PEtab

Release latest

Feb 03, 2020

Data format

CHAPTER 1

PEtab data format specification

1.1 Version: 1

This document explains the PEtab data format.

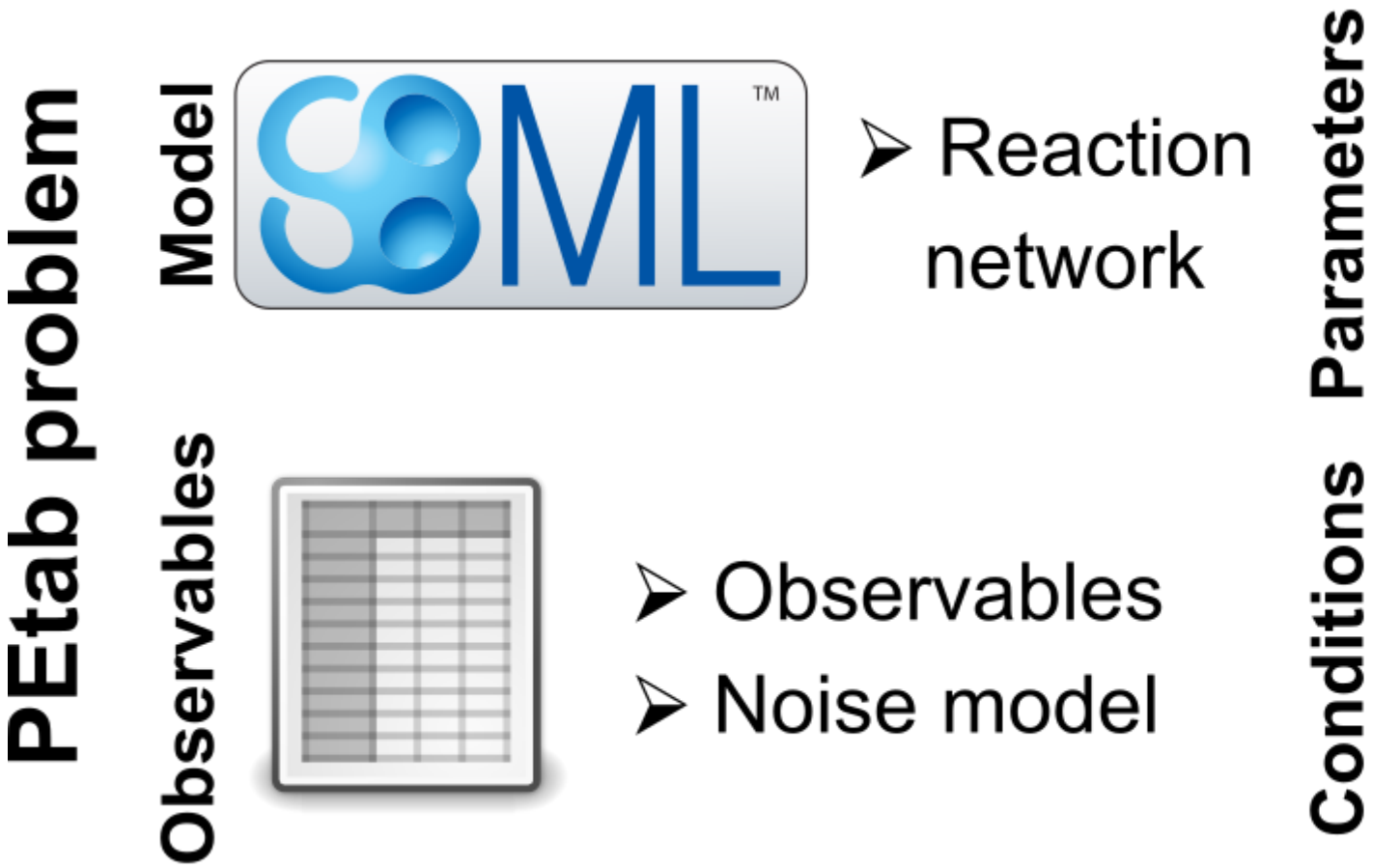
1.2 Purpose

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

1.3 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]
- An observable file specifying the observation model [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]



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, downstream analysis, or for more efficient parameter estimation, but they should not affect the optimization problem as such.

General remarks

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

1.4 SBML model definition

The model must be specified as valid SBML. There are no further restrictions.

1.5 Condition table

The condition table specifies parameters, or initial values of species and compartments for specific simulation conditions (generally corresponding to different experimental conditions).

This is specified as a tab-separated value file in the following way:

Row- and column-ordering are arbitrary, although specifying `conditionId` first may improve human readability.

Additional columns are *not* allowed.

1.5.1 Detailed field description

- `conditionId` [STRING, NOT NULL]

  Unique identifier for the simulation/experimental condition, to be referenced by the measurement table described below.

- `conditionName` [STRING, OPTIONAL]

  Condition names are arbitrary strings to describe the given condition. They may be used for reporting or visualization.

- `${parameterOrStateOrCompartmentId1}`

  Further columns may be global parameter IDs, IDs of species or compartments as defined in the SBML model. Only one column is allowed per ID. Values for these condition parameters may be provided either as numeric values, or as IDs defined in the SBML model, the parameter table or both.

  - `${parameterId}`

    The values will override any parameter values specified in the model.

  - `${speciesId}`

    If a species ID is provided, it is interpreted as the initial concentration/amount of that species and will override the initial concentration/amount given in the SBML model or given by a preequilibration condition. If `NaN` is provided for a condition, the result of the preequilibration (or initial concentration/amount from the SBML model, if no preequilibration is defined) is used.

  - `${compartmentId}`

    If a compartment ID is provided, it is interpreted as the initial compartment size.

1.6 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.

1.6.1 Detailed field description

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

  Observable ID as defined in the observables table described below.

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

  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, OPTIONAL]

  This field allows overriding or introducing condition-specific versions of output parameters defined in the observation model. The model can define observables (see below) 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 observation model must be overwritten here. If there are no placeholders used, this column may be omitted.

- `noiseParameters` [STRING, OPTIONAL]

  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.

- `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.

1.6.2 Observables table

Parameter estimation requires linking experimental observations to the model of interest. Therefore, one needs to define observables (model outputs) and respective noise models, which represent the measurement process. Since parameter estimation is beyond the scope of SBML, there exists no standard way to specify observables (model outputs) and respective noise models. Therefore, in PEtab observables are specified in a separate table as described in

the following. This allows for a clear separation of the observation model and the underlying dynamic model, which allows, in most cases, to reuse any existing SBML model without modifications.

The observable table has the following columns:

1.6.3 Detailed field description:

- `observableId` [STRING]

  Any identifier which would be a valid identifier in SBML. This is referenced by the `observableId` column in the measurement table.

- [`observableName`] [STRING, OPTIONAL]

  Name of the observable. Only used for output, not for identification.

- `observableFormula` [STRING]

  Observation function as plain text formula expression. May contain any symbol defined in the SBML model or parameter table. In the simplest case just an SBML species ID or an `AssignmentRule` target.

  May introduce new parameters of the form `observableParameter${n}`, which are overridden by `observableParameters` in the measurement table (see description there).

- `observableTransformation` [STRING, OPTIONAL]

  Transformation of the observable and measurement for computing the objective function. Must be one of `lin`, `log` or `log10`. Defaults to `lin`. The measurements and model outputs are both assumed to be provided in linear space.

- `noiseFormula` [STRING]

  Noise model parameters as plain text formula expression.

  Measurement noise can be specified as a numerical value which will default to a Gaussian noise model if not specified differently in `noiseDistribution` with standard deviation as provided here. In this case, the same standard deviation is assumed for all measurements for the given observable.

  Alternatively, some formula expression can be provided to specify more complex noise models. A noise model which accounts for relative and absolute contributions could, e.g., be defined as

  ```
  noiseParameter1_pErk + noiseParameter2_pErk*pErk
  ```

  with `noiseParameter1_pErk` denoting the absolute and `noiseParameter2_pErk` the relative contribution for the observable `pErk`. IDs of noise parameters that need to have different values for different measurements have the structure: `noiseParameter${indexOfNoiseParameter}_${observableId}` to facilitate automatic recognition. The specific values or parameters are assigned in the `noiseParameters` field of the *measurement table* (see above). Any parameters named `noiseParameter${1..n}` *must* be overwritten in the measurement table.

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

  Assumed noise distribution for the given measurement. Only normally or Laplace distributed noise is currently allowed (log-normal and log-laplace are obtained by setting `observableTransformation` to `log`). Defaults to `normal`. If `normal`, the specified `noiseParameters` will be interpreted as standard deviation (*not* variance).

1.7 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*, unless defined in the SBML model
- 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*
- Parameters that are AssignmentRule targets in the SBML model

it *may* include:

- Any SBML model parameter that was not excluded above
- Named parameter overrides introduced in the *conditions table*

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

Additional columns may be added.

1.7.1 Detailed field description:

- `parameterId` [STRING, NOT NULL]

  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`. Must be provided in linear space, independent of `parameterScale`.

- `upperBound` [NUMERIC]

  Upper bound of the parameter used for optimization. Optional, if `estimate==0`. Must be provided in linear space, independent of `parameterScale`.

- `nominalValue` [NUMERIC]

  Some parameter value to be used if the parameter is not subject to estimation (see `estimate` below). Must be provided in linear space, independent of `parameterScale`. 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`).

- `initializationPriorType` [STRING, OPTIONAL]

  Prior types used for sampling of initial points for optimization. Sampled points are clipped to lie inside the parameter boundaries specified by `lowerBound` and `upperBound`. Defaults to `parameterScaleUniform`.

  Possible prior types are:

  - *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

- `initializationPriorParameters` [STRING, OPTIONAL]

  Prior parameters used for sampling of initial points for optimization, separated by a semicolon. Defaults to `lowerBound;upperBound`.

  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

- `objectivePriorType` [STRING, OPTIONAL]

  Prior types used for the objective function during optimization or sampling. For possible values, see `initializationPriorType`.

- `objectivePriorParameters` [STRING, OPTIONAL]

  Prior parameters used for the objective function during optimization. For more detailed documentation, see `initializationPriorParameters`.

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 corresponding 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`.

1.8.2 Extensions

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

1.9 YAML file for grouping files

To link the SBML model, measurement table, condition table, etc. in an unambiguous way, we use a YAML file.

This file also allows specifying a PEtab version (as the format is not unlikely to change in the future).

Furthermore, this can be used to describe parameter estimation problems comprising multiple models (more details below).

The format is described in the schema ../petab/petab_schema.yaml, which allows for easy validation.

1.9.1 Parameter estimation problems combining multiple models

Parameter estimation problems can comprise multiple models. For now, PEtab allows to specify multiple SBML models with corresponding condition and measurement tables, and one joint parameter table. This means that the parameter namespace is global. Therefore, parameters with the same ID in different models will be considered identical.

