

Check if all entries in the lowerBound are smaller than upperBound column in the parameter table.

petab.lint.**condition_table_is_parameter_free**(*condition_df*)

Check if all entries in the condition table are numeric (no parameter IDs)

petab.lint.**lint_problem**(*problem: petab.core.Problem*)
Run PEtab validation on problem

Parameters **problem** – PEtab problem to check

Returns True is errors occurred, False otherwise

petab.lint.**measurement_table_has_observable_parameter_numeric_overrides**(*measurement_df*)
Are there any numbers to override observable parameters?

petab.lint.**measurement_table_has_timepoint_specific_mappings**(*measurement_df*)
Are there time-point or replicate specific parameter assignments in the measurement table.

2.4 petab.parameter_mapping

Functions related to mapping parameter from model to parameter estimation problem

Functions

fill_in_nominal_values(mapping, Union[str, ...)	Replace non-estimated parameters in mapping list for a given condition by nominalValues provided in parameter table.
get_optimization_to_simulation_parameter_mapping(...)	Create list of mapping dicts from PEtab-problem to SBML parameters.
get_optimization_to_simulation_scale_mapping(...)	Get parameter scale mapping for all conditions
get_parameter_mapping_for_condition(...)	Create dictionary of mappings from PEtab-problem to SBML parameters for the given condition.
get_scale_mapping_for_condition(...)	Get parameter scale mapping for the given condition.
handle_missing_overrides(...)	Find all observable parameters and noise parameters that were not mapped and set their mapping to np.nan.
merge_preeq_and_sim_pars_condition(...)	Merge preequilibration and simulation parameters and scales while checking for compatibility.
perform_mapping_checks(measurement_df)	

petab.parameter_mapping.**_apply_dynamic_parameter_overrides**(*mapping: Dict[str, Union[str, numbers.Number]], condition_id: str, condition_df: pandas.core.frame.DataFrame*) → None
Apply dynamic parameter overrides from condition table (in-place).

Parameters

- **par_sim_id_to_ix** (*mapping,*) – see get_parameter_mapping_for_condition
- **condition_df** – PEtab condition and parameter table

petab.parameter_mapping.**_apply_output_parameter_overrides**(*mapping: Dict[str, Union[str, numbers.Number]]*, *cur_measurement_df: pandas.core.frame.DataFrame*) → None

Apply output parameter overrides to the parameter mapping dict for a given condition as defined in the measurement table (observableParameter, noiseParameters).

Parameters

- **mapping** – parameter mapping dict
- **cur_measurement_df** – Subset of the measurement table for the current condition

petab.parameter_mapping.**_apply_overrides_for_observable**(*mapping: Dict[str, Union[str, numbers.Number]]*, *observable_id: str*, *override_type: str*, *overrides: list*) → None

Apply parameter-overrides for observables and noises to mapping matrix.

Parameters

- **mapping** – mapping dict to which to apply overrides
- **observable_id** – observable ID
- **override_type** – 'observable' or 'noise'
- **overrides** – list of overrides for noise or observable parameters

petab.parameter_mapping.**fill_in_nominal_values**(*mapping: Dict[str, Union[str, numbers.Number]]*, *parameter_df: pandas.core.frame.DataFrame*) → None

Replace non-estimated parameters in mapping list for a given condition by nominalValues provided in parameter table.

Parameters

- **mapping** – mapping dict obtained from get_parameter_mapping_for_condition
- **parameter_df** – PEtab parameter table

petab.parameter_mapping.**get_optimization_to_simulation_parameter_mapping**(*condition_df: pandas.core.frame.DataFrame*, *measurement_df: pandas.core.frame.DataFrame*, *parameter_df: pandas.core.frame.DataFrame = None*, *sbml_model: libsbml.Model = None*, *simulation_conditions=None*, *warn_unmapped: bool = True*) → List[Tuple[Dict[str, Union[str, numbers.Number]], Dict[str, Union[str, numbers.Number]]]]

Create list of mapping dicts from PEtab-problem to SBML parameters.

