PTA

Mattia Gollub

Aug 23, 2023

CONTENTS

This is the documentation for the PTA python package. PTA is available on git and via pip. See *getting started* for the installation instructions and a simple example.

CHAPTER

ONE

PROBABILISTIC THERMODYNAMIC ANALYSIS

Probabilistic Thermodynamic Analysis (PTA)[1] is a framework for the exploration of the thermodynamic properties of a metabolic network. In PTA, we consider the *steady-state thermodynamic space* of a network, that is, the space of standard reaction energies and metabolite concentrations that are compatible with steady state flux constraints. The uncertainty of the variables in the thermodynamic space is modeled with a probability distribution, allowing analysis with optimization and sampling approaches.

1.1 Probabilistic Metabolic Optimization (PMO)

PMO aims at finding the most probable values of reaction energies and metabolite concentrations that are compatible with the steady state constrain. This method is particularly useful to indentify features of the network that are thermodynamically unrealistic. For example, PMO can identify substrate channeling, incorrect cofactors or inaccurate directionalities.

1.2 Thermodynamic and Flux Sampling (TFS)

TFS allows to jointly sample the thermodynamic and flux spaces of a network. The method provides estimates of metabolite concentrations, reactions directions, and flux distributions.

1.2.1 Getting started

Requirements

In order to install PTA you need:

- Windows, Linux or OSX operating system.
- Python 3.8 or newer.
- One of the QP + MILP solvers supported by CVXPY. For models where all reaction directions are fixed, a QP solver is sufficient.
- *Optional (only for running the samplers)*:
  - A compiler supporting C++ 17.

[1] Mattia G Gollub, Hans-Michael Kaltenbach, and Jörg Stelling. Probabilistic thermodynamic analysis of metabolic networks. *Bioinformatics*, 37(18):2938–2945, 2021.

  – The Gurobi solver (free for academia). Make sure you install the python bindings. On OSX systems you may need to manually set the environment variable `GUROBI_HOME` to `/Library/gurobi<version>/<platform>`, since the Gurobi installer does not do it automatically.

Installation

Once your system satisfies all the requirements, you can install PTA through the Python Package Index:

```
pip install pta
```

Installing PTA without samplers

You can install PTA without compiling the samplers by defining the *PTA_NO_SAMPLERS* environment variable:

```
PTA_NO_SAMPLERS=1 pip install pta            # Linux, OSX
set "PTA_NO_SAMPLERS=1" && pip install pta    # Windows
```

Installing PTA this way does not require Gurobi and the C++ compiler, but the functionalities will be limited to running PMO and model assessment methods. The sampler will not work.

Simple example

This example shows a basic workflow with PTA. First it loads a model and creates its thermodynamic space, then it runs PMO and TFS on it.

```
import enkie
import pta

# Load a SBML model and set bounds on known rates (e.g. growth).
model = pta.load_example_model("e_coli_core")
model.reactions.BIOMASS_Ecoli_core_w_GAM.lower_bound = 0.5

# Preprocess the model to make sure it is thermodynamically consistent.
pta.prepare_for_pta(model)

# Construct the thermodynamic space of the model.
thermodynamic_space = pta.ThermodynamicSpace.from_cobrapy_model(
  model,
  metabolites_namespace="bigg.metabolite",
  parameters=enkie.CompartmentParameters.load("e_coli")
)

# Run PMO on the model (by default it maximizes the probability of concentrations
# and reaction energies).
problem = pta.PmoProblem(model, thermodynamic_space)
problem.solve()

# Analyze the predicted concentrations and reaction energies, revealing potential
# knowledge gaps and inaccuracies in the model.
pta.QuantitativeAssessment(problem).summary()

```

(continues on next page)

(continued from previous page)

```python
# Sample the thermodynamic space of the network.
tfs_model = pta.TFSModel(model, thermodynamic_space, solver="GUROBI")
sampling_result = pta.sample_drg(tfs_model)
# sampling_result now contains samples of reaction energies as well as the
# probability of each orthant.
```

Expanding the example

The next pages of this documentation provide more details about each step and how to customize it for your work. You can also look at additional examples.

- You can gain more control over the *preprocessing step*.
- The thermodynamic space should be *constructed* using parameters (pH, ionic strength, ...) and metabolite concentrations representative of the system you are modeling.
- You can use different objectives for PMO and *access* the predicted values directly.
- You can use the information *provided* by `QuantitativeAssessment` to curate your model.
- TFS can be used to *characterize* the feasible thermodynamic and flux spaces of the network.

1.2.2 Model preprocessing

A model must be thermodynamically sound in order to be used in PTA, which is usually not the case for published genome-scale models. We provide a simple method that automatically preprocesses a model for use in PTA.

```python
pta.prepare_for_pta(model)
```

The modifications performed by this method include relaxing flux bounds, removing blocked reactions and removing "dead-end" futile cycles, thus they do not add further constraints to the capabilities of the model.

However, in some cases it may be relevant to obtain a complete report of the issues present in the model or to resolve these issues manually. The following sections explain how to do that. Only either `prepare_for_pta()` or the steps below are required.

Blocked reactions

In PTA all reactions have a well-defined direction, meaning that all reaction must have non-zero flux. This requires that all blocked reactions are removed from the model in advance. You can easily do this using COBRApy's functions:

```python
import cobra
from cobra.flux_analysis.variability import find_blocked_reactions

model.remove_reactions(
    find_blocked_reactions(model),
    remove_orphans=True
)
```

Forced internal cycles

Many models contain thermodynamically unfeasible internal cycles that must be active in any non-zero solution. This is usually a consequence of incorrect irreversibilities and "dead-end" cycles. For example:

- The `SUCDi` and `FRD7` reactions in the *e_coli_core* model both have the same stoichiometry and are irreversible, but in opposing directions.

  ```
  SUCDi: succ_c + q8_c  -> q8h2_c + fum_c
  FRD7:  q8h2_c + fum_c -> succ_c + q8_c
  ```

  This would imply that both directions are thermodynamically favorable, which is not possible. The solution is to make both reactions reversible and let flux and thermodynamic constraints determine their direction.

- The `FOMETRi` and `THFAT` in the iML1515 model suffer of the same problem:

  ```
  THFAT:   h2o_c + methf_c -> 5fthf_c + h_c
  FOMETRi: 5fthf_c + h_c   -> h2o_c + methf_c
  ```

  However, this is a "dead-end" cycle, since only `5fthf_c` does not participate to any other reaction. In this case, even making the two reactions reversible would not help, because any non-zero flux solution at steady state requires the two reactions to have opposing directions. Since cycles like this would anyway have no effect on the rest of the network we simply remove the reactions involved.

We emphasize that the presence of internal cycles in general is not a problem, and one of the strengths of PTA is to obtain loopless, thermodynamically feasible solutions. This section deals with internal cycles that must always be active and thus make the model unfeasible in principle.

Detecting inconsistencies

You can use the `StructuralAssessment` class to list all the problematic cycles in the model.

```
assessment = pta.StructuralAssessment(model, 'Biomass', 'ATPM')
assessment.summary()
```

```
> The following internal cycles are thermodynamically unfeasible, but must be active
in any non-zero flux solution, meaning that the model is thermodynamically
inconsistent.
...
0. THMDt2pp_copy1, THMDt2pp_copy2
1. ALAt2pp_copy1, ALAt2pp_copy2
2. CYTDt2pp_copy1, CYTDt2pp_copy2
3. ACCOAL, PPAKr, PTA2
4. ADNt2pp_copy1, ADNt2pp_copy2
5. FOMETRi, THFAT
...
```

Note that you need to specify the identifier of the biomass and ATP maintenance reactions.