CHAPTER 2

API Reference

petab	PEtab exports
petab.composite_problem	PEtab problems consisting of multiple models
petab.core	PEtab core functions (or functions that don't fit anywhere else)
petab.conditions	Functions operating on the PEtab condition table
petab.C	This file contains constant definitions.
petab.lint	Integrity checks and tests for specific features used
petab.measurements	Functions operating on the PEtab measurement table
petab.parameter_mapping	Functions related to mapping parameter from model to parameter estimation problem
petab.parameters	Functions operating on the PEtab parameter table
petab.problem	PEtab Problem class
petab.sampling	Functions related to parameter sampling
petab.sbml	Functions for interacting with SBML models
petab.yaml	Code regarding the PEtab YAML config files
petab.visualize.data_overview	Functions for creating an overview report of a PEtab problem
petab.visualize.helper_functions	This file should contain the functions, which PEtab internally needs for plotting, but which are not meant to be used by non-developers and should hence not be directly visible/usable when using *import petab.visualize*.
petab.visualize.plot_data_and_simulation(...)	Main function for plotting data and simulations.
petab.visualize.plotting_config	Plotting config

2.1 petab

PEtab exports

petab.**ENV_NUM_THREADS**

Name of environment variable to set number of threads or processes PEtab should use for operations that can be performed in parallel. By default, all operations are performed sequentially.

2.2 petab.composite_problem

PEtab problems consisting of multiple models

Classes

CompositeProblem(parameter_df, problems)	Representation of a PEtab problem consisting of multiple models

class petab.composite_problem.**CompositeProblem**(*parameter_df: pandas.core.frame.DataFrame = None*, *problems: List[petab.problem.Problem] = None*)

Bases: object

Representation of a PEtab problem consisting of multiple models

problems
List petab.Problems

parameter_df
PEtab parameter DataFrame

static from_yaml(*yaml_config: Union[Dict[KT, VT], str]*) → petab.composite_problem.CompositeProblem
Create from YAML file

Factory method to create a CompositeProblem instance from a PEtab YAML config file

Parameters **yaml_config** – PEtab configuration as dictionary or YAML file name

2.3 petab.core

PEtab core functions (or functions that don't fit anywhere else)

Functions

concat_tables(tables, ...)	Concatenate DataFrames provided as DataFrames or filenames, and a parser
flatten_timepoint_specific_output_overrides()	Flatten timepoint-specific output parameter overrides.
get_notnull_columns(df, candidates)	Return list of df-columns in candidates which are not all null/nan.
get_observable_id(parameter_id)	Get PEtab observable ID from PEtab-style sigma or observable *AssignmentRule*-target parameter_id.
get_simulation_df(simulation_file)	Read PEtab simulation table
get_visualization_df(visualization_file)	Read PEtab visualization table
to_float_if_float(x)	Return input as float if possible, otherwise return as is

Continued on next page

Table 3 – continued from previous page

write_simulation_df(df, filename)	Write PEtab simulation table
write_visualization_df(df, filename)	Write PEtab visualization table

petab.core.**concat_tables**(*tables: Union[str, pandas.core.frame.DataFrame, Iterable[Union[pandas.core.frame.DataFrame, str]]], file_parser: Optional[Callable] = None*) → pandas.core.frame.DataFrame

Concatenate DataFrames provided as DataFrames or filenames, and a parser

Parameters

- **tables** – Iterable of tables to join, as DataFrame or filename.
- **file_parser** – Function used to read the table in case filenames are provided, accepting a filename as only argument.

Returns The concatenated DataFrames

petab.core.**flatten_timepoint_specific_output_overrides**(*petab_problem: petab.problem.Problem*) → None

Flatten timepoint-specific output parameter overrides.

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. The observable table and measurement table are modified in place.

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

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`.

Parameters

- **df** – Dataframe
- **candidates** – Columns of `df` to consider

petab.core.**get_observable_id**(*parameter_id: str*) → str

Get PEtab observable ID from PEtab-style sigma or observable *AssignmentRule*-target `parameter_id`.

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

Parameters **parameter_id** – Some parameter ID

Returns Observable ID

petab.core.**get_simulation_df**(*simulation_file: str*) → pandas.core.frame.DataFrame

Read PEtab simulation table

Parameters **simulation_file** – URL or filename of PEtab simulation table

Returns Simulation DataFrame

petab.core.**get_visualization_df**(*visualization_file: str*) → pandas.core.frame.DataFrame

Read PEtab visualization table

Parameters **visualization_file** – URL or filename of PEtab visualization table

Returns Visualization DataFrame

petab.core.**to_float_if_float**(*x: Any*) → Any
Return input as float if possible, otherwise return as is

Parameters **x** – Anything

Returns x as float if possible, otherwise x

petab.core.**write_simulation_df**(*df: pandas.core.frame.DataFrame*, *filename: str*) → None
Write PEtab simulation table

Parameters

- **df** – PEtab simulation table
- **filename** – Destination file name

petab.core.**write_visualization_df**(*df: pandas.core.frame.DataFrame*, *filename: str*) → None
Write PEtab visualization table

Parameters

- **df** – PEtab visualization table
- **filename** – Destination file name

2.4 petab.conditions

Functions operating on the PEtab condition table

Functions

create_condition_df(parameter_ids, condition_ids)	Create empty condition DataFrame
get_condition_df(condition_file_name, ...)	Read the provided condition file into a pandas.Dataframe
get_parametric_overrides(condition_df)	Get parametric overrides from condition table
write_condition_df(df, filename)	Write PEtab condition table

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

Parameters

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

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

petab.conditions.**get_condition_df**(*condition_file_name: Union[str, pandas.core.frame.DataFrame, None]*) → pandas.core.frame.DataFrame
Read the provided condition file into a pandas.Dataframe

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

Parameters **condition_file_name** – File name of PEtab condition file

petab.conditions.**get_parametric_overrides**(*condition_df: pandas.core.frame.DataFrame*) → List[str]

Get parametric overrides from condition table

Parameters **condition_df** – PEtab condition table

Returns List of parameter IDs that are mapped in a condition-specific way

petab.conditions.**write_condition_df**(*df: pandas.core.frame.DataFrame*, *filename: str*) → None

Write PEtab condition table

Parameters

- **df** – PEtab condition table
- **filename** – Destination file name

2.5 petab.C

This file contains constant definitions.

2.6 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_defined(...)	Check if all observables in the measurement table have been defined in the observable table
assert_measurement_conditions_present_in_condition_table(...)	Ensure that all entries from measurement_df.simulationConditionId and measurement_df.preequilibrationConditionId are present in condition_df.index.
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(...)	Check that there is no trailing whitespace in elements of Iterable
assert_noise_distributions_valid(observable_df)	Ensure that noise distributions and transformations for observables are valid.
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.

Continued on next page

Table 5 – continued from previous page

assert_parameter_prior_type_is_valid(...)	Check that valid prior types have been selected
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)	Run sanity checks on PEtab condition table
check_measurement_df(df, observable_df)	Run sanity checks on PEtab measurement table
check_observable_df(observable_df)	Check validity of observable table
check_parameter_bounds(parameter_df)	Check if all entries in the lowerBound are smaller than upperBound column in the parameter table and that bounds are positive for parameterScale log|log10.
check_parameter_df(df, sbml_model, ...)	Run sanity checks on PEtab parameter table
condition_table_is_parameter_free(condition_df)	Check if all entries in the condition table are numeric (no parameter IDs)
is_valid_identifier(x)	Check whether *x* is a valid identifier
lint_problem(problem)	Run PEtab validation on problem
measurement_table_has_observable_parameter_numbers(...)	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.**_check_df**(*df: pandas.core.frame.DataFrame, req_cols: Iterable[T_co], name: str*) → None

Check if given columns are present in DataFrame

Parameters

- **df** – Dataframe to check
- **req_cols** – Column names which have to be present
- **name** – Name of the DataFrame to be included in error message

Raises `AssertionError` – if a column is missing

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*) → None

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

Parameters

- **parameter_df** – PEtab parameter DataFrame
- **sbml_model** – PEtab SBML Model
- **measurement_df** – PEtab measurement table
- **condition_df** – PEtab condition table

Raises `AssertionError` – in case of problems

petab.lint.**assert_measured_observables_defined**(*measurement_df: pandas.core.frame.DataFrame*, *observable_df: pandas.core.frame.DataFrame*) → None

Check if all observables in the measurement table have been defined in the observable table

Parameters

- **measurement_df** – PEtab measurement table
- **observable_df** – PEtab observable table

Raises AssertionError – in case of problems

petab.lint.**assert_measurement_conditions_present_in_condition_table**(*measurement_df: pandas.core.frame.DataFrame*, *condition_df: pandas.core.frame.DataFrame*) → None

Ensure that all entries from measurement_df.simulationConditionId and measurement_df.preequilibrationConditionId are present in condition_df.index.

Parameters

- **measurement_df** – PEtab measurement table
- **condition_df** – PEtab condition table

Raises AssertionError – in case of problems

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*) → None

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.

Parameters

- **parameter_df** – PEtab parameter DataFrame
- **sbml_model** – PEtab SBML Model
- **condition_df** – PEtab condition table

Raises AssertionError – in case of problems

petab.lint.**assert_no_leading_trailing_whitespace**(*names_list: Iterable[str], name: str*) → None

Check that there is no trailing whitespace in elements of Iterable

Parameters

- **names_list** – strings to check for whitespace
- **name** – name of *names_list* for error messages

Raises AssertionError – if there is trailing whitespace

petab.lint.**assert_noise_distributions_valid**(*observable_df: pandas.core.frame.DataFrame*) → None

Ensure that noise distributions and transformations for observables are valid.

Parameters **observable_df** – PEtab observable table

Raises AssertionError – in case of problems

petab.lint.**assert_parameter_bounds_are_numeric**(*parameter_df: pandas.core.frame.DataFrame*) → None

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

Parameters **parameter_df** – PEtab parameter DataFrame

Raises AssertionError – in case of problems

petab.lint.**assert_parameter_estimate_is_boolean**(*parameter_df: pandas.core.frame.DataFrame*) → None

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

Parameters **parameter_df** – PEtab parameter DataFrame

Raises AssertionError – in case of problems

petab.lint.**assert_parameter_id_is_string**(*parameter_df: pandas.core.frame.DataFrame*) → None

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

Parameters **parameter_df** – PEtab parameter DataFrame

Raises AssertionError – in case of problems

petab.lint.**assert_parameter_id_is_unique**(*parameter_df: pandas.core.frame.DataFrame*) → None

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

Parameters **parameter_df** – PEtab parameter DataFrame

Raises AssertionError – in case of problems

petab.lint.**assert_parameter_prior_type_is_valid**(*parameter_df: pandas.core.frame.DataFrame*) → None

Check that valid prior types have been selected

Parameters **parameter_df** – PEtab parameter table

Raises AssertionError in case of invalid prior

petab.lint.**assert_parameter_scale_is_valid**(*parameter_df: pandas.core.frame.DataFrame*) → None

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.