Parameters

- **measurement_df, parameter_df** (*condition_df,*) – The dataframes in the PEtab format.

  parameter_df is optional if par_sim_ids is provided

- **sbml_model** – The sbml model with observables and noise specified according to the petab format. Optional if par_sim_ids is provided.
- **simulation_conditions** (*pd.DataFrame*) – Table of simulation conditions as created by *petab.get_simulation_conditions*.
- **warn_unmapped** – If True, log warning regarding unmapped parameters

Returns

- *The length of the returned array is n_conditions, each entry is a tuple of*
- *two dicts of length n_par_sim, listing the optimization parameters or*
- *constants to be mapped to the simulation parameters, first for*
- *preequilibration (empty if no preequilibration condition is specified),*
- *second for simulation. NaN is used where no mapping exists.*

petab.parameter_mapping.**get_optimization_to_simulation_scale_mapping**(*parameter_df: pandas.core.frame.DataFrame*, *mapping_par_opt_to_par_sim: List[Tuple[Dict[str, Union[str, numbers.Number]], Dict[str, Union[str, numbers.Number]]]]*, *measurement_df: pandas.core.frame.DataFrame*, *simulation_conditions: Union[dict, pandas.core.frame.DataFrame]* = *None*) → List[Tuple[Dict[str, str], Dict[str, str]]]

Get parameter scale mapping for all conditions

petab.parameter_mapping.**get_parameter_mapping_for_condition**(*condition_id: str*, *is_preeq: bool*, *cur_measurement_df: pandas.core.frame.DataFrame*, *condition_df: pandas.core.frame.DataFrame*, *parameter_df: pandas.core.frame.DataFrame = None*, *sbml_model: libsbml.Model = None*, *warn_unmapped: bool = True*) → Dict[str, Union[str, numbers.Number]]

Create dictionary of mappings from PEtab-problem to SBML parameters for the given condition.

Parameters

- **condition_id** (*Condition ID for which to perform mapping*) –
- **is_preeq** (*If true, output parameters will not be mapped*) –
- **cur_measurement_df** (*Measurement sub-table for current condition*) –
- **parameter_df** (*condition_df,*) – The dataframes in the PEtab format. parameter_df is optional if par_sim_ids is provided
- **sbml_model** – The sbml model with observables and noise specified according to the petab format. Optional if par_sim_ids is provided.
- **warn_unmapped** – If True, log warning regarding unmapped parameters

Returns

- *Dictionary of parameter IDs with mapped parameters IDs to be estimated or*
- *filled in values in case of non-estimated parameters. NaN is used where no*
- *mapping exists.*

petab.parameter_mapping.**get_scale_mapping_for_condition**(*parameter_df: pandas.core.frame.DataFrame, mapping_par_opt_to_par_sim: Dict[str, Union[str, numbers.Number]]*) → Dict[str, str]

Get parameter scale mapping for the given condition.

Parameters

- **parameter_df** – PEtab parameter table
- **mapping_par_opt_to_par_sim** – Mapping as obtained from get_parameter_mapping_for_condition

petab.parameter_mapping.**handle_missing_overrides**(*mapping_par_opt_to_par_sim: Dict[str, Union[str, numbers.Number]], warn: bool = True, condition_id: str = None*) → None

Find all observable parameters and noise parameters that were not mapped and set their mapping to np.nan.

Assumes that parameters matching "(noise|observable)Parameter[0-9]+_" were all supposed to be overwritten.

Parameters

- **mapping_par_opt_to_par_sim** – Output of get_parameter_mapping_for_condition
- **warn** – If True, log warning regarding unmapped parameters

petab.parameter_mapping.**merge_preeq_and_sim_pars_condition**(*condition_map_preeq: Dict[str, Union[str, numbers.Number]], condition_map_sim: Dict[str, Union[str, numbers.Number]], condition_scale_map_preeq: Dict[str, str], condition_scale_map_sim: Dict[str, str], condition: Any*) → None

Merge preequilibration and simulation parameters and scales while checking for compatibility.

This function is meant for the case where we cannot have different parameters (and scales) for preequilibration and simulation. Therefore, merge both and ensure matching scales and parameters. *condition_map_sim* and *condition_scale_map_sim* will ne modified in place.