Resolving inconsistencies

You can then resolve the problems manually or attempt to solve them automatically (logging is only needed to print the actions performed):

```
import logging
logging.basicConfig()
logging.getLogger().setLevel(logging.INFO)

assessment.autoresolve(model)
```

```
Removing bounds for all reactions in the cycle: THMDt2pp_copy1, THMDt2pp_copy2.
Removing bounds for all reactions in the cycle: ALAt2pp_copy1, ALAt2pp_copy2.
Removing bounds for all reactions in the cycle: CYTDt2pp_copy1, CYTDt2pp_copy2.
Removing bounds for all reactions in the cycle: ACCOAL, PPAKr, PTA2.
Removing bounds for all reactions in the cycle: ADNt2pp_copy1, ADNt2pp_copy2.
Removing all reactions of the dead-end cycle: FOMETRi, THFAT.
...
```

If you identified inconsistencies in a condition-specific model, you can apply `autoresolve()` on the base model in order to have the curation available for any condition-specific model generated from it.

Computing tighter bounds

To make the model simpler for later steps (PMO and TFS) it is good practice to restrict the bounds of each reaction to the bounds computed with Flux Variability Analysis.

```
pta.utils.tighten_model_bounds(model)
```

In order to not overconstrain the model and to avoid introducing small coefficients that would be challenging for PMO, this function may add small numerical margins. In any case, the resulting bounds are guaranteed to be equal or tighter than the original ones.

1.2.3 The thermodynamic space

The quick way

The thermodynamic space of a network models metabolite and free energies for a selected set of reactions. In absence of additional information, a thermodynamic space can be constructed from a COBRApy model:

```
thermodynamic_space = pta.ThermodynamicSpace.from_cobrapy_model(model)
```

This is a wrapper around the constructor of the :class:.:*ThermodynamicSpace* class, which can be used when interfacing with other modeling frameworks.

Advanced initialization

When additional information is available, one can provide a series of additional optional parameters. The parameters are explained in the following sections.

```
thermodynamic_space = pta.ThermodynamicSpace.from_cobrapy_model(
    model,
    metabolites_namespace,
    constrained_rxns,
    estimator,
    parameters,
    concentrations
)
```

Metabolites namespace

PTA needs to know how to find standardized metabolite identifiers. PTA reads identifiers from the annotations in the SBML, but you need to specify which annotations you want to use. For example, for models downloaded from the BiGG database, it is common to choose the `"bigg.metabolite"` namespace. By default, the first annotation for each metabolite is used, but it is always recommended to specify a namespace manually to avoid inconsistencies.

Constrained reactions

The `constrained_rxns` parameter specifies which reactions in the network should be modeled in the thermodynamic space. Thermodynamic constraints should be applied only to balanced reaction, as pseudo-reactions are not thermodynamically realistic. Thus, by default, boundary reactions and biomass are excluded. Reactions can be specified as lists of indices, lists of identifiers or lists of reactions. As a starting point, we recommend building the list of constrained reactions using:

```
pta.utils.get_candidate_thermodynamic_constraints(
    model,
    metabolites_namespace
)
```

Estimator

An object used for estimating standard reaction energies. At the moment PTA only uses eQuilibrator (`EquilibratorGibbsEstimator`), but you are free to implement an interface for alternative estimation tools.

Compartment parameters

An object specifying the parameters (pH, pMg, ionic strength, electrostatic potential) of each compartment. Temperature must be constant for the entire system. You can either fill this object yourself or load one of the default ones (currently *e_coli* and *human*):

```
import enkie

parameters = enkie.CompartmentParameters.load('e_coli')
```

Concentrations prior

`ConcentrationsPrior` objects specify measured or assumed distributions for the concentration of each metabolite. Since measurements are usually not available for all metabolites, PTA will use the following information, in order of preference, to determine the distribution of metabolite M in compartment C:

1. The distribution of the concentration of M itself.
2. A default distribution for any metabolite in C.
3. A default distribution for any metabolite.

You can load priors from files or create your own. PTA includes priors for the extracellular concentrations in M9 (`M9_aerobic`, `M9_anaerobic`) and for the intracellular concentrations with different carbon sources (`ecoli_M9_<s>`, where <s> can be `ac`, `fru`, `gal`, `glc`, `glcn`, `glyc`, `pyr`, `succ`) based on[1]. Different can also be combined. In case of conflict, the added prior overwrites distributions of the original prior.

```
# Load the prior for metabolite concentrations in M9 media.
concentrations = pta.ConcentrationsPrior.load('M9_aerobic')

# Add the prior for intracellular concentrations when growing on succinate.
concentrations.add(pta.ConcentrationsPrior.load('ecoli_M9_succ'))

# Manually add a distribution. Note that we must use log-normal distributions.
import numpy as np
concentrations.metabolite_distributions[('bigg.metabolite:g3p', 'c')] = \
    pta.LogNormalDistribution(log_mean=np.log(1e-4), log_std=0.2)

# Updating identifiers after a modification is recommended to make sure that we can
properly recognize identifiers from different namespaces.
concentrations.update_identifiers()

# Save your newly created prior for later use.
concentrations.save('my_prior.csv')
```

Thermodynamic space basis

Because of correlation existing within and between metabolite concentrations, standard reaction energies and reaction energies, the dimensionality of the thermodynamic space is usually lower than the number of variables. One can easily obtain a full-dimensional representation that can be used for PMO and TFS:

```
basis = pta.ThermodynamicSpaceBasis(thermodynamic_space)
```

This class contains a mapping between a full dimensional basis and the variables of the thermodynamic space. Variables of the basis are referred to as **m** in[2].

[1] Luca Gerosa, Bart RB Haverkorn van Rijsewijk, Dimitris Christodoulou, Karl Kochanowski, Thomas SB Schmidt, Elad Noor, and Uwe Sauer. Pseudo-transition analysis identifies the key regulators of dynamic metabolic adaptations from steady-state data. *Cell systems*, 1(4):270–282, 2015.

[2] Mattia G Gollub, Hans-Michael Kaltenbach, and Jörg Stelling. Probabilistic thermodynamic analysis of metabolic networks. *Bioinformatics*, 37(18):2938–2945, 2021.

References

1.2.4 Optimizing thermodynamics

Probabilistic Metabolic Optimization (PMO) allows to run optimization problems using both flux and thermodynamic constraints. This is done using the `PmoProblem` class.

```python
# Create a PMO problem.
problem = pta.PmoProblem(model, thermodynamic_space)

# Solve the PMO problem. The object now contains the solution.
status = problem.solve()
```

Internally, optimization problems are constructed using CVXPY as interface for different solvers. The CVXPY optimization variables are `problem.m` (the basis of the thermodynamic space), `problem.d` (the direction of the reactions constrained by thermodynamics) and `problem.vs` (the scaled fluxes). For convenience, we provide properties to access predicted values of the biologically relevant variables `v` (fluxes), `log_c` (log-concentrations), `drg` (reaction energies), `drg0` (standard reaction energies).

Setting the objective

By default, PMO maximizes the probability of the variables in the thermodynamic space. It is possible to set custom objective (e.g. on fluxes or concentrations) by passing a function creating the objective to the constructor.

```python
import cvxpy as cp
objective = lambda p: cp.Minimize(p.vs[26])
problem = pta.PmoProblem(model, thermodynamic_space, objective=objective)
```

Numerics

PMO uses the Big-M formulation to encode the directionality constraints in a MILP problem. While this is an efficient representation, it is also known to have numerical limitations. In particular, PMO may fail or give inaccurate solutions if fluxes and reaction energies span more than nine orders of magnitude.

For this reason, we internally transform the problem to use scaled fluxes and a full-dimensional basis for the thermodynamic space. While we can automatically compute optimal values for the Big-M coefficients of fluxes, PMO guidance for the reaction energies. In practice, we assume that all reaction energies have magnitude between 0.1 and 1000 kJ/mol, which we believe is biologically reasonable. It is possible to change these values in the constructor of `PmoProblem` in case larger or smaller reaction energies are expected. However, this should be done with care, keeping in mind that it may lead to numerical issues.

Currently we optimized numerics-related solver parameters for Gurobi only. If you use other solvers, you may need to use the `solver_options` argument to set stricter feasibility tolerances for your solver.