Parameters **parameter_df** – PEtab parameter DataFrame

Raises AssertionError – in case of problems

petab.lint.**check_condition_df**(*df: pandas.core.frame.DataFrame, sbml_model: Optional[libsbml.Model]*) → None

Run sanity checks on PEtab condition table

Parameters

- **df** – PEtab condition DataFrame
- **sbml_model** – SBML Model for additional checking of parameter IDs

Raises AssertionError – in case of problems

petab.lint.**check_measurement_df**(*df: pandas.core.frame.DataFrame*, *observable_df: Optional[pandas.core.frame.DataFrame] = None*) → None

Run sanity checks on PEtab measurement table

Parameters

- **df** – PEtab measurement DataFrame
- **observable_df** – PEtab observable DataFrame for checking if measurements are compatible with observable transformations.

Raises *AssertionError, ValueError* – in case of problems

petab.lint.**check_observable_df**(*observable_df: pandas.core.frame.DataFrame*) → None

Check validity of observable table

Parameters **observable_df** – PEtab observable DataFrame

Raises AssertionError – in case of problems

petab.lint.**check_parameter_bounds**(*parameter_df: pandas.core.frame.DataFrame*) → None

Check if all entries in the lowerBound are smaller than upperBound column in the parameter table and that bounds are positive for parameterScale log|log10.

Parameters **parameter_df** – PEtab parameter DataFrame

Raises AssertionError – in case of problems

petab.lint.**check_parameter_df**(*df: pandas.core.frame.DataFrame, sbml_model: Optional[libsbml.Model], measurement_df: Optional[pandas.core.frame.DataFrame], condition_df: Optional[pandas.core.frame.DataFrame]*) → None

Run sanity checks on PEtab parameter table

Parameters

- **df** – PEtab condition DataFrame
- **sbml_model** – SBML Model for additional checking of parameter IDs
- **measurement_df** – PEtab measurement table for additional checks
- **condition_df** – PEtab condition table for additional checks

Raises AssertionError – in case of problems

petab.lint.**condition_table_is_parameter_free**(*condition_df: pandas.core.frame.DataFrame*) → bool

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

Parameters **condition_df** – PEtab condition table

Returns True if there are no parameter overrides in the condition table, False otherwise.

petab.lint.**is_valid_identifier**(*x: str*) → bool

Check whether *x* is a valid identifier

Check whether *x* is a valid identifier for conditions, parameters, observables. . . . Identifiers may contain upper and lower case letters, digits and underscores, but must not start with a digit.

Parameters **x** – string to check

Returns `True` if valid, `False` otherwise

`petab.lint.`**`lint_problem`**(*problem: petab.problem.Problem*) → bool
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: pandas.core.frame.DataFrame*) → bool
Are there any numbers to override observable parameters?

Parameters **measurement_df** – PEtab measurement table

Returns True if there any numbers to override observable parameters, False otherwise.

`petab.lint.`**`measurement_table_has_timepoint_specific_mappings`**(*measurement_df: pandas.core.frame.DataFrame*) → bool
Are there time-point or replicate specific parameter assignments in the measurement table.

Parameters **measurement_df** – PEtab measurement table

Returns True if there are time-point or replicate specific parameter assignments in the measurement table, False otherwise.

2.7 petab.measurements

Functions operating on the PEtab measurement table

Functions

assert_overrides_match_parameter_count(...)	Ensure that number of parameters in the observable definition matches the number of overrides in `measurement_df`
create_measurement_df()	Create empty measurement dataframe
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_noise_distributions(measurement_df)	Returns dictionary of cost definitions per observable, if specified.
get_placeholders(formula_string, ...)	Get placeholder variables in noise or observable definition for the given observable ID.
get_rows_for_condition(measurement_df, ...)	Extract rows in *measurement_df* for *condition* according to 'preequilibrationConditionId' and 'simulationConditionId' in *condition*.
get_simulation_conditions(measurement_df)	Create a table of separate simulation conditions.

Continued on next page

Table 6 – continued from previous page

measurements_have_replicates(measurement_df)	Tests whether the measurements come with replicates
split_parameter_replacement_list(...)	Split values in observableParameters and noiseParameters in measurement table.
write_measurement_df(df, filename)	Write PEtab measurement table

petab.measurements.**assert_overrides_match_parameter_count**(*measurement_df: pandas.core.frame.DataFrame*, *observable_df: pandas.core.frame.DataFrame*) → None

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

Parameters

- **measurement_df** – PEtab measurement table
- **observable_df** – PEtab observable table

petab.measurements.**create_measurement_df**() → pandas.core.frame.DataFrame

Create empty measurement dataframe

Returns Created DataFrame

petab.measurements.**get_measurement_df**(*measurement_file_name: Union[None, str, pandas.core.frame.DataFrame]*) → pandas.core.frame.DataFrame

Read the provided measurement file into a `pandas.Dataframe`.

Parameters **measurement_file_name** – Name of file to read from

Returns Measurement DataFrame

petab.measurements.**get_measurement_parameter_ids**(*measurement_df: pandas.core.frame.DataFrame*) → List[str]

Return list of ID of parameters which occur in measurement table as observable or noise parameter overrides.

Parameters **measurement_df** – PEtab measurement DataFrame

Returns List of parameter IDs

petab.measurements.**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 Dictionary with *observableId => cost definition*

petab.measurements.**get_placeholders**(*formula_string: str*, *observable_id: str*, *override_type: str*) → List[str]

Get placeholder variables in noise or observable definition for the given observable ID.

Parameters

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

Returns List of placeholder parameter IDs in the order expected in the observableParameter column of the measurement table.

petab.measurements.**get_rows_for_condition**(*measurement_df: pandas.core.frame.DataFrame, condition: Union[pandas.core.frame.DataFrame, Dict[KT, VT]]*) → pandas.core.frame.DataFrame

Extract rows in *measurement_df* for *condition* according to 'preequilibrationConditionId' and 'simulationConditionId' in *condition*.

Parameters

- **measurement_df** – PEtab measurement DataFrame
- **condition** – DataFrame with single row and columns 'preequilibrationConditionId' and 'simulationConditionId'. Or dictionary with those keys.

Returns The subselection of rows in `measurement_df` for the condition `condition`.

petab.measurements.**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.measurements.**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.measurements.**split_parameter_replacement_list**(*list_string: Union[str, numbers.Number], delim: str = ';'*) → List[Union[str, float]]

Split values in observableParameters and noiseParameters in measurement table.

Parameters

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

Returns List of split values. Numeric values converted to float.

petab.measurements.**write_measurement_df**(*df: pandas.core.frame.DataFrame*, *filename: str*) → None

Write PEtab measurement table

Parameters

- **df** – PEtab measurement table
- **filename** – Destination file name

2.8 petab.parameter_mapping

Functions related to mapping parameter from model to parameter estimation problem

Functions

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(parameter_mappings, ...)	Merge preequilibration and simulation parameters and scales for a list of conditions while checking for compatibility.
merge_preeq_and_sim_pars_condition(...)	Merge preequilibration and simulation parameters and scales for a single condition while checking for compatibility.

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

Replace parameter IDs in parameter mapping dictionary by condition table parameter values (in-place).

Parameters

- **mapping** – see get_parameter_mapping_for_condition
- **condition_id** – ID of condition to work on
- **condition_df** – PEtab condition 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 as obtained from `get_parameter_mapping_for_condition`
- **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[str]*) → 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.**_apply_parameter_table**(*mapping: Dict[str, Union[str, numbers.Number]]*, *parameter_df: Optional[pandas.core.frame.DataFrame] = None*) → None

Replace parameters from parameter table in mapping list for a given condition.

Replace non-estimated parameters by `nominalValues` (un-scaled / lin-scaled), replace estimated parameters by the respective ID.

Parameters

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

petab.parameter_mapping.**_map_condition**(*packed_args*)

Helper function for parallel condition mapping.

For arguments see get_optimization_to_simulation_parameter_mapping

petab.parameter_mapping.**_map_condition_arg_packer**(*simulation_conditions*, *measurement_df*, *condition_df*, *parameter_df*, *simulation_parameters*, *warn_unmapped*)

Helper function to pack extra arguments for _map_condition

petab.parameter_mapping.**_output_parameters_to_nan**(*mapping: Dict[str, Union[str, numbers.Number]]*) → None

Set output parameters in mapping dictionary to nan

petab.parameter_mapping.**_perform_mapping_checks**(*measurement_df: pandas.core.frame.DataFrame*) → None

Check for PEtab features which we can't account for during parameter mapping.

petab.parameter_mapping.**get_optimization_to_simulation_parameter_mapping**(*condition_df: pandas.core.frame.DataFrame*, *measurement_df: pandas.core.frame.DataFrame*, *parameter_df: Optional[pandas.core.frame.*] = *None*, *observable_df: Optional[pandas.core.frame.*] = *None*, *sbml_model: Optional[libsbml.Model]* = *None*, *simulation_conditions: Optional[pandas.core.frame.*] = *None*, *warn_unmapped: Optional[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.

Mapping can be performed in parallel. The number of threads is controlled by the environment variable with the name of petab.ENV_NUM_THREADS.

Parameters

- **measurement_df, parameter_df, observable_df** (*condition_df,*) – The dataframes in the PEtab format.
- **sbml_model** – The sbml model with observables and noise specified according to the PEtab format.
- **simulation_conditions** – 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: Optional[pandas.core.frame.DataF = None*) → List[Tuple[Dict[str, str], Dict[str, str]]]

Get parameter scale mapping for all conditions

Parameters

- **parameter_df** – PEtab parameter DataFrame
- **mapping_par_opt_to_par_sim** – Parameter mapping as obtained from `get_optimization_to_simulation_parameter_mapping`
- **measurement_df** – PEtab measurement DataFrame
- **simulation_conditions** – Result of `petab.measurements.get_simulation_conditions` to avoid reevaluation.

Returns List of tuples with mapping dictionaries.