Parameters

- **condition_map_sim** (*condition_map_preeq,*) – Parameter mapping as obtained from *get_parameter_mapping_for_condition*
- **condition_scale_map_sim** (*condition_scale_map_preeq,*) – Parameter scale mapping as obtained from *get_get_scale_mapping_for_condition*
- **condition** – Condition identifier for more informative error messages

2.5 petab.petablint

Command line tool to check for correct format

Functions

main()	
parse_cli_args()	Parse command line arguments

Classes

LintFormatter([fmt, datefmt, style])	Custom log formatter

class petab.petablint.**LintFormatter**(*fmt=None*, *datefmt=None*, *style='%'*)

Bases: logging.Formatter

Custom log formatter

converter()

localtime([seconds]) -> (tm_year,tm_mon,tm_mday,tm_hour,tm_min,
tm_sec,tm_wday,tm_yday,tm_isdst)

Convert seconds since the Epoch to a time tuple expressing local time. When 'seconds' is not passed in, convert the current time instead.

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

formatException(*ei*)

Format and return the specified exception information as a string.

This default implementation just uses traceback.print_exception()

formatStack(*stack_info*)

This method is provided as an extension point for specialized formatting of stack information.

The input data is a string as returned from a call to traceback.print_stack(), but with the last trailing newline removed.

The base implementation just returns the value passed in.

formatTime(*record*, *datefmt=None*)

Return the creation time of the specified LogRecord as formatted text.

This method should be called from format() by a formatter which wants to make use of a formatted time. This method can be overridden in formatters to provide for any specific requirement, but the basic behaviour is as follows: if datefmt (a string) is specified, it is used with time.strftime() to format the creation time of the record. Otherwise, an ISO8601-like (or RFC 3339-like) format is used. The resulting string is returned. This function uses a user-configurable function to convert the creation time to a tuple. By default, time.localtime() is used; to change this for a particular formatter instance, set the 'converter' attribute to a function with the same signature as time.localtime() or time.gmtime(). To change it for all formatters, for example if you want all logging times to be shown in GMT, set the 'converter' attribute in the Formatter class.

usesTime()

Check if the format uses the creation time of the record.

petab.petablint.**parse_cli_args**()

Parse command line arguments

2.6 petab.sbml

Functions for direct access of SBML models

Functions

add_global_parameter(sbml_model, ...)	Add new global parameter to SBML model
add_model_output(sbml_model, observable_id, ...)	Add PEtab-style output to model
add_model_output_sigma(sbml_model, ...)	Add PEtab-style sigma for the given observable id
add_model_output_with_sigma(sbml_model, ...)	Add PEtab-style output and corresponding sigma with single (newly created) parameter
assignment_rules_to_dict(sbml_model[, ...])	Turn assignment rules into dictionary.
constant_species_to_parameters(sbml_model)	Convert constant species in the SBML model to constant parameters.
create_assigment_rule(sbml_model, ...)	Create SBML AssignmentRule
globalize_parameters(sbml_model, ...)	Turn all local parameters into global parameters with the same properties
is_sbml_consistent(sbml_document, check_units)	Check for SBML validity / consistency
log_sbml_errors(sbml_document[, ...])	Log libsbml errors

petab.sbml.**add_global_parameter**(*sbml_model: libsbml.Model*, *parameter_id: str*, *parameter_name: str = None*, *constant: bool = False*, *units: str = 'dimensionless'*, *value: float = 0.0*) → libsbml.Parameter

Add new global parameter to SBML model

petab.sbml.**add_model_output**(*sbml_model: libsbml.Model*, *observable_id: str*, *formula: str*, *observable_name: str = None*) → None

Add PEtab-style output to model

We expect that all formula parameters are added to the model elsewhere.

Parameters

- **sbml_model** – Model to add output to
- **formula** – Formula string for model output
- **observable_id** – ID without "observable_" prefix
- **observable_name** – Any observable name

petab.sbml.**add_model_output_sigma**(*sbml_model: libsbml.Model*, *observable_id: str*, *formula: str*)

Add PEtab-style sigma for the given observable id

We expect that all formula parameters are added to the model elsewhere.

Parameters

- **sbml_model** – Model to add to
- **observable_id** – Observable id for which to add sigma
- **formula** – Formula for sigma

petab.sbml.**add_model_output_with_sigma**(*sbml_model: libsbml.Model*, *observable_id: str*, *observable_formula: str*, *observable_name: str = None*)

Add PEtab-style output and corresponding sigma with single (newly created) parameter

We expect that all formula parameters are added to the model elsewhere.