1.2.5 Model assessment

Thermodynamics-based model assessment consists in finding parts of the network where the model structure and the measured or assumed metabolite concentrations are not compatible, suggesting model inaccuracies or novel mechanisms. Such cases can have different explanations:

- COBRA models have been constructed with Flux Balance Analysis (FBA) as the main application and frequently contain thermodynamic inaccuracies. Some reactions may be annotated as irreversible in the model, while they are not in practice.
- The direct interpretation of COBRA models is that individual reactions are independent events, i.e. (1) substrates diffuse to the enzyme, (2) substrates are converted to products (3) products equilibrate with the compartment. However the actual mechanism could be different. For example, substrate channeling could occur. In that case, an intermediate is passed directly from one enzyme to the other, without equilibrating with the metabolite pool in the compartment. While this mechanism is irrelevant for FBA, it has significant influence on thermodynamics.
- Despite continuous improvements, we still don't know the exact stoichiometry of metabolic networks, even for well-studies organisms. Reactions may be missing or have incorrect cofactors (with different thermodynamic properties).

Additionally, model assessment can tell us which reactions impose the strongest constraints on the thermodynamics of the network.

See[1] for additional information.

Thermodynamic anomalies

We can use PMO to detect *anomalies* in the network, i.e. variables whose predicted value is significantly different from its prior. A predicted value is flagged as anomaly if its z-score is larger than a certain threshold. A threshold of one means that a variable is flagged only if its predicted value is at least one standard deviation away from the its initial estimate.

Identifying anomalies

The `QuantitativeAssessment` class can be used to detect anomalies in a PMO solution. While you can set arbitrary objective in PMO, we recommend to use the default objective (maximize the probability of the thermodynamic variables).

```python
# Construct and solve the PMO problem.
problem = pta.PmoProblem(model, thermodynamic_space)
problem.solve()

# Analyze the result to find anomalies in the predicted values.
assessment = pta.QuantitativeAssessment(problem)
assessment.summary()
```

```
Quantitative thermodynamic assessment summary:
------------------------------------------------
conentrations: mM, free energies: kJ/mol

> The following metabolites have been flagged as anomalies because their predicted
  concentration has an absolute z-score greater than 1.0:
```

(continues on next page)

[1] Mattia G Gollub, Hans-Michael Kaltenbach, and Jörg Stelling. Probabilistic thermodynamic analysis of metabolic networks. *Bioinformatics*, 37(18):2938–2945, 2021.

(continued from previous page)

```
            id       conc  z_log_c
524     mqn8_c  3.831e+03    4.843
520     mql8_c  1.434e-05   -4.843
234    fadh2_c  2.214e-05   -4.627
544     iasp_c  7.845e-05   -3.995
...

> No anomaly found in non-intracellular concentrations.

> The following reactions have been flagged as anomalies because their predicted
  free energy or standard free energy has an absolute z-score greater than 1.0:
            id          v     drg0        drg  z_drg     z_drg0     sp_drg
178         ASPO4  2.020e-08   84.123 -1.000e-03 -9.908 -5.824e+00  1.547e+00
479       GLYCTO3  1.373e-07   59.109 -1.000e-03 -7.215 -4.758e+00  2.611e-01
378          FRD2  7.783e-07  -75.359 -2.391e+01  6.513  4.571e+00  4.807e-12
...

> The following reactions are predicted to impose strong thermodynamic constraints
  on the network because their shadow price is greater than 0.1:
        id          v    drg0        drg  z_drg     z_drg0  sp_drg
178    ASPO4  2.020e-08  84.123 -1.000e-03 -9.908 -5.824e+00   1.547
127    AIRC3 -2.692e-01 -30.990  1.000e-03  5.580  3.627e+00   1.161
588     IMPD  1.462e-01  48.407 -1.000e-03 -5.900 -4.249e+00   0.991
...
```

Note that you can adjust the threshold for anomalies in the constructor of the `QuantitativeAssessment` class.

Interpreting and curating anomalies

Once you obtain the predicted anomalies you should investigate what is the reason. Here is a checklist of the most common reasons for anomalies:

1. Is the anomaly expected? E.g. oxygen concentration is often flagged as anomaly, but it is normal to expect low concentrations.
2. If the reaction is irreversible, is this annotation correct? Sometimes COBRA models are too restrictive in that sense.
3. Is the metabolite channeled between two reactions? Or is there at least some hint for that in the literature or in databases such as STRING?
4. Is the stoichiometry of the reaction correct? Maybe a reaction can use different cofactors. Moreover, some core models may use a single electron acceptor for all reactions, even if different ones are used in practice.
5. How reliable is the standard reaction energy for the reaction? Free energies for exotic or large compounds are determined with group contribution, which may underestimate the uncertainty.
6. Is the structure of the metabolite defined? Complex sugars such as glycogen can have different structures in different databases.

Points 3 and 4 are particularly interesting, as they can reveal new biological mechanisms. See[Page 11, 1] for additional information.

The snippet above shows part of the output obtained running the assessment on the iJO1366 model. It shows anomalies in the concentrations of `mqn8`, `mql8` and `iasp` as well as in the standard free energy of of the reaction they participate to (aspartate oxidase, `ASPO4`). In this case the most probable explanation is that iminoaspartate (`iasp`) is channeled between aspartate oxidase and quinolinate synthase ([2]).

We can now curate the model by replacing the two reactions with a lumped reaction representing the channel. Since aspartate oxidase can use different electron acceptors, we should do the same with all variants of the reaction.

We recommend investigating and resolving anomalies iteratively: start from the highest anomaly and either resolve it or accept it. If you modified the model, run the assessment again and continue with the next highest anomaly.

Performance considerations

Running PMO can take a considerable amount of time on medium-large models. The runtime is even higher when the model contains several anomalies because the solution is pushed towards regions of low probability (further away from the objective). In these cases we recommend to separate the process in two steps:

1. Run the assessment and curate the model for each compartment independently. For example you can construct a thermodynamic space that only covers intracellular reactions as follows:

```python
# Get candidate thermodynamic constraints for the model, excluding
# extrcellular and periplasmic reactions.
constrained_rxn_ids = pta.get_candidate_thermodynamic_constraints(
    model,
    metabolites_namespace="bigg.metabolite",
    exclude_compartments=['p', 'e']
)

# Construct a thermodynamic space covering only the selected reactions.
thermodynamic_space = pta.ThermodynamicSpace.from_cobrapy_model(
    model,
    metabolites_namespace="bigg.metabolite",
    constrained_rxns=constrained_rxn_ids
)
```

2. After curating each compartment, you can apply PMO and model assessment on the entire network. This should now be much faster.

References

1.2.6 Sampling thermodynamics and fluxes

Thermodynamics and Flux Sampling (TFS) allows to sample steady state reaction energies, standard reaction energies, metabolite concentrations and fluxes in a metabolic network.

TFS consists in two main steps:

1. Sample reaction energies and orthants (sets of reaction directions in the model).
2. Sample standard reaction energies, metabolite concentrations and reaction fluxes conditioned on the sampled reaction energies and orthants.

[2] Ilaria Marinoni, Simona Nonnis, Carmine Monteferrante, Peter Heathcote, Elisabeth Härtig, Lars H Böttger, Alfred X Trautwein, Armando Negri, Alessandra M Albertini, and Gabriella Tedeschi. Characterization of l-aspartate oxidase and quinolinate synthase from bacillus subtilis. *The FEBS journal*, 275(20):5090–5107, 2008.

Sampling reaction energies

The first step is to sample reaction energies and estimate the probability of the orthants of the thermodynamic space:

```python
# Wrap a cobrapy model and the corresponding thermodynamic space in a single object.
tfs_model = pta.TFSModel(model, thermodynamic_space)

# Sample the thermodynamic space.
result = pta.sample_drg(tfs_model)
```

The `result` object contains the following fields:

- `samples`: Pandas data frame containing the samples of reaction energies.
- `orthants`: Pandas data frame containing the signs (-1 / +1) of the sampled orthants. An additional column `weight` stores the weight of each orthant.
- `psrf` Pandas series containing the Potential Scale Reduction Factor (PSRF) of each reaction energy. This is only used to assess convergence of the sampling procedure.

Note that the space of steady state reaction energies is generally non-convex and disconnected, thus convergence can take a high number of steps. While PTA tries to automatically choose a good number of steps, this may not be sufficient for challenging models and you may get a warning that sampling did not converge. In this case you should manually tell `sample_drg()` to use more steps using the `num_steps` option.