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: Optional[libsbml.Model] = None*, *simulation_parameters: Optional[Dict[str, str]] = 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
- **condition_df** – PEtab condition DataFrame
- **parameter_df** – PEtab parameter DataFrame
- **sbml_model** – The sbml model with observables and noise specified according to the PEtab format used to retrieve simulation parameter IDs. Mutually exclusive with `simulation_parameter_ids`.
- **simulation_parameters** – Model simulation parameter IDs mapped to parameter values (output of `petab.sbml.get_model_parameters(.., with_values=True)`). Mutually exclusive with `sbml_model`.
- **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`

Returns parameterId => parameterScale

Return type Mapping dictionary

`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 "(noiselobservable)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
- **condition_id** – Optional condition ID for more informative output

`petab.parameter_mapping.`**`merge_preeq_and_sim_pars`**(*parameter_mappings: Iterable[Tuple[Dict[str, Union[str, numbers.Number]], Dict[str, Union[str, numbers.Number]]]], scale_mappings: Iterable[Tuple[Dict[str, str], Dict[str, str]]]*) → Tuple[List[Tuple[Dict[str, Union[str, numbers.Number]], Dict[str, Union[str, numbers.Number]]]], List[Tuple[Dict[str, str], Dict[str, str]]]]

Merge preequilibration and simulation parameters and scales for a list of conditions while checking for compatibility.

Parameters

- **parameter_mappings** – As returned by petab.get_optimization_to_simulation_parameter_mapping
- **scale_mappings** – As returned by petab.get_optimization_to_simulation_scale_mapping.

Returns The parameter and scale simulation mappings, modified and checked.

`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 for a single condition 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

2.9 petab.parameters

Functions operating on the PEtab parameter table

Functions

create_parameter_df(sbml_model, ...)	Create a new PEtab parameter table
get_optimization_parameter_scaling(parameter_df)	Get Dictionary with optimization parameter IDs mapped to parameter scaling strings.
get_optimization_parameters(parameter_df)	Get list of optimization parameter IDs from parameter table.
get_parameter_df(parameter_file_name)	Read the provided parameter file into a `pandas.Dataframe`.
get_priors_from_df(parameter_df, mode)	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_valid_parameters_for_parameter_table()	Get set of parameters which may be present inside the parameter table
map_scale(parameters, scale_strs)	As scale(), but for Iterables
scale(parameter, scale_str)	Scale parameter according to scale_str
write_parameter_df(df, filename)	Write PEtab parameter table

petab.parameters.**create_parameter_df**(*sbml_model: libsbml.Model*, *condition_df: pandas.core.frame.DataFrame*, *measurement_df: pandas.core.frame.DataFrame*, *include_optional: bool = False*, *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** – SBML Model
- **condition_df** – PEtab condition DataFrame
- **measurement_df** – PEtab measurement DataFrame
- **include_optional** – By default this only returns parameters that are required to be

present in the parameter table. If set to True, this returns all parameters that are allowed to be present in the parameter table (i.e. also including parameters specified in the SBML model).

- **parameter_scale** – parameter scaling
- **lower_bound** – lower bound for parameter value
- **upper_bound** – upper bound for parameter value

Returns The created parameter DataFrame

petab.parameters.**get_optimization_parameter_scaling**(*parameter_df: pandas.core.frame.DataFrame*) → Dict[str, str]

Get Dictionary with optimization parameter IDs mapped to parameter scaling strings.

Parameters **parameter_df** – PEtab parameter DataFrame

Returns Dictionary with optimization parameter IDs mapped to parameter scaling strings.

petab.parameters.**get_optimization_parameters**(*parameter_df: pandas.core.frame.DataFrame*) → List[str]

Get list of optimization parameter IDs from parameter table.

Parameters **parameter_df** – PEtab parameter DataFrame

Returns List of IDs of parameters selected for optimization.

petab.parameters.**get_parameter_df**(*parameter_file_name: str*) → pandas.core.frame.DataFrame

Read the provided parameter file into a `pandas.Dataframe`.

Parameters **parameter_file_name** – Name of the file to read from.

Returns Parameter DataFrame

petab.parameters.**get_priors_from_df**(*parameter_df: pandas.core.frame.DataFrame*, *mode: str*) → List[Tuple]

Create list with information about the parameter priors

Parameters

- **parameter_df** – PEtab parameter table
- **mode** – 'initialization' or 'objective'

Returns List with prior information.

petab.parameters.**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

Parameters

- **sbml_model** – PEtab SBML model
- **condition_df** – PEtab condition table
- **measurement_df** – PEtab measurement table

Returns Set of parameter IDs which PEtab requires to be present in the parameter table. That is all {observable,noise}Parameters from the measurement table as well as all parametric condition table overrides that are not defined in the SBML model.

petab.parameters.**get_valid_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 may be present inside the parameter table

Parameters

- **sbml_model** – PEtab SBML model
- **condition_df** – PEtab condition table
- **measurement_df** – PEtab measurement table

Returns Set of parameter IDs which PEtab allows to be present in the parameter table.

petab.parameters.**map_scale**(*parameters: Iterable[numbers.Number], scale_strs: Iterable[str]*) → Iterable[numbers.Number]

As scale(), but for Iterables

petab.parameters.**scale**(*parameter: numbers.Number*, *scale_str: str*) → numbers.Number

Scale parameter according to scale_str

Parameters

- **parameter** – Parameter to be scaled
- **scale_str** – One of 'lin' (synonymous with ''), 'log', 'log10'

petab.parameters.**write_parameter_df**(*df: pandas.core.frame.DataFrame*, *filename: str*) → None

Write PEtab parameter table

Parameters

- **df** – PEtab parameter table
- **filename** – Destination file name

2.10 petab.problem

PEtab Problem class

Functions

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

Continued on next page

Table 9 – continued from previous page

get_default_sbml_file_name(model_name, folder)	Get file name according to proposed convention

Classes

Problem(sbml_model, sbml_reader, ...)	PEtab parameter estimation problem as defined by

class petab.problem.**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*, *visualization_df: pandas.core.frame.DataFrame = None*, *observable_df: pandas.core.frame.DataFrame = None*)

Bases: object

PEtab parameter estimation problem as defined by

- SBML model
- condition table
- measurement table
- parameter table
- observables table

Optionally it may contain visualization tables.

condition_df
PEtab condition table

measurement_df
PEtab measurement table

parameter_df
PEtab parameter table

observable_df
PEtab observable table

visualization_df
PEtab visualization table

sbml_reader
Stored to keep object alive.

sbml_document
Stored to keep object alive.

sbml_model
PEtab SBML 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: Union[str, Iterable[str]] = None*, *parameter_file: str = None*, *visualization_files: Union[str, Iterable[str]] = None*, *observable_files: Union[str, Iterable[str]] = None*) → petab.problem.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
- **visualization_files** – PEtab visualization tables
- **observable_files** – PEtab observables tables

static from_folder(*folder: str*, *model_name: str = None*) → petab.problem.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`.

static from_yaml(*yaml_config: Union[Dict[KT, VT], str]*) → petab.problem.Problem

Factory method to load model and tables as specified by YAML file.

Parameters **yaml_config** – PEtab configuration as dictionary or YAML file name

get_model_parameters()

See *petab.sbml.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_parameter_scales()

Return list of optimization parameter scaling strings.

See `petab.parameters.get_optimization_parameters`.

get_optimization_parameters()

Return list of optimization parameter IDs.

See `petab.parameters.get_optimization_parameters`.

get_optimization_to_simulation_parameter_mapping(*warn_unmapped: bool = True*)

See get_simulation_to_optimization_parameter_mapping.

get_optimization_to_simulation_scale_mapping(*mapping_par_opt_to_par_sim: List[Tuple[Dict[str, Union[str, numbers.Number]], Dict[str, Union[str, numbers.Number]]]]*) → List[Tuple[Dict[str, str], Dict[str, str]]]

See get_optimization_to_simulation_scale_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

lb_scaled

Parameter table lower bounds with applied parameter scaling

sample_parameter_startpoints(*n_starts: int = 100*)

Create starting points for optimization

See sample_parameter_startpoints

to_files(*sbml_file: Optional[str] = None*, *condition_file: Optional[str] = None*, *measurement_file: Optional[str] = None*, *parameter_file: Optional[str] = None*, *visualization_file: Optional[str] = None*, *observable_file: Optional[str] = None*) → None

Write PEtab tables to files for this problem

Writes PEtab files for those entities for which a destination was passed.

NOTE: If this instance was created from multiple measurement or visualization tables, they will be merged and written to a single file.

Parameters

- **sbml_file** – SBML model destination
- **condition_file** – Condition table destination
- **measurement_file** – Measurement table destination
- **parameter_file** – Parameter table destination
- **visualization_file** – Visualization table destination
- **observable_file** – Observables table destination

Raises

- `ValueError` – If a destination was provided for a non-existing
- `entity`.

ub

Parameter table upper bounds

ub_scaled

Parameter table upper bounds with applied parameter scaling

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

x_nominal_scaled
Parameter table nominal values with applied parameter scaling

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

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

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

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

2.11 petab.sampling

Functions related to parameter sampling

Functions

sample_from_prior(prior, list, str, list], ...)	Creates samples for one parameter based on prior
sample_parameter_startpoints(parameter_df, ...)	Create numpy.array with starting points for an optimization

petab.sampling.**sample_from_prior**(*prior: Tuple[str, list, str, list]*, *n_starts: int*) → numpy.array
Creates samples for one parameter based on prior

Parameters

- **prior** – A tuple as obtained from petab.parameter.get_priors_from_df
- **n_starts** – Number of samples

Returns Array with sampled values

petab.sampling.**sample_parameter_startpoints**(*parameter_df: pandas.core.frame.DataFrame*, *n_starts: int = 100*, *seed: int = None*) → numpy.array
Create numpy.array with starting points for an optimization

Parameters

- **parameter_df** – PEtab parameter DataFrame
- **n_starts** – Number of points to be sampled
- **seed** – Random number generator seed (see numpy.random.seed)

Returns Array of sampled starting points with dimensions n_startpoints x n_optimization_parameters

2.12 petab.sbml

Functions for interacting with 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.
create_assigment_rule(sbml_model, ...)	Create SBML AssignmentRule
get_model_parameters(sbml_model[, with_values])	Return SBML model parameters which are not AssignmentRule targets for observables or sigmas
get_observables(sbml_model, remove)	Get observables defined in SBML model according to PEtab format.
get_sigmas(sbml_model, remove)	Get sigmas defined in SBML model according to PEtab format.
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
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.
write_sbml(sbml_doc, filename)	Write PEtab visualization table

`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

Parameters

- **`sbml_model`** – SBML model
- **`parameter_id`** – ID of the new parameter
- **`parameter_name`** – Name of the new parameter
- **`constant`** – Is parameter constant?
- **`units`** – SBML unit ID

- **value** – parameter value

Returns The created parameter

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*) → None

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*) → 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.**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

Returns The created `AssignmentRule`

petab.sbml.**get_model_parameters**(*sbml_model: libsbml.Model*, *with_values=False*) → Union[List[str], Dict[str, float]]

Return SBML model parameters which are not AssignmentRule targets for observables or sigmas

Parameters

- **sbml_model** – SBML model
- **with_values** – If false, returns list of SBML model parameter IDs which
- **not AssignmentRule targets for observables or sigmas. If true,** (*are*) –
- **a dictionary with those parameter IDs as keys and parameter** (*returns*) –
- **from the SBML model as values.** (*values*) –

petab.sbml.**get_observables**(*sbml_model: libsbml.Model*, *remove: bool = False*) → dict

Get observables defined in SBML model according to PEtab format.