Parameters

- **sbml_model** – Model to add output to
- **observable_formula** – Formula string for model output
- **observable_id** – ID without "observable_" prefix
- **observable_name** – Any name

petab.sbml.**assignment_rules_to_dict**(*sbml_model: libsbml.Model*, *filter_function=<function <lambda>>*, *remove: bool = False*) → Dict[str, Dict[str, Any]]

Turn assignment rules into dictionary.

Parameters

- **sbml_model** – a sbml model instance.
- **filter_function** – callback function taking assignment variable as input and returning True/False to indicate if the respective rule should be turned into an observable.
- **remove** – Remove the all matching assignment rules from the model

Returns

```
{
    assigneeId:
    {
        'name': assigneeName,
        'formula': formulaString
    }
}
```

petab.sbml.**constant_species_to_parameters**(*sbml_model: libsbml.Model*) → list

Convert constant species in the SBML model to constant parameters.

This can be used e.g. for setting up models with condition-specific constant species for PEtab, since there it is not possible to specify constant species in the condition table.

Parameters **sbml_model** – libsbml model instance

Returns species IDs that have been turned into constants

petab.sbml.**create_assigment_rule**(*sbml_model: libsbml.Model*, *assignee_id: str*, *formula: str*, *rule_id: str = None*, *rule_name: str = None*) → libsbml.AssignmentRule

Create SBML AssignmentRule

Parameters

- **sbml_model** – Model to add output to
- **assignee_id** – Target of assignment
- **formula** – Formula string for model output
- **rule_id** – SBML id for created rule
- **rule_name** – SBML name for created rule

petab.sbml.**globalize_parameters**(*sbml_model: libsbml.Model*, *prepend_reaction_id: bool = False*) → None

Turn all local parameters into global parameters with the same properties

Local parameters are currently ignored by other PEtab functions. Use this function to convert them to global parameters. There may exist local parameters with identical IDs within different kinetic laws. This is not checked here. If in doubt that local parameter IDs are unique, enable *prepend_reaction_id* to create global parameters named ${reaction_id}_${local_parameter_id}.

Parameters

- **sbml_model** – The SBML model to operate on
- **prepend_reaction_id** – Prepend reaction id of local parameter when creating global parameters

petab.sbml.**is_sbml_consistent**(*sbml_document: libsbml.SBMLDocument*, *check_units: bool = False*)

Check for SBML validity / consistency

Parameters

- **sbml_document** – SBML document to check
- **check_units** – Also check for unit-related issues

Returns False if problems were detected, otherwise True

petab.sbml.**log_sbml_errors**(*sbml_document: libsbml.SBMLDocument*, *minimum_severity=1*) → None

Log libsbml errors

Parameters

- **sbml_document** – SBML document to check
- **minimum_severity** – Minimum severity level to report (see libsbml)

CHAPTER 3

PEtab changelog

3.1 0.0.0a17

Data format: *No changes*

Library:

- Extended visualization support
- Add helper function and test case to deal with timepoint-specific parameters flatten_timepoint_specific_output_overrides (#128) (Closes #125)
- Fix get_noise_distributions: so far we got 'normal' everywhere due to wrong grouping (#147)
- Fix create_parameter_df: Exclude rule targets (#149)
- Verify condition table column names occur as model parameters (Closes #150) (#151)
- More informative error messages in case of wrongly set observable and noise parameters (Closes #118) (#155)
- Update doc for copasi import and github installation (#158)
- Extend validator to check if all required parameters are present in parameter table (Closes #43) (#159)
- Setup documentation for RTD (#161)
- Handle None in petab.core.split_parameter_replacement_list (Closes #121)
- Fix(lint) correct handling of optional columns. Check before access.
- Remove obsolete generate_experiment_id.py (Closes #111) #166

3.2 0.0.0a16 and earlier

See git history

CHAPTER 4

License

```
MIT License

Copyright (c) 2018 Data-driven Computational Modelling

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.
```

CHAPTER 5

PEtab logo license

The PEtab logo is free for use under the CC0 license.

Logo

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

C

F

G

H

I

L

M

P

S

U

X