Sampling the conditionals

The remaining quantities can be sampled from the reaction energies:

```python
# Sample the natural logarithm of the metabolite concentrations. This function draws
# one concentration sample for each reaction energy sample.
log_conc = sample_log_conc_from_drg(thermodynamic_space, result.samples)

# Sample the standard reaction energies. This function draws one standard reaction
# energy sample for each reaction energy sample.
drg0 = sample_drg0_from_drg(thermodynamic_space, result.samples)

# Sample reaction fluxes. For each unique orthant represented by the reaction energy
# samples this function draws a number of flux samples proportional to the weight of
# the orthant.
fluxes = sample_fluxes_from_drg(model, result.samples, result.orthants)
```

Each function returns a Pandas data frame containing the samples. For concentrations and standard reaction energies, PTA can use an analytical expression of the conditional probability, which makes sampling very fast. However, for fluxes we need to run uniform sampling on each orthant separately, which can take a long time for medium-large models. If sampling fluxes takes too long, pass fewer reaction energy samples to reduce the number of orthants sampled.

1.2.7 API Reference

This page contains auto-generated API reference documentation[1].

`pta`

Probabilistic Thermodynamic Analysis of metabolic networks.

Subpackages

`pta.data`

Subpackages

`pta.data.concentration_priors`

`pta.data.models`

`pta.sampling`

Submodules

`pta.sampling.commons`

Common classes and methods for sampling algorithms.

Module Contents

exception `pta.sampling.commons.`**`SamplingException`**

Bases: `Exception`

Raised when an exception occurs during sampling.

class `pta.sampling.commons.`**`SamplingResult`**(*samples: pandas.DataFrame*, *psrf: pandas.Series*, *basis_samples: pandas.DataFrame = None*, *chains: numpy.ndarray = None*)

Encapsulates the result of a sampler.

Parameters

- **samples** (*pd.DataFrame*) – Data frame containing the samples as rows.
- **psrf** (*pd.Series*) – Series containing the PSRF of each variable.
- **basis_samples** (*pd.DataFrame, optional*) – The samples in the basis.
- **chains** (*np.ndarray, optional*) – The simulated chains.

property basis_samples: pandas.DataFrame

Gets a data frame containing the samples in the basis.

[1] Created with sphinx-autoapi

property samples: pandas.DataFrame

Gets a data frame containing the samples.

property chains: numpy.ndarray

Get the simulated chains.

property psrf: pandas.Series

Gets the Potential Scale Reduction Factor (PSRF) of each variable.

check_convergence(*max_psrf: float = default_max_psrf*) → bool

Checks whether the chains converged according to the given criteria.

Parameters
`max_psrf` (*float, optional*) – Maximum PSRF value for convergence.

Returns
True if the test succeeded, false otherwise.

Return type
bool

class pta.sampling.commons.**SamplerInterface**

Interface for an MCMC sampler.

abstract property dimensionality: int

Get the dimensionality of the sampling space.

abstract simulate(*settings: _pta_python_binaries.SamplerSettings, initial_points: numpy.ndarray, directions_transform: numpy.ndarray*) → Any

Run the sampler with the given parameters.

Parameters

- `settings` (*pb.SamplerSettings*) – Sampling settings.
- `initial_points` (*np.ndarray*) – The initial points for the chains.
- `directions_transform` (*np.ndarray*) – The transform for the directions sampler.

abstract compute_psrf(*result: Any*) → pandas.Series

Compute the potential scale reduction factors for the variables of interest on a given set of chains.

Parameters
`result` (*Any*) – The result of the sampling function.

Returns
The computed potential scale reduction factors.

Return type
pd.Series

abstract get_chains(*result: Any*) → numpy.ndarray

Extract the simulated chains from a given result.

Parameters
`result` (*Any*) – The result of the native sampling function.

Returns
The simulated chains.

Return type
np.ndarray

`pta.sampling.commons.`**`sample_from_chains`**(*chains: numpy.ndarray*, *num_samples: int*) → numpy.ndarray

Draws samples from a given set of chains.

Parameters

- **`chains`** (*`np.ndarray`*) – Numpy 3D array containing the chains.
- **`num_samples`** (*`int`*) – Number of samples to draw.

Returns

Array of samples.

Return type

np.ndarray

`pta.sampling.commons.`**`split_chains`**(*chains: numpy.ndarray*) → numpy.ndarray

Split the chains in two, threating each half as a new chain. This is often used to detect systematic trends in a random walk.

Parameters

`chains` (*`np.ndarray`*) – Numpy 3D array containing the input chains.

Returns

Numpy 3D array containing the resulting chains.

Return type

np.ndarray

`pta.sampling.commons.`**`apply_to_chains`**(*chains: numpy.ndarray*, *f: Callable[[numpy.ndarray], numpy.ndarray]*) → numpy.ndarray

Apply a function to each sample of a group of chains.

Parameters

- **`chains`** (*`np.ndarray`*) – Numpy 3D array containing the input chains.
- **`f`** (*`Callable[[np.ndarray], np.ndarray]`*) – Function to apply to each sample.

Returns

Numpy 3D array containing the transformed samples.

Return type

np.ndarray

`pta.sampling.commons.`**`split_R`**(*chains: numpy.ndarray*) → numpy.ndarray

Compute the split-R (or Potential Scale Reduction Factor) of each variable in the given chains.

Parameters

`chains` (*`np.ndarray`*) – Numpy 3D array containing the input chains.

Returns

Vector containing the split-R value for each variable.

Return type

np.ndarray

`pta.sampling.commons.`**`fill_common_sampling_settings`**(*settings: _pta_python_binaries.SamplerSettings*, *num_samples: int*, *num_steps: int*, *num_chains: int = -1*, *num_warmup_steps: int = -1*, *max_threads: int = default_max_threads*)

Fills default values for common sampling settings.

Parameters

- **settings** (*SamplerSettings*) – The settings object to be filled.
- **log_directory** (*str*) – Directory in which the sampling logs should be stored.
- **num_samples** (*int*) – Number of samples to draw.
- **num_steps** (*int*) – Number of steps to take with each chain.
- **num_chains** (*int, optional*) – Number of chains to use. Will be set to the number of CPUs by default.
- **num_warmup_steps** (*int, optional*) – Number of burn-in steps to discard. Will be set to half the number of steps by default.
- **log_interval** (*datetime.timedelta, optional*) – Interval between each logging event.

Raises
`ValueError` – If the inputs are inconsistent.

`pta.sampling.convergence_manager`

Management of MCMC simulation to achieve desired convergence criteria.

Module Contents

class `pta.sampling.convergence_manager.`**ConvergenceManager**(*sampler: pta.sampling.commons.SamplerInterface, num_initial_steps: int = -1, samples_based_rounding: bool = False*)

Class managing MCMC simulations to achieve desired convergence criteria.

Parameters

- **sampler** (*Sampler*) – Object implementing a specific MCMC sampler.
- **num_initial_steps** (*int, optional*) – Initial number of steps for simulations, by default -1 (automatic).
- **samples_based_rounding** (*bool, optional*) – If true, the direction sampling distribution will be adjusted after each iteration based on the distribution of samples so far, by default False

run(*settings: _pta_python_binaries.SamplerSettings, update_settings_function: Callable[[_pta_python_binaries.SamplerSettings, int], None], make_sampling_result_function: Callable[[Any], pta.sampling.commons.SamplingResult], initial_points: numpy.ndarray, max_steps: int, max_psrf: float*) → *pta.sampling.commons.SamplingResult*

Run the sampler using this manager until the given PSRF or maximum number of steps is reached.

Parameters

- **settings** (*pb.SamplerSettings*) – The initial settings for the sampler.
- **update_settings_function** (*Callable[[pb.SamplerSettings, int], None]*) – Function for updating the settings with different numbers of steps.
- **make_sampling_result_function** (*Callable[[Any], SamplingResult]*) – Function for packing the result of the native sampler in a SamplingResult object.
- **initial_points** (*np.ndarray*) – The initial points for the simulation.