Returns Dictionary of observable definitions. See *assignment_rules_to_dict* for details.

petab.sbml.**get_sigmas**(*sbml_model: libsbml.Model*, *remove: bool = False*) → dict

Get sigmas defined in SBML model according to PEtab format.

Returns

Dictionary of sigma definitions.

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

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*) → bool

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)

petab.sbml.**sbml_parameter_is_observable**(*sbml_parameter: libsbml.Parameter*) → bool

Returns whether the `libsbml.Parameter` `sbml_parameter` matches the defined observable format.

petab.sbml.**sbml_parameter_is_sigma**(*sbml_parameter: libsbml.Parameter*) → bool

Returns whether the `libsbml.Parameter` `sbml_parameter` matches the defined sigma format.

petab.sbml.**write_sbml**(*sbml_doc: libsbml.SBMLDocument*, *filename: str*) → None

Write PEtab visualization table

Parameters

- **sbml_doc** – SBML document containing the SBML model
- **filename** – Destination file name

2.13 petab.yaml

Code regarding the PEtab YAML config files

Functions

add_constructor
add_implicit_resolver
add_multi_constructor
add_multi_representer
add_path_resolver
add_representer

Continued on next page

Table 13 – continued from previous page

compose	
compose_all	
dump	
dump_all	
emit	
full_load	
full_load_all	
load	
load_all	
load_warning	
parse	
safe_dump	
safe_dump_all	
safe_load	
safe_load_all	
scan	
serialize	
serialize_all	
unsafe_load	
unsafe_load_all	
warnings	Python part of the warnings subsystem.

Classes

YAMLObject
YAMLObjectMetaclass

Exceptions

YAMLLoadWarning

petab.yaml.**assert_single_condition_and_sbml_file**(*problem_config: Dict[KT, VT]*) → None

Check that there is only a single condition file and a single SBML file specified.

Parameters **problem_config** – Dictionary as defined in the YAML schema inside the *problems* list.

Raises NotImplementedError – If multiple condition or SBML files specified.

petab.yaml.**is_composite_problem**(*yaml_config: Union[Dict[KT, VT], str]*) → bool

Does this YAML file comprise multiple models?

Parameters **yaml_config** – PEtab configuration as dictionary or YAML file name

petab.yaml.**load_yaml**(*yaml_config: Union[Dict[KT, VT], str]*) → Dict[KT, VT]

Load YAML

Convenience function to allow for providing YAML inputs either as filename or as dictionary.

Parameters **yaml_config** – PEtab YAML config as filename or dict.

Returns The unmodified dictionary if yaml_config was dictionary. Otherwise the parsed the YAML file.

petab.yaml.**validate**(*yaml_config: Union[Dict[KT, VT], str], path_prefix: Optional[str] = None*)
Validate syntax and semantics of PEtab config YAML

Parameters

- **yaml_config** – PEtab YAML config as filename or dict.
- **path_prefix** – Base location for relative paths. Defaults to location of YAML file if a filename was provided for `yaml_config` or the current working directory.

petab.yaml.**validate_yaml_semantics**(*yaml_config: Union[Dict[KT, VT], str], path_prefix: Optional[str] = None*)
Validate PEtab YAML file semantics

Check for existence of files. Assumes valid syntax.

Version number and contents of referenced files are not yet checked.

Parameters

- **yaml_config** – PEtab YAML config as filename or dict.
- **path_prefix** – Base location for relative paths. Defaults to location of YAML file if a filename was provided for `yaml_config` or the current working directory.

Raises `AssertionError` – in case of problems

petab.yaml.**validate_yaml_syntax**(*yaml_config: Union[Dict[KT, VT], str], schema: Union[None, Dict[KT, VT], str] = None*)
Validate PEtab YAML file syntax

Parameters

- **yaml_config** – PEtab YAML file to validate, as file name or dictionary
- **schema** – Custom schema for validation

Raises see jsonschema.validate

2.14 petab.visualize.data_overview

Functions for creating an overview report of a PEtab problem

Functions

create_report(problem, model_name)	Create an HTML overview data / model overview report
get_data_per_observable(measurement_df)	Get table with number of data points per observable and condition
main()	Data overview generation with example data from the repository for testing

petab.visualize.data_overview.**create_report**(*problem: petab.problem.Problem, model_name: str*) → None
Create an HTML overview data / model overview report

Parameters

- **problem** – PEtab problem
- **model_name** – Name of the model, used for file name for report

petab.visualize.data_overview.**get_data_per_observable**(*measurement_df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Get table with number of data points per observable and condition

Parameters **measurement_df** – PEtab measurement data frame

petab.visualize.data_overview.**main**()

Data overview generation with example data from the repository for testing

2.15 petab.visualize.helper_functions

This file should contain the functions, which PEtab internally needs for plotting, but which are not meant to be used by non-developers and should hence not be directly visible/usable when using *import petab.visualize*.

Functions

check_vis_spec_consistency(dataset_id_list, ...)	Helper function for plotting data and simulations, which check the visualization setting, if no visualization specification file is provided.
create_dataset_id_list(simcond_id_list, ...)	Create dataset id list
create_figure(uni_plot_ids, plots_to_file)	Helper function for plotting data and simulations, open figure and axes
get_data_to_plot(vis_spec, m_data, ...)	group the data, which should be plotted and return it as dataframe.
get_default_vis_specs(exp_data, exp_conditions)	Helper function for plotting data and simulations, which creates a default visualization table.
handle_dataset_plot(i_visu_spec, ind_plot, ...)	Handle dataset plot
import_from_files(data_file_path, ...)	Helper function for plotting data and simulations, which imports data from PEtab files.

petab.visualize.helper_functions.**check_vis_spec_consistency**(*dataset_id_list*, *sim_cond_id_list*, *sim_cond_num_list*, *observable_id_list*, *observable_num_list*, *exp_data*)

Helper function for plotting data and simulations, which check the visualization setting, if no visualization specification file is provided.

For documentation, see main function plot_data_and_simulation()

petab.visualize.helper_functions.**create_dataset_id_list**(*simcond_id_list*, *sim_cond_num_list*, *observable_id_list*, *observable_num_list*, *exp_data*, *exp_conditions*, *group_by*)

Create dataset id list

petab.visualize.helper_functions.**create_figure**(*uni_plot_ids*, *plots_to_file*)

Helper function for plotting data and simulations, open figure and axes

Parameters

- **uni_plot_ids** (*ndarray*) – Array with unique plot indices
- **plots_to_file** (*bool*) – Indicator if plots are saved to file

Returns

- **fig** (*Figure object of the created plot.*)
- **ax** (*Axis object of the created plot.*)
- **num_row** (*int, number of subplot rows*)
- **num_col** (*int, number of subplot columns*)

petab.visualize.helper_functions.**get_data_to_plot**(*vis_spec: pandas.core.frame.DataFrame*, *m_data: pandas.core.frame.DataFrame*, *simulation_data: pandas.core.frame.DataFrame*, *condition_ids: numpy.ndarray*, *i_visu_spec: int*, *col_id: str*, *simulation_field: str = 'simulatedData'*) → pandas.core.frame.DataFrame

group the data, which should be plotted and return it as dataframe.

Parameters

- **vis_spec** – pandas data frame, contains defined data format (visualization file)
- **m_data** – pandas data frame, contains defined data format (measurement file)
- **simulation_data** – pandas data frame, contains defined data format (simulation file)
- **condition_ids** – numpy array, containing all unique condition IDs which should be plotted in one figure (can be found in measurementData file, column simulationConditionId)
- **i_visu_spec** – int, current index (row number) of row which should be plotted in visualizationSpecification file
- **col_id** – str, the name of the column in visualization file, whose entries should be unique (depends on condition in column independentVariableName)
- **simulation_field** – Column name in `simulation_data` that contains the actual simulation result.

Returns pandas.DataFrame containing the data which should be plotted (Mean and Std)

Return type data_to_plot

petab.visualize.helper_functions.**get_default_vis_specs**(*exp_data*, *exp_conditions*, *dataset_id_list=None*, *sim_cond_id_list=None*, *sim_cond_num_list=None*, *observable_id_list=None*, *observable_num_list=None*, *plotted_noise='MeanAndSD'*)

Helper function for plotting data and simulations, which creates a default visualization table.

For documentation, see main function plot_data_and_simulation()

petab.visualize.helper_functions.**handle_dataset_plot**(*i_visu_spec*, *ind_plot*, *ax*, *i_row*, *i_col*, *exp_data*, *exp_conditions*, *vis_spec*, *sim_data*)

Handle dataset plot

petab.visualize.helper_functions.**import_from_files**(*data_file_path*, *condition_file_path*, *visualization_file_path*, *simulation_file_path*, *dataset_id_list*, *sim_cond_id_list*, *sim_cond_num_list*, *observable_id_list*, *observable_num_list*, *plotted_noise*)

Helper function for plotting data and simulations, which imports data from PEtab files.

For documentation, see main function plot_data_and_simulation()

2.16 petab.visualize.plot_data_and_simulation

petab.visualize.**plot_data_and_simulation**(*exp_data: Union[str, pandas.core.frame.DataFrame]*, *exp_conditions: Union[str, pandas.core.frame.DataFrame]*, *vis_spec: Union[str, pandas.core.frame.DataFrame, None] = ''*, *sim_data: Union[str, pandas.core.frame.DataFrame, None] = None*, *dataset_id_list: Optional[List[List[str]]] = None*, *sim_cond_id_list: Optional[List[List[str]]] = None*, *sim_cond_num_list: Optional[List[List[int]]] = None*, *observable_id_list: Optional[List[List[str]]] = None*, *observable_num_list: Optional[List[List[int]]] = None*, *plotted_noise: Optional[str] = 'MeanAndSD'*, *subplot_file_path: str = ''*)

Main function for plotting data and simulations.

What exactly should be plotted is specified in a visualizationSpecification.tsv file.

Also, the data, simulations and conditions have to be defined in a specific format (see "doc/documentation_data_format.md").

Parameters

- **exp_data** – measurement DataFrame in the PEtab format or path to the data file.
- **exp_conditions** – condition DataFrame in the PEtab format or path to the condition file.
- **vis_spec**(*str or pandas.Dataframe (optional)*) –

  Visualization specification DataFrame in the PEtab format or path to visualization file.
- **sim_data** – simulation DataFrame in the PEtab format or path to the simulation output data file.

- **dataset_id_list** – A list of lists. Each sublist corresponds to a plot, each subplot contains the datasetIds for this plot. Only to be used if no visualization file was available.
- **sim_cond_id_list** – A list of lists. Each sublist corresponds to a plot, each subplot contains the simulationConditionIds for this plot. Only to be used if no visualization file was available.
- **sim_cond_num_list** – A list of lists. Each sublist corresponds to a plot, each subplot contains the numbers corresponding to the simulationConditionIds for this plot. Only to be used if no visualization file was available.
- **observable_id_list** – A list of lists. Each sublist corresponds to a plot, each subplot contains the observableIds for this plot. Only to be used if no visualization file was available.
- **observable_num_list** – A list of lists. Each sublist corresponds to a plot, each subplot contains the numbers corresponding to the observableIds for this plot. Only to be used if no visualization file was available.
- **plotted_noise** – String indicating how noise should be visualized: ['MeanAndSD' (default), 'MeanAndSEM', 'replicate', 'provided']
- **subplot_file_path** – String which is taken as file path to which single subplots are saved. PlotIDs will be taken as file names.

Returns

- **ax** (*Axis object of the created plot.*)
- **None** (*In case subplots are save to file*)

2.17 petab.visualize.plotting_config

Plotting config

Functions

plot_lowlevel(vis_spec, ax, axx, axy, ...)	plotting routine / preparations: set properties of figure and plot the data with given specifications (lineplot with errorbars, or barplot)

petab.visualize.plotting_config.**plot_lowlevel**(*vis_spec: pandas.core.frame.DataFrame*, *ax: numpy.ndarray*, *axx: int*, *axy: int*, *conditions: pandas.core.series.Series*, *ms: pandas.core.frame.DataFrame*, *ind_plot: pandas.core.series.Series*, *i_visu_spec: int*, *plot_sim: bool*)

plotting routine / preparations: set properties of figure and plot the data with given specifications (lineplot with errorbars, or barplot)

Parameters

- **vis_spec** – pandas data frame, contains defined data format (visualization file)
- **ax** – np.ndarray, matplotlib.Axes
- **axx** – int, subplot axis indices for x
- **axy** – int, subplot axis indices for y

- **conditions** – pd.Series, Values on x-axis
- **ms** – pd.DataFrame, containing measurement data which should be plotted
- **ind_plot** – pd.Series, boolean vector, with size: len(rows in visualization file) x 1 with 'True' entries for rows which should be plotted
- **i_visu_spec** – int64, current index (row number) of row which should be plotted in visualizationSpecification file
- **plot_sim** – bool, tells whether or not simulated data should be plotted as well

Returns ax

Return type matplotlib.Axes

CHAPTER 3

PEtab changelog

3.1 0.1.1

Library:

- Fix parameter mapping: include output parameters not present in SBML model
- Fix missing `petab/petab_schema.yaml` in source distribution
- Let get_placeholders return an (ordered) list of placeholders
- Deprecate `petab.problem.from_folder` and related functions (obsolete after introducing more flexible YAML files for grouping tables and models)

3.2 0.1.0

Data format:

- Introduce observables table instead of SBML assignment rules for defining observation model (#244) (moves observableTransformation and noiseModel from the measurement table to the observables table)
- Allow initial concentrations / sizes in condition table (#238)
- Fixes and clarifications in the format documentation
- Changes in prior columns of the parameter table (#222)
- Introduced separate version number of file format, this release being version 1

Library:

- Adaptations to new file formats
- Various bugfixes and clean-up, especially in visualization and validator
- Parameter mapping changed to include all model parameters and not only those differing from the ones defined inside the SBML model

- Introduced constants for all field names and string options, replacing most string literals in the code (#228)
- Added unit tests and additional format validation steps
- Optional parallelization of parameter mapping (#205)
- Extended documentation (in-source and example Jupyter notebooks)

3.3 0.0.2

Bugfix release

- Fix `petablint` error
- Fix minor issues in `petab.visualize`

3.4 0.0.1

Data format:

- Update format and documentation with respect to data and parameter scales (#169)
- Define YAML schema for grouping PEtab files, also allowing for more complex combinations of files (#183)

Library:

- Refactor library. Reorganize `petab.core` functions.
- Fix visualization w/o condition names #142
- Extend validator
- Removed deprecated functions petab.Problem.get_constant_parameters and petab.sbml.constant_species_to_parameters
- Minor fixes and extensions

3.5 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

3.6 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.

CHAPTER 6

Examples

The following examples should help to get a better idea of how to use the PEtab library.

6.1 Using petablint

`petablint` is a tool to validate a model against the PEtab standard. When you have installed PEtab, you can simply call it from the command line. It takes the following arguments:

[1]:
```
!petablint -h
```

```
usage: petablint [-h] [-v] [-s SBML_FILE_NAME] [-m MEASUREMENT_FILE_NAME]
                 [-c CONDITION_FILE_NAME] [-p PARAMETER_FILE_NAME]
                 [-y YAML_FILE_NAME | -n MODEL_NAME] [-d DIRECTORY]

Check if a set of files adheres to the PEtab format.

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbose         More verbose output
  -s SBML_FILE_NAME, --sbml SBML_FILE_NAME
                        SBML model filename
  -m MEASUREMENT_FILE_NAME, --measurements MEASUREMENT_FILE_NAME
                        Measurement table
  -c CONDITION_FILE_NAME, --conditions CONDITION_FILE_NAME
                        Conditions table
  -p PARAMETER_FILE_NAME, --parameters PARAMETER_FILE_NAME
                        Parameter table
  -y YAML_FILE_NAME, --yaml YAML_FILE_NAME
                        PEtab YAML problem filename
  -n MODEL_NAME, --model-name MODEL_NAME
                        Model name where all files are in the working
                        directory and follow PEtab naming convention.
                        Specifying -[smcp] will override defaults
  -d DIRECTORY, --directory DIRECTORY
```

Let's look at an example: In the example_Fujita folder, we have a PEtab configuration file `Fujita.yaml` telling which files belong to the Fujita model:

[2]:
```
!cat example_Fujita/Fujita.yaml
```

```
parameter_file: Fujita_parameters.tsv
petab_version: 0.0.0a17
problems:
- condition_files:
  - Fujita_experimentalCondition.tsv
  measurement_files:
  - Fujita_measurementData.tsv
  sbml_files:
  - Fujita_model.xml
```

To verify everything is ok, we can just call:

[3]:
```
!petablint -y example_Fujita/Fujita.yaml
```

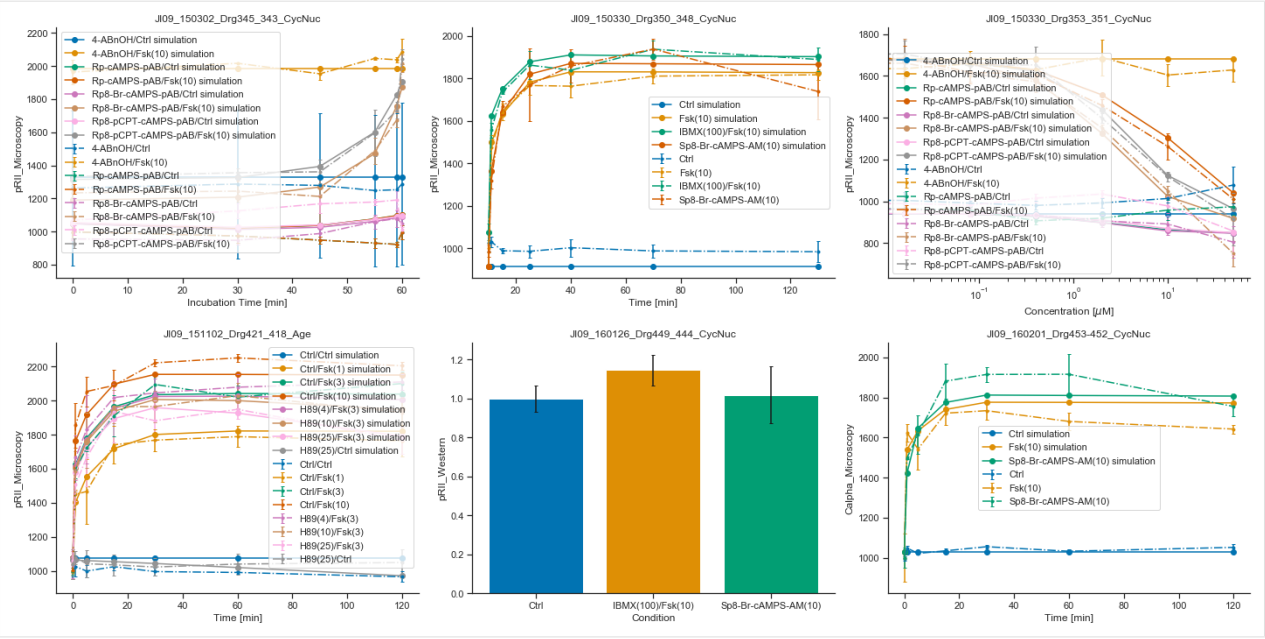
If there were some inconsistency or error, we would see that here. `petablint` can be called in different ways. You can e.g. also pass SBML, measurement, condition, and parameter file directly, or, if the files follow PEtab naming conventions, you can just pass the model name.

6.2 Visualization of data and simulations

In this notebook, we illustrate the visualization functions of petab.