- **max_steps** (*int*) – Maximum number of steps to simulate.
- **max_psrf** (*float*) – Maximum PSRF to declare convergence.

Returns
The result of the sampler.

Return type
SamplingResult

Raises
SamplingException – If the native sampler fails.

pta.sampling.primitives

Abstract base class for sampling over the flux polytope.

Module Contents

class pta.sampling.primitives.**FluxSpaceSamplingModel**(*polytope: PolyRound.api.Polytope*, *reaction_ids: List[str]*)

Bases: *pta.sampling.commons.SamplerInterface*

Object holding the information necessary to sample a flux space.

Parameters

- **polytope** (*Polytope*) – Polytope object describing the flux space.
- **reaction_ids** (*List[str]*) – Identifiers of the reactions in the flux space.

property G: numpy.array

Gets the left-hand side of the constraints of the flux space.

property h: numpy.array

Gets the right-hand side of the constraints of the flux space.

property to_fluxes_transform: Tuple[numpy.array, numpy.array]

Gets the transform from a point in the model to a point in the flux space.

property reaction_ids

Gets the IDs of the reactions in the model.

property dimensionality: int

Gets the dimensionality of the flux space.

to_fluxes(*value: numpy.array*) → numpy.array

Transform a point or matrix from the model to the flux space.

Parameters
value (*np.array*) – Input values.

Returns
The corresponding fluxes.

Return type
np.array

get_initial_points(*num_points: int*) → numpy.array

Gets initial points for sampling fluxes.

Parameters

- **model** (`UniformSamplingModel`) – The model to sample.
- **num_points** (*int*) – Number of initial points to generate.

Returns

Array containing the initial points.

Return type

np.array

pta.sampling.tfs

Thermodynamics-based sampling of free energies, concentrations and fluxes.

Module Contents

class pta.sampling.tfs.**TFSModel**(*network: Union[cobra.Model, pta.flux_space.FluxSpace], thermodynamic_space: pta.thermodynamic_space.ThermodynamicSpace, thermodynamic_space_basis: pta.thermodynamic_space.ThermodynamicSpaceBasis = None, confidence_level: float = 0.95, min_drg: float = 0.1, max_drg: float = 1000, solver: Optional[str] = None, solver_options: dict = None*)

Bases: *pta.sampling.commons.SamplerInterface*

Object holding the information necessary to run TFS.

Parameters

- **network** (*Union[cobra.Model, FluxSpace]*) – Cobra model or *FluxSpace* object describing the flux space of the metabolic network.
- **thermodynamic_space** (`ThermodynamicSpace`) – Description of the thermodynamic space of the metabolic network.
- **thermodynamic_space_basis** (`ThermodynamicSpaceBasis`, *optional*) – A basis for the thermodynamic space. If specified, *m* will be defined in this basis.
- **objective** (*Callable[[PmoProblem], cp.problems.objective.Objective], optional*) – A function used to set the optimization objective. By default the probability of in thermodynamic space is maximized.
- **confidence_level** (*float, optional*) – Confidence level (in the range $[0.0, 1.0[$) on the joint of the thermodynamic variables, by default 0.95.
- **min_drg** (*float, optional*) – Minimum magnitude for the reaction energy of each reaction, by default 1e-1.
- **max_drg** (*float, optional*) – Maximum magnitude for the reaction energy of each reaction, by default 1000.
- **solver** (*Optional[str], optional*) – Name of the solver to use, this can be any of the solvers supported by CVXPY, by default None.

- **`solver_options`** (*`dict`*, *`optional`*) – Dictionary specifying additional options for the solver.

`property T:` *`pta.thermodynamic_space.ThermodynamicSpace`*

Gets the thermodynamic space used for sampling.

`property F:` *`pta.flux_space.FluxSpace`*

Gets the flux space used for sampling.

`property B:` *`pta.thermodynamic_space.ThermodynamicSpaceBasis`*

Gets the basis of the thermodynamic space used for sampling.

`property pmo_args`

Gets the arguments used to construct PMO problems.

`property dimensionality:` `int`

Gets the dimensionality of the basis of the thermodynamic space.

`property confidence_radius`

Gets the radius of the selected confidence region.

`property drg_epsilon`

Gets the minimum magnitude of the reaction energy of irreversible reactions.

`property reversible_rxn_ids`

Gets the identifiers of the reversible reactions in the thermodynamic space.

`get_reversible_reactions_ids_T``()` → List

Get the ids of the reversible reactions in the thermodynamic space

Returns
The corresponding reactions ids

Return type
List

`to_drg`(*value: numpy.ndarray*) → numpy.ndarray

Transform a point or matrix from the the basis to reaction energies.

Parameters
`value` (*`np.ndarray`*) – Input values in the basis.

Returns
The corresponding reaction energies.

Return type
np.ndarray

`simulate`(*settings: _pta_python_binaries.SamplerSettings*, *initial_points: numpy.ndarray*, *directions_transform: numpy.ndarray*) → _pta_python_binaries.TFSResult

Run the sampler with the given parameters.

Parameters

- **`settings`** (*`pb.SamplerSettings`*) – Sampling settings.
- **`initial_points`** (*`np.ndarray`*) – The initial points for the chains.
- **`directions_transform`** (*`np.ndarray`*) – The transform for the directions sampler.

`compute_psrf`(*result: _pta_python_binaries.TFSResult*) → pandas.Series

Compute the potential scale reduction factors for the variables of interest on a given set of chains.

Parameters
`result` (*pb.TFSResult*) – The result of the sampling function.

Returns
The computed potential scale reduction factors.

Return type
pd.Series

`get_chains`(*result: _pta_python_binaries.TFSResult*) → numpy.ndarray

Extract the simulated chains from a given result.

Parameters
`result` (*pb.TFSResult*) – The result of the native sampling function.

Returns
The simulated chains.

Return type
np.ndarray

`get_initial_points`(*num_points: int*) → numpy.ndarray

Gets initial points for sampling reaction energies.

Parameters
`num_points` (*int*) – Number of initial points to generate.

Returns
Array containing the initial points.

Return type
np.ndarray

`class pta.sampling.tfs.FreeEnergiesSamplingResult`(*samples: pandas.DataFrame*, *psrf: pandas.Series*, *orthants: pandas.DataFrame*, *basis_samples: pandas.DataFrame = None*, *chains: numpy.ndarray = None*)

Bases: `pta.sampling.commons.SamplingResult`

Encapsulates the result of sampling reaction energies.

Parameters

- `samples` (*pd.DataFrame*) – Data frame containing the free energy samples.
- `psrf` (*pd.Series*) – The Potential Scale Reduction Factors of each variable.
- `orthants` (*pd.DataFrame*) – Data frame containing the signs of the reversible reactions for each orthants. Contains an additional column ("weight") describing the weight of the orthant.
- `basis_samples` (*pd.DataFrame, optional*) – The samples in the basis.
- `chains` (*np.ndarray, optional*) – The simulated chains.

`property orthants: pandas.DataFrame`

Gets a data frame containing the sampled orthants. Contains an additional column ("weight") describing the weight of the orthant.

`pta.sampling.tfs.`**`sample_drg`**`(`*model:* TFSModel, *num_samples:* *int = default_num_samples*, *num_direction_samples:* *int = default_num_samples*, *max_steps:* *int = -1*, *max_psrf:* *float = default_max_psrf*, *num_chains:* *int = -1*, *initial_points:* *numpy.ndarray = None*, *num_initial_steps:* *int = -1*, *feasibility_cache_size:* *int = tfs_default_feasibility_cache_size*, *min_rel_region_length:* *float = tfs_default_min_rel_region_length*, *max_threads:* *int = default_max_threads*, *convergence_manager:* pta.sampling.convergence_manager.ConvergenceManager = *None*`)` → *FreeEnergiesSamplingResult*

Sample reaction energies under steady state flux constraints in the given model.

Parameters

- **`model`** (`TFSModel`) – The model to sample.
- **`num_samples`** (*`int, optional`*) – Number of samples to draw.
- **`num_direction_samples`** (*`int, optional`*) – Number of orthant samples to collect.
- **`max_steps`** (*`int, optional`*) – The maximum number fo steps to simulate.
- **`max_psrf`** (*`float, optional`*) – Maximum value of the PSRFs for convergence.
- **`num_chains`** (*`int, optional`*) – The number of chains to simulate.
- **`initial_points`** (*`np.ndarray, optional`*) – The initial points for the chains.
- **`num_initial_steps`** (*`int, optional`*) – Initial chains length.
- **`feasibility_cache_size`** (*`int, optional`*) – Maximum size of the cache storing the feasibility of the orthants encountered during the random walk.
- **`min_rel_region_length`** (*`float, optional`*) – Minimum length (relative to the length of the entire ray) of a segment in order to consider it for sampling.
- **`max_threads`** (*`int, optional`*) – The maximum number of parallel threads to use.
- **`convergence_manager`** (`ConvergenceManager`, *`optional`*) – The object to use to monitor and improve convergence.

Returns
The sampling result.

Return type
FreeEnergiesSamplingResult

Raises
`SamplingException` – If sampling fails.

`pta.sampling.tfs.`**`sample_log_conc_from_drg`**`(`*thermodynamic_space:* pta.thermodynamic_space.ThermodynamicSpace, *drg_samples: pandas.DataFrame*, *min_eigenvalue:* *float = default_min_eigenvalue_tds_basis*`)` → pandas.DataFrame

Sample the natural logarithm of the metabolite concentrations conditioned on samples of free energies. This function draws one sample for each sample of reaction energies.

Parameters

- **`thermodynamic_space`** (`ThermodynamicSpace`) – The thermodynamic space of the network.
- **`drg_samples`** (*`pd.DataFrame`*) – Data frame containing the samples of reaction energies.

- **min_eigenvalue** (*float, optional*) – Minimum eigenvalue to keep when performing the truncated SVD of the covariance of the conditional probability.

Returns
Data frame containing the log-concentration samples.

Return type
pd.DataFrame

pta.sampling.tfs.**sample_drg0_from_drg**(*thermodynamic_space:* pta.thermodynamic_space.ThermodynamicSpace, *drg_samples: pandas.DataFrame*, *min_eigenvalue: float = default_min_eigenvalue_tds_basis*) → pandas.DataFrame

Sample standard reaction energies conditioned on samples of reaction energies. This function draws one sample for each sample of reaction energies.

Parameters

- **thermodynamic_space** (ThermodynamicSpace) – The thermodynamic space of the network.
- **drg_samples** (*pd.DataFrame*) – Data frame containing the samples of reaction energies.
- **min_eigenvalue** (*float, optional*) – Minimum eigenvalue to keep when performing the truncated SVD of the covariance of the conditional probability.

Returns
Data frame containing the standard reaction energy samples.

Return type
pd.DataFrame

pta.sampling.tfs.**sample_fluxes_from_drg**(*model: cobra.Model*, *drg_samples: pandas.DataFrame*, *orthants: pandas.DataFrame*, *num_approx_samples: int = default_num_samples*) → pandas.DataFrame

Sample the flux space using the samples of orthant of reaction energies and orthants as prior. For each unique orthant implied by the reaction energy samples, this function draws a number of uniform flux samples proportional to the probability of the orthant in the thermodynamic space.

Parameters

- **model** (*cobra.Model*) – cobrapy model describing the flux space.
- **drg_samples** (*pd.DataFrame*) – The input reaction energy samples.
- **orthants** (*pd.DataFrame*) – Data frame containing the sampled orthants and their weights.
- **num_approx_samples** (*int, optional*) – Approximate number of samples to draw.

Returns
Data frame containing the flux samples.

Return type
pd.DataFrame

pta.sampling.uniform

Uniform sampling of the flux space of a metabolic network.

Module Contents

class pta.sampling.uniform.**UniformSamplingModel**(*polytope: PolyRound.api.Polytope*, *reaction_ids: List[str]*)

Bases: *pta.sampling.primitives.FluxSpaceSamplingModel*

Object holding the information necessary to run uniform sampling on a flux space.

Parameters

- **polytope** (*Polytope*) – Polytope object describing the flux space.
- **reaction_ids** (*List[str]*) – Identifiers of the reactions in the flux space.

classmethod from_cobrapy_model(*model: cobra.Model*, *infinity_flux_bound: float = default_flux_bound*) → *UniformSamplingModel*

Builds a uniform sampler model from a cobrapy model.

Parameters

- **model** (*cobra.Model*) – The input cobra model.
- **infinity_flux_bound** (*float, optional*) – Default bound to use for unbounded fluxes.

Returns
The constructed model.

Return type
UniformSamplingModel

simulate(*settings: _pta_python_binaries.SamplerSettings*, *initial_points: numpy.ndarray*, *directions_transform: numpy.ndarray*) → numpy.ndarray

Run the sampler with the given parameters.

Parameters

- **settings** (*pb.SamplerSettings*) – Sampling settings.
- **initial_points** (*np.ndarray*) – The initial points for the chains.
- **directions_transform** (*np.ndarray*) – The transform for the directions sampler.

compute_psrf(*result: numpy.ndarray*) → pandas.Series

Compute the potential scale reduction factors for the variables of interest on a given set of chains.

Parameters
result (*np.ndarray*) – The result of the sampling function.

Returns
The computed potential scale reduction factors.

Return type
pd.Series

`get_chains`(*result: numpy.ndarray*) → numpy.ndarray

Extract the simulated chains from a given result.

Parameters
`result` (*np.ndarray*) – The result of the native sampling function.

Returns
The simulated chains.

Return type
np.ndarray

`pta.sampling.uniform.sample_flux_space_uniform`(*model: Union[cobra.Model, UniformSamplingModel], num_samples: int = default_num_samples, max_steps: int = -1, max_psrf: float = default_max_psrf, num_chains: int = -1, initial_points: numpy.array = None, num_initial_steps: int = -1, max_threads: int = default_max_threads, convergence_manager: pta.sampling.convergence_manager.ConvergenceManager = None*) → *pta.sampling.commons.SamplingResult*

Sample steady state fluxes in the given model. The sampler run until either max_steps or max_psrf is reached.

Parameters

- **model** (*Union[cobra.Model, UniformSamplingModel]*) – The model to sample.
- **num_samples** (*int, optional*) – Number of samples to draw.
- **max_steps** (*int, optional*) – The maximum number fo steps to simulate.
- **max_psrf** (*float, optional*) – Maximum value of the PSRFs for convergence.
- **num_chains** (*int, optional*) – The number of chains to simulate.
- **initial_points** (*np.array, optional*) – The initial points for the chains.
- **num_initial_steps** (*int, optional*) – Initial chains length.
- **max_threads** (*int, optional*) – The maximum number of parallel threads to use.
- **convergence_manager** (*ConvergenceManager, optional*) – The object to use to monitor and improve convergence.

Returns
The sampling result.

Return type
SamplingResult

Raises
SamplingException – If sampling fails.

Submodules

`pta.commons`

Common definitions used in the PTA package.

Module Contents

`pta.commons.Q`

Type used for describing quantities.

`pta.concentrations_prior`

Definition of the prior distribution of metabolite concentrations.

Module Contents

class `pta.concentrations_prior.`**`ConcentrationsPrior`**(*metabolite_distributions: Dict[Tuple[str, str], Any] = None*, *compartment_distributions: Dict[str, Any] = None*, *default_distribution: Any = default_log_conc*)

Manages prior distributions of concentrations for specific metabolites or compartments. This class provides a distribution for the concentration of any metabolite using (depending on the availability) (1) the distribution for the metabolite itself, (2) the default distribution of the compartment or (3) a default distribution.

Internally, metabolites are stored and accessed MetaNetX compound IDs or, when the compound is not recognized, by metabolite name.

Parameters

- **`metabolite_distributions`** (*`Dict[Tuple[str, str], Any], optional`*) – Mapping from metabolites to their prior. Metabolites are specified as a tuple of metabolite ID and compartment ID.
- **`compartment_distributions`** (*`Dict[str, Any], optional`*) – Mapping from compartment IDs to the prior of the concentration for the compartment. This prior is used for compounds for which no explicit prior is given.
- **`default_distribution`** (*`Any, optional`*) – Prior distribution to use for metabolites and compartments for which no prior is specified.