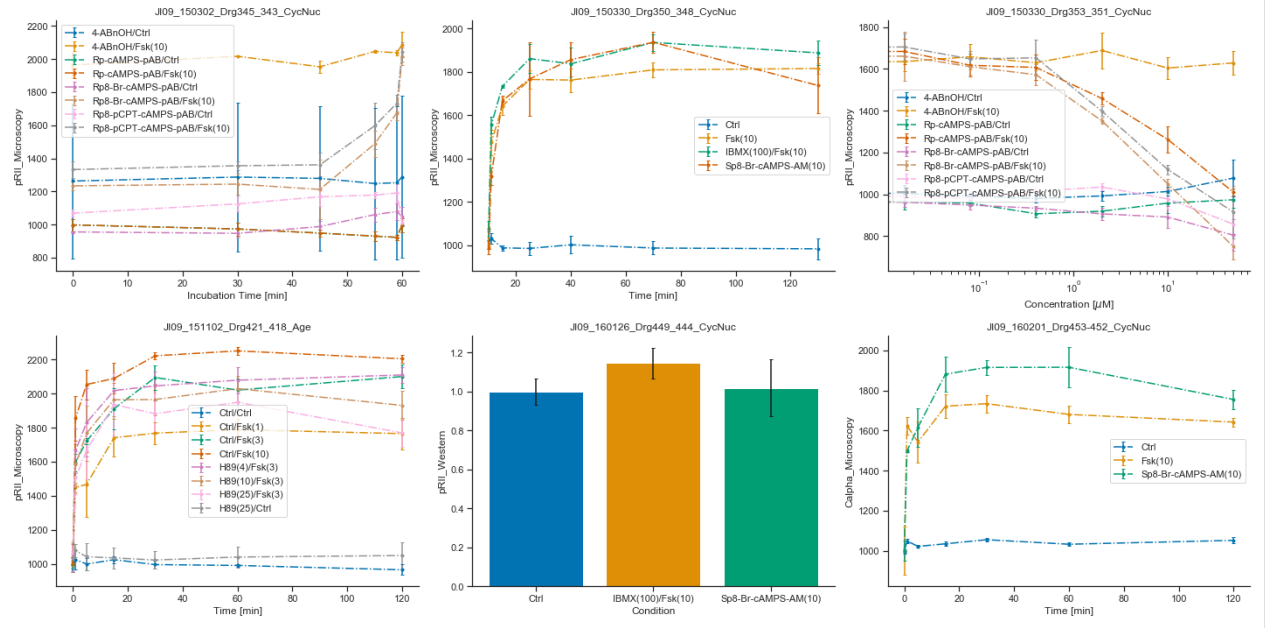
[1]:
```
from petab.visualize import plot_data_and_simulation
import matplotlib.pyplot as plt
```

[2]:
```
folder = "example_Isensee/"

data_file_path = folder + "Isensee_measurementData.tsv"
condition_file_path = folder + "Isensee_experimentalCondition.tsv"
visualization_file_path = folder + "Isensee_visualizationSpecification.tsv"
simulation_file_path = folder + "Isensee_simulationData.tsv"

# function to call, to plot data and simulations
ax = plot_data_and_simulation(data_file_path,
                              condition_file_path,
                              visualization_file_path,
                              simulation_file_path)
plt.show()
```



Now, we want to call the plotting routines without using the simulated data, only the visualization specification file.

[3]:
```
ax_without_sim = plot_data_and_simulation(
    data_file_path,
    condition_file_path,
    visualization_file_path)
plt.show()
```



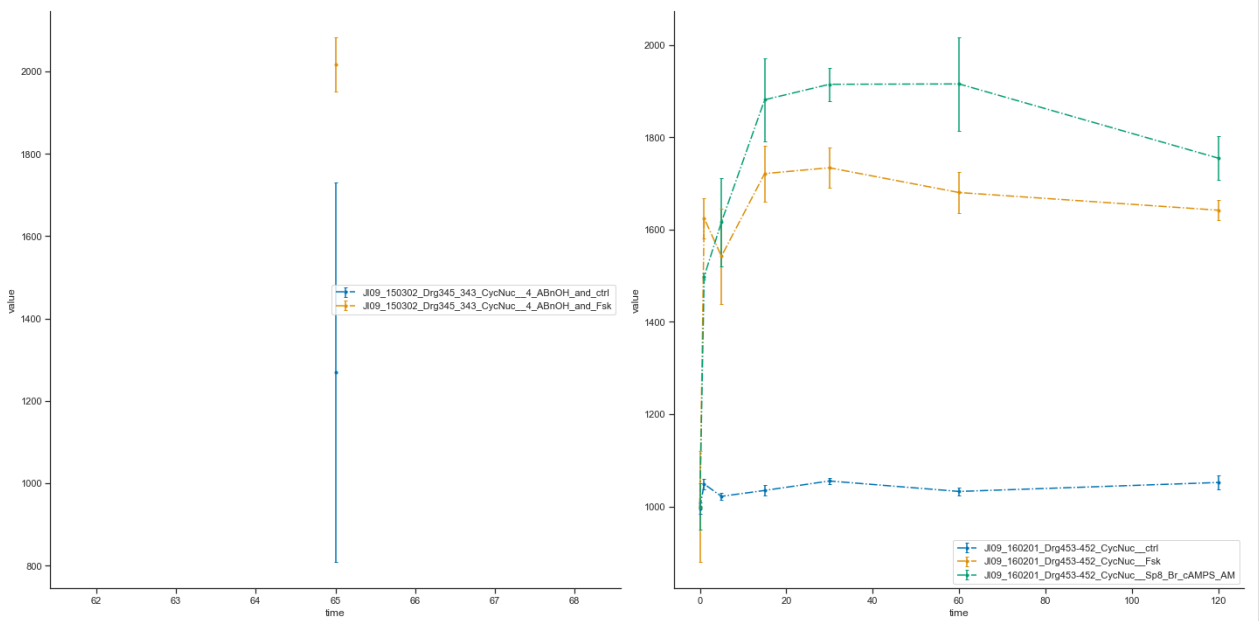
We can also call the plotting routine without the visualization specification file, but by passing a list of lists as `dataset_id_list`. Each sublist corresponds to a plot, and contains the datasetIds which should be plotted. In this simply structured plotting routine, the independent variable will always be time.

[4]:
```
ax_without_sim = plot_data_and_simulation(
```

(continues on next page)

(continued from previous page)

```
    data_file_path,
    condition_file_path,
    dataset_id_list = [['JI09_150302_Drg345_343_CycNuc__4_ABnOH_and_ctrl',
                        'JI09_150302_Drg345_343_CycNuc__4_ABnOH_and_Fsk'],
                       ['JI09_160201_Drg453-452_CycNuc__ctrl',
                        'JI09_160201_Drg453-452_CycNuc__Fsk',
                        'JI09_160201_Drg453-452_CycNuc__Sp8_Br_cAMPS_AM']])
plt.show()
```

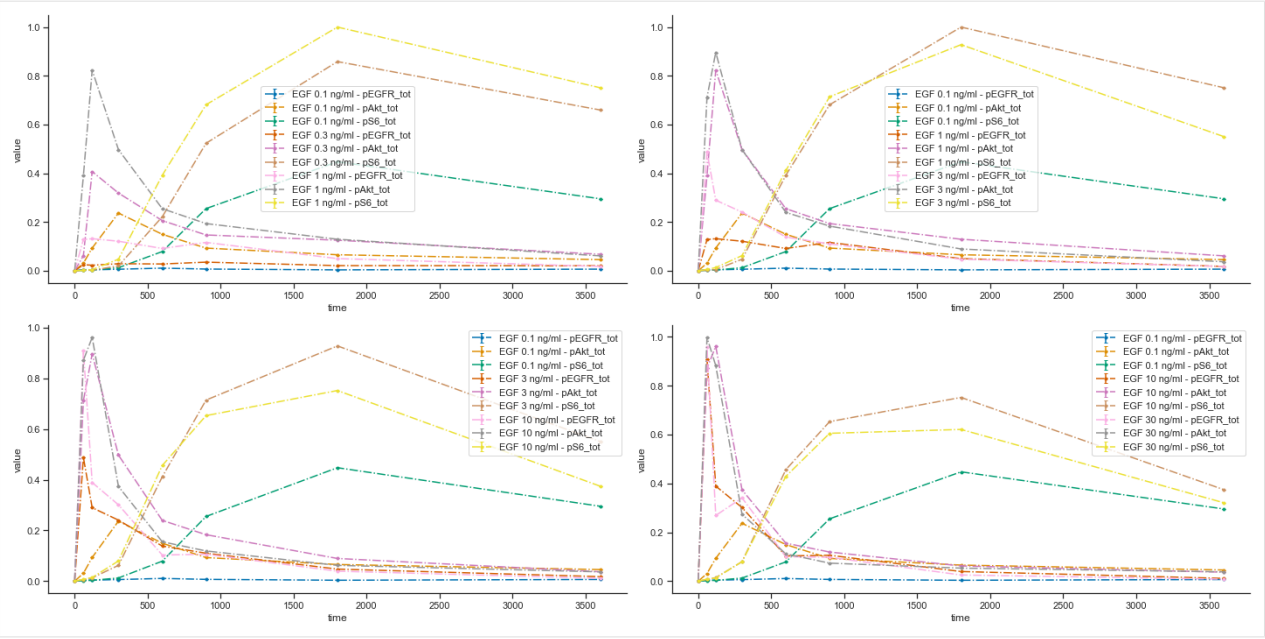


Let's look more closely at the plotting routines, if no visualization specification file is provided. If such a file is missing, PEtab needs to know how to group the data points. For this, five options can be used: * dataset_id_list * sim_cond_id_lis * sim_cond_num_list * observable_id_list * observable_num_list

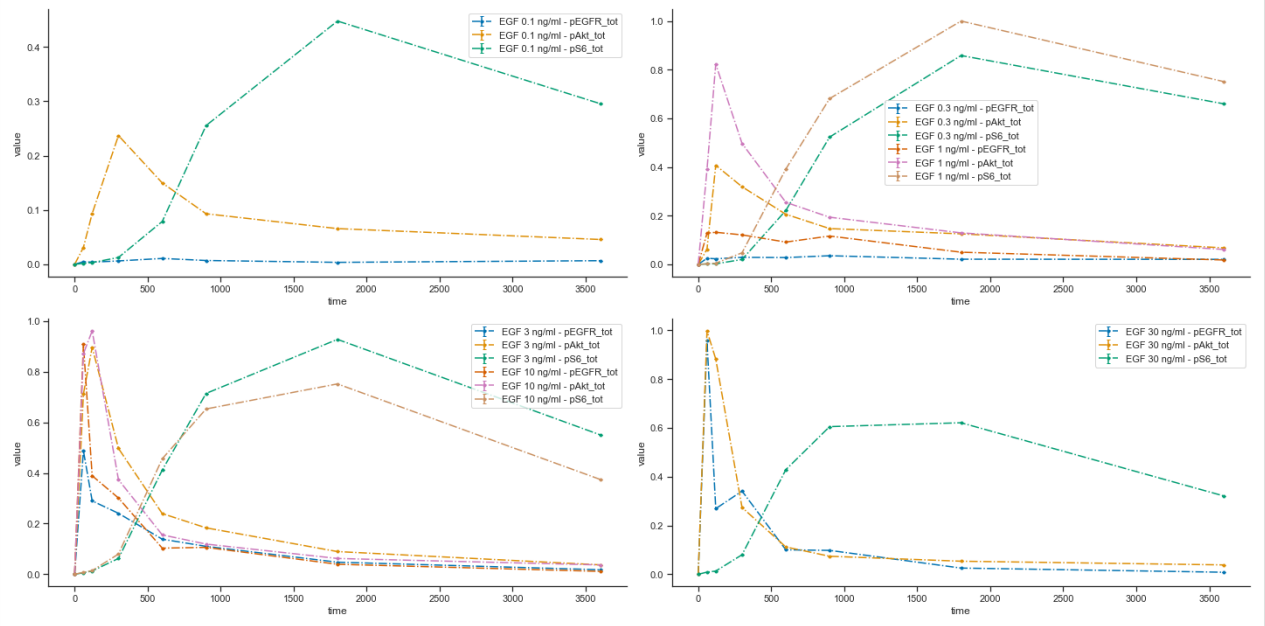
Each of them is a list of lists. Again, each sublist is a plot and its content are either simulation condition IDs or observable IDs (or their corresponding number when being enumerated) or the dataset IDs.