`property metabolite_distributions: Dict[Tuple[Any, str], Any]`

Gets the prior concentrations for specific metabolites. If the distributions are modified through this property, *update_identifiers()* should be called to make sure that compounds are identified correctly across different namespaces.

`property compartment_distributions: Dict[str, Any]`

Gets the prior concentrations for specific compartments.

`property default_distribution`

Gets the default prior for concentrations.

static load(*prior_file: Union[pathlib.Path, str]*) → *ConcentrationsPrior*

Loads the concentration priors from a .csv file.

Parameters

prior_file (*Union[Path, str]*) – Path to the file containing the parameter values or name of a builtin priors set (any file present in data/concentration_priors/, e.g. 'ecoli_M9_ac').

Returns

The loaded ConcentrationPrior object.

Return type

ConcentrationsPrior

save(*prior_file: Union[pathlib.Path, str]*)

Saves the concentration priors to a .csv file.

Parameters

prior_file (*Union[Path, str]*) – Path to the destination file.

add(*other:* ConcentrationsPrior, *overwrite_metabolite_priors: bool = True*, *overwrite_compartment_priors: bool = True*, *overwrite_default_prior: bool = True*)

Adds the distributions from another prior to this object.

Parameters

- **other** (ConcentrationsPrior) – The concentrations prior from which the distributions have to be copied.
- **overwrite_metabolite_priors** (*bool, optional*) – Specifies whether, in case of duplicates, metabolite distributions in this object should be overwritten or not.
- **overwrite_compartment_priors** (*bool, optional*) – Specifies whether, in case of duplicates, compartment distributions in this object should be overwritten or not.
- **overwrite_default_prior** (*bool, optional*) – Specifies whether the default distribution in this object should be overwritten or not.

get(*metabolite: str*, *compartment: str*) → Any

Gets the distribution for the concentration of a given compound. This uses (depending on the availability) (1) the distribution for the compound itself, (2) the default distribution of the compartment or (3) a default distribution.

Parameters

- **metabolite** (*str*) – Identifier (in any namespace supported by eQuilibrator) of the compound.
- **compartment** (*str*) – Identifier of the compartment.

Returns

Distribution of the concentration of the specified compound.

Return type

Any

update_identifiers()

Updates the internal representation of the compound identifiers.

When possible, this class uses MetaNetX identifiers to represent compounds. This has the advantage that priors can be created and accessed using different namespaces. This function should be called whenever *metabolite_distributions* is modified.

pta.constants

Constants used in tha PTA package.

Module Contents

`pta.constants.default_min_eigenvalue_tds_basis = 0.001`

Default minimum eigenvalue for vector of the basis of the thermodynamic space.

`pta.constants.default_log_conc`

Default distribution of metabolite concentrations.

`pta.constants.default_flux_bound = 1000`

Default magnitude of lower/upper bounds of fluxes.

`pta.constants.default_confidence_level = 0.95`

Default confidence level for PMO problems and TFS

`pta.constants.default_min_drg = 0.1`

Default minimum DrG magnitude for PMO problems and TFS

`pta.constants.default_max_drg = 1000`

Default maximum DrG magnitude for PMO problems and TFS

`pta.constants.non_intracellular_compartment_ids = ['e', 'p']`

Identifiers of non-intracellular compartments.

`pta.constants.default_theta_z = 1.0`

Default threshold on the z-score to consider a predicted value an anomaly.

`pta.constants.default_theta_s = 0.1`

Default threshold on the shadow price to consider a reaction a strong thermodynamic constrain.

`pta.constants.default_max_non_intracellular_conc_mM = 10`

Default threshold on the concentration to consider a predicted concentration an anomaly.

`pta.constants.default_min_chains = 4`

Minimum number of chains to use by default.

`pta.constants.default_num_samples = 1000`

Default number of samples to store.

`pta.constants.us_steps_multiplier = 64`

Coefficient for the number of steps sampling a flux space.

`pta.constants.tfs_steps_multiplier = 100`

Coefficient for the number of steps sampling a thermodynamic space.

`pta.constants.tfs_default_feasibility_cache_size = 10000`

Size of the cache storing whether encountered orthants are feasible or not.

`pta.constants.tfs_default_min_rel_region_length = 1e-06`

Minimum relative size for an orthant to be considered during Hit-and-Run.

`pta.constants.default_max_psrf = 1.1`

Default treshold on the Potential Scale Reduction Factors (PSRFs).

`pta.constants.`**`default_max_threads`**

`pta.constants.`**`min_samples_per_chain = 100`**

Minimum number of samples that must be drawn from each chain in order to obtain a meaningful PSRF score.

`pta.flux_space`

Description of the flux space of a metabolic network.

Module Contents

`class pta.flux_space.`**`FluxSpace`**`(`*S: numpy.ndarray*, *lb: numpy.ndarray*, *ub: numpy.ndarray*, *reaction_ids: List[str]*, *metabolite_ids: List[str]*`)`

Description of the flux space of a metabolic network.

Parameters

- **`S`** (*`np.ndarray`*) – Stoichiometric matrix of the network.
- **`lb`** (*`np.ndarray`*) – Vector of lower bounds on the reaction fluxes.
- **`ub`** (*`np.ndarray`*) – Vector of upper bounds on the reaction fluxes.
- **`reaction_ids`** (*`List[str]`*) – Identifiers of the reactions in the network.
- **`metabolite_ids`** (*`List[str]`*) – Identifiers of the metabolites in the network.

`property S: numpy.ndarray`

Gets the stoichiometric matrix of the network.

`property lb: numpy.ndarray`

Gets the vector of lower bounds on the reaction fluxes.

`property ub: numpy.ndarray`

Gets the vector of upper bounds on the reaction fluxes.

`property reaction_ids: List[str]`

Gets the IDs of the reactions in the flux space.

`property metabolite_ids: List[str]`

Gets the IDs of the metabolites in the flux space.

`property n_metabolites: int`

Gets the number of metabolites in the network.

`property n_reactions: int`

Gets the number of reactions in the network.

`property n_reversible_reactions: int`

Gets the number of reversible reactions in the network.

`copy``()` → *FluxSpace*

Create a copy of this object.

Returns

A copy of this object.

Return type
FluxSpace

`static from_cobrapy_model`(*model: cobra.Model*) → *FluxSpace*

Creates an instance of this class from a cobrapy model.

Parameters
model (*cobra.Model*) – The input model.

Returns
The constructed object.

Return type
FluxSpace

`class pta.flux_space.FluxSpaceBasis`(*flux_space:* FluxSpace)

Bases: `object`

Full-dimensional basis of the flux space. The parametrization is automatically rounded using PolyRound.

`property G: numpy.ndarray`

Gets the left-hand side of the constraints of the flux space.

`property h: numpy.ndarray`

Gets the right-hand side of the constraints of the flux space.

`property to_fluxes_transform: Tuple[numpy.ndarray, numpy.ndarray]`

Gets the transform from a point in the model to a point in the flux space.

`property dimensionality: int`

Gets the dimensionality of the flux space.

`pta.model_assessment`

Methods for the thermodynamic assessment of a metabolic network.

Module Contents

`class pta.model_assessment.StructuralAssessment`(*model: cobra.Model, biomass_id: Optional[str] = None, atpm_id: str = 'ATPM'*)

This class is used to find thermodynamic inconsistencies in the definition of the network that make the model infeasible. Inconsistencies arise when flux constraints (steady state and irreversibilities) prevent any thermodynamically feasible non-zero solution. These inconsistencies occur with any assignment of Gibbs free energies.

Parameters

- **model** (*cobra.Model*) – The target model.
- **biomass_id** (*Optional[str], optional*) – ID of the biomass reactions. If None, the method will try to find it automatically.
- **atpm_id** (*str, optional*) – ID of the ATP maintenance reaction.

`property forced_internal_cycles: List[List[str]]`

Gets a list of all the forced internal cycles of the network, i.e. the internal cycles that must be active in any non-zero flux solution and thus make the model thermodynamically inconsistent.

summary()

Print a summary of the structural assessment.

autoresolve(*model: cobra.Model*, *default_bound: float = default_flux_bound*)

Attempt to automatically resolve thermodynamic inconsistencies. This method removes all reactions involved in dead-end cycles (internal cycles that exchange flux with the rest of the network through a single metabolite) and makes all the reactions in the remaining cycles reversible. While this method should always resolve all inconsistencies, it is still recommended to inspect the inconsistencies and the curated model manually to verify that the model behave as expected. The output of the curation can be seen by enabling logging level INFO.

Parameters

- **model** (*cobra.Model*) – The model in which the curation actions must be applied. This can be the model on which the assessment was run or a similar model. This is useful for cases where assessment is run on a condition specific model but actions should be applied to a base model.
- **default_bound** (*float, optional*) – Default bound to use for unconstrained reactions.

class pta.model_assessment.**QuantitativeAssessment**(*pmo_problem: pta.pmo.PmoProblem*, *z_score_threshold: float = default_theta_z*, *shadow_price_threshold: float = default_theta_s*, *max_extracellular_conc: pta.commons.Q = None*)

Quantitative thermodynamic assessment of a metabolic network. This process combines thermodynamic data and flux constraints to identify parts of the network where the mechanism in the model is possibly incorrect.

Parameters

- **pmo_problem** (PmoProblem) – A solved PMO problem.
- **z_score_threshold** (*float, optional*) – Threshold on the z-score to consider a predicted value an anomaly.
- **shadow_price_threshold** (*float, optional*) – Threshold on the shadow price to consider a reaction a strong thermodynamic constrain.
- **max_extracellular_conc** (*Q, optional*) – Threshold on the concentration to consider a predicted concentration an anomaly.

property metabolites_df: pandas.DataFrame

Gets a data frame with the metabolite-related quantities.

property reactions_df: pandas.DataFrame

Gets a data frame with the reaction-related quantities.

property theta_z: float

Gets the mimimum absolute z-score used to classify a metabolite concentration as anomaly.

property theta_s: float

Gets the mimimum absolute shadow price to flag a constraint.

property max_ex_concentration: float

Gets the minimum concentration used to classify a non-intracellular metabolite concentration as anomaly.

summary()

Print a summary of the quantitative assessment.

`pta.model_assessment.`**`prepare_for_pta`**`(`*model: cobra.Model*, *biomass_id: Optional[str] = None*, *atpm_id: str = 'ATPM'*, *default_bound: float = default_flux_bound*, *autoresolve_inconsistencies: bool = True*, *remove_blocked_reactions: bool = True*, *tighten_bounds: bool = True*, *prevent_loops: bool = False*`)`

Attempt to automatically prepare a model for use in PTA.

This method performs three actions:

- Runs structural assessment on the model and attempts to autoresolve possible inconsistencies.
- Removes all blocked reactions from the model.
- Runs FVA to tighten the flux bounds of each reaction.

Parameters

- **model** (*cobra.Model*) – The target model.
- **biomass_id** (*Optional[str], optional*) – ID of the biomass reactions. If None, the method will try to find it automatically.
- **atpm_id** (*str, optional*) – ID of the ATP maintenance reaction.
- **default_bound** (*float, optional*) – Default bound to use for unconstrained reactions.
- **autoresolve_inconsistencies** (*bool, optional*) – True if the method should attempt to automatically resolve inconsistencies, false otherwise. By default True.
- **remove_blocked_reactions** (*bool, optional*) – True if the method should remove blocked reactions, false otherwise. By default True.
- **tighten_bounds** (*bool, optional*) – True if the method should restrict the flux bounds with FVA, false otherwise. By default True.

pta.pmo

Construction and solution of Probabilistic Metabolic Optimization (PMO) problems.

Module Contents

class `pta.pmo.`**`PmoProblemPool`**`(`*num_processes: Optional[int]*, **argv*`)`

Creation of a process pool for solving multiple PMO problems on the same model.

`map``(`*fn: Callable[[PmoProblem, Any], Any]*, *inputs: Iterable[Any]*`)` → List[Any]

Execute a function on each input element.

This is the same as the regular `map` function, except that it executes in parallel on the available workers.

Parameters

- **fn** (*Callable[[PmoProblem, Any], Any]*) – Function to be applied to each element.
- **inputs** (*Iterable[Any]*) – An iterable containing the input elements.

Returns

A list containing the result of applying the function to each input element.

Return type

List[Any]

`close()`

Wait for all jobs to be done and closes the pool.

`class pta.pmo.PmoProblem(`*network: Union[cobra.Model,* pta.flux_space.FluxSpace*]*, *thermodynamic_space:* pta.thermodynamic_space.ThermodynamicSpace, *thermodynamic_space_basis:* pta.thermodynamic_space.ThermodynamicSpaceBasis *= None*, *objective: Callable[[*PmoProblem*], cvxpy.problems.objective.Objective] = None*, *confidence_level:* float *= default_confidence_level*, *min_drg:* float *= default_min_drg*, *max_drg:* float *= default_max_drg*, *solver: Optional[*str*] = None*, *solver_options:* dict *= None*`)`

Construction and solution of a PMO problem.

Parameters

- **network** (*Union[cobra.Model, FluxSpace]*) – Cobra model or *FluxSpace* object describing the flux space of the metabolic network.
- **thermodynamic_space** (ThermodynamicSpace) – Description of the thermodynamic space of the metabolic network.
- **thermodynamic_space_basis** (*ThermodynamicSpaceBasis, optional*) – A basis for the thermodynamic space. If specified, *m* will be defined in this basis.
- **objective** (*Callable[[PmoProblem], cp.problems.objective.Objective], optional*) – A function used to set the optimization objective. By default the probability of in thermodynamic space is maximized.
- **confidence_level** (*float, optional*) – Confidence level (in the range $[0.0, 1.0[$) on the joint of the thermodynamic variables, by default 0.95.
- **min_drg** (*float, optional*) – Minimum magnitude for the reaction energy of each reaction, by default 1e-1.
- **max_drg** (*float, optional*) – Maximum magnitude for the reaction energy of each reaction, by default 1000.
- **solver** (*Optional[str], optional*) – Name of the solver to use, this can be any of the solvers supported by CVXPY, by default None.
- **solver_options** (*dict, optional*) – Dictionary specifying additional options for the solver.

`property confidence_level`

Gets the confidence level on the thermodynamic space.

`property min_drg: float`

Gets the minimum absolute value of DrG allowed in the model.

`property max_drg: float`

Gets the maximum absolute value of DrG in the model.

`property flux_scale: numpy.ndarray`

Gets the vector of scaling factors for the scaled fluxes.

`property objective: Optional[Callable[[PmoProblem], cvxpy.problems.objective.Objective]]`

Gets the function that constructs the PMO objective.

`property vs: cvxpy.Variable`

Gets the CVXPY variable representing scaled reaction fluxes.

property m: cvxpy.Variable

Gets the CVXPY variable representing the thermodynamic space in the minimal basis.

property d: cvxpy.Variable

Gets the CVXPY variable representing reaction directions.

property v: numpy.array

Gets the predicted reaction fluxes.

property log_c: numpy.ndarray

Gets the predicted log-concentrations. Can be none if concentrations are not represented explicitely.

property drg0: numpy.ndarray

Gets the predicted standard reaction energies. Can be none if standard reaction energies are not represented explicitely.

property drg: numpy.ndarray

Gets the predicted reaction energies. Can be none if reaction energies are not represented explicitely.

property solver: Optional[str]

Gets the selected solver.

property solver_options: Dict[str, Any]

Gets the solver options.

property F: *pta.flux_space.FluxSpace*

Gets the flux space associated with the PMO problem.

property T: *pta.thermodynamic_space.ThermodynamicSpace*

Gets the thermodynamic space associated with the PMO problem.

property B: *pta.thermodynamic_space.ThermodynamicSpaceBasis*

Gets the thermodynamic space basis associated with the PMO problem.

property flux_lb_constraint: cvxpy.constraints.constraint.Constraint

Gets the lower bound constraint for fluxes.

property flux_ub_constraint: cvxpy