We want to illustrate this functionality by using a simpler example, a model published in 2010 by Fujita et al.

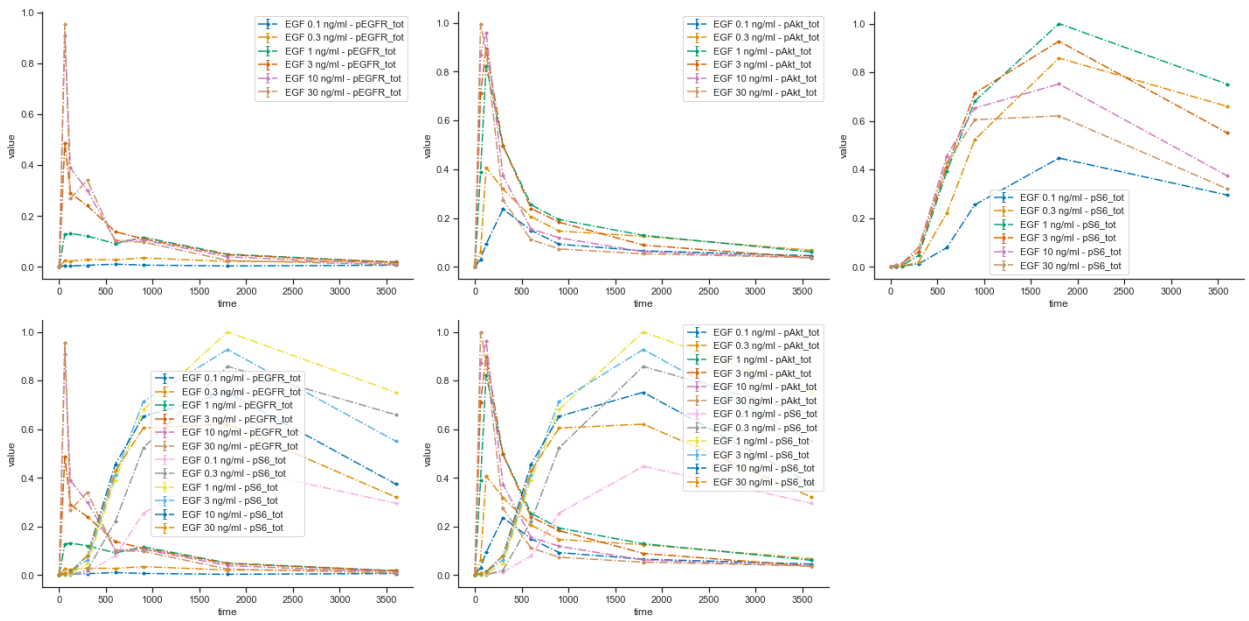
[5]:
```
data_file_path = "example_Fujita/Fujita_measurementData.tsv"
condition_file_path = "example_Fujita/Fujita_experimentalCondition.tsv"

# Plot 4 axes objects, plotting
# - in the first window all observables of the  1st, 2nd, and 3rd simulation condition
# - in the second window all observables of the 1st, 3rd, and 4th simulation condition
# - in the third window all observables of the  1st, 4th, and 5th simulation condition
# - in the fourth window all observables of the 1st, 5th, and 6th simulation condition
plot_data_and_simulation(data_file_path, condition_file_path,
                         sim_cond_num_list = [[0, 1, 2], [0, 2, 3], [0, 3, 4], [0, 4,␣
↪5]])
plt.show()
```



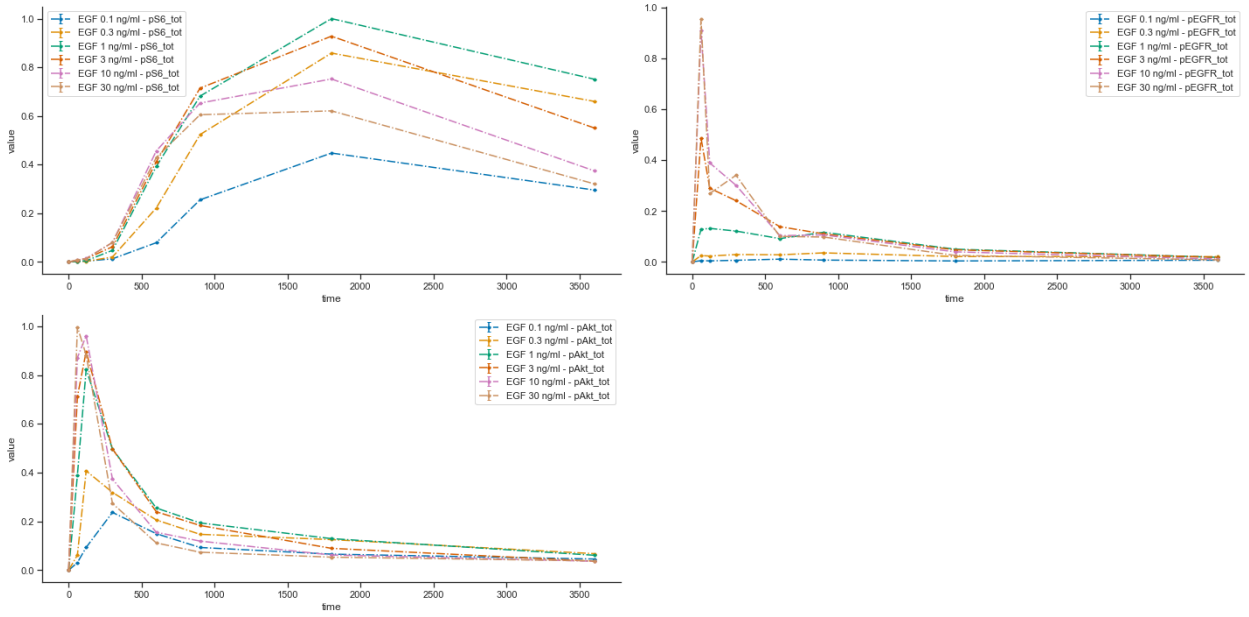
[6]:
```
# Plot 4 axes objects, plotting
# - in the first window  all observables of the simulation condition 'model1_data1'
# - in the second window all observables of the simulation conditions 'model1_data2',
↪'model1_data3'
# - in the third window  all observables of the simulation conditions 'model1_data4',
↪'model1_data5'
# - in the fourth window all observables of the simulation condition 'model1_data6'
plot_data_and_simulation(
    data_file_path, condition_file_path,
    sim_cond_id_list = [['model1_data1'], ['model1_data2', 'model1_data3'],
                        ['model1_data4', 'model1_data5'], ['model1_data6']])
plt.show()
```



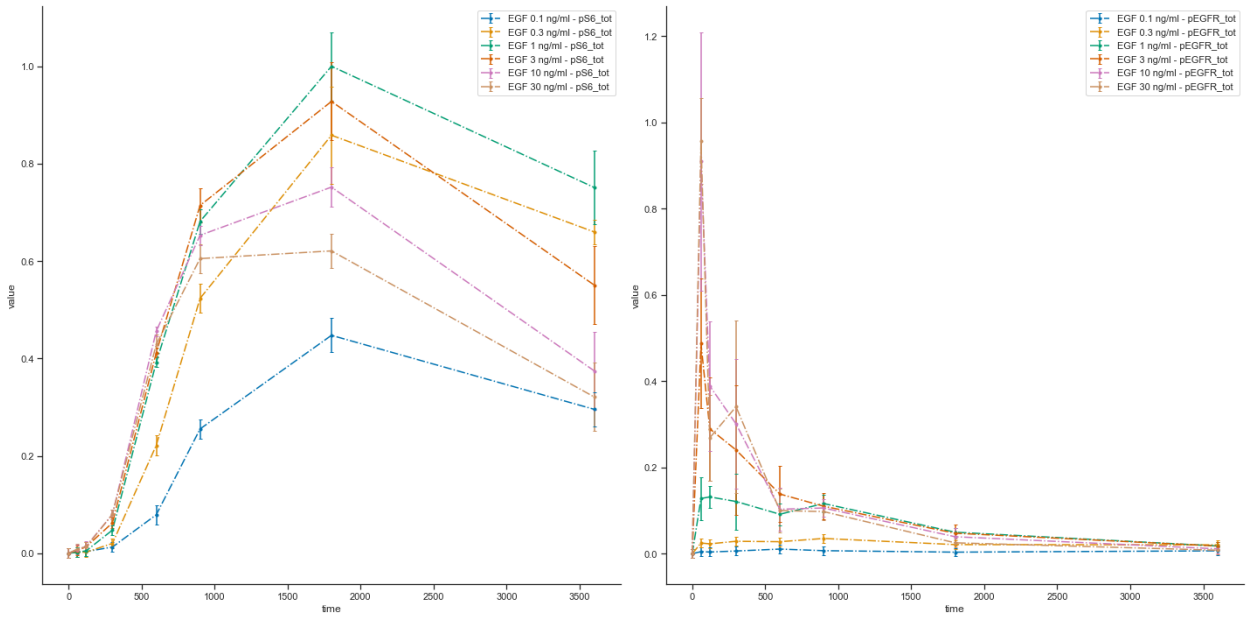
[7]:
```
# Plot 5 axes objects, plotting
# - in the first window  the 1st observable for all simulation conditions
# - in the second window the 2nd observable for all simulation conditions
# - in the third window  the 3rd observable for all simulation conditions
# - in the fourth window the 1st and 3rd observable for all simulation conditions
# - in the fifth window  the 2nd and 3rd observable for all simulation conditions
plot_data_and_simulation(
    data_file_path, condition_file_path,
    observable_num_list = [[0], [1], [2], [0, 2], [1, 2]])
plt.show()
```



[8]:
```
# Plot 3 axes objects, plotting
# - in the first window  the observable 'pS6_tot' for all simulation conditions
# - in the second window the observable 'pEGFR_tot' for all simulation conditions
# - in the third window  the observable 'pAkt_tot' for all simulation conditions

plot_data_and_simulation(
    data_file_path, condition_file_path,
    observable_id_list = [['pS6_tot'], ['pEGFR_tot'], ['pAkt_tot']])
plt.show()
```



[9]:
```
# Plot 2 axes objects, plotting
# - in the first window  the observable 'pS6_tot' for all simulation conditions
# - in the second window the observable 'pEGFR_tot' for all simulation conditions
# - in the third window  the observable 'pAkt_tot' for all simulation conditions
# while using the noise values which are saved in the PEtab files

plot_data_and_simulation(
    data_file_path, condition_file_path,
    observable_id_list = [['pS6_tot'], ['pEGFR_tot']],
    plotted_noise='provided')
plt.show()
```



CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

C

E

F

G

W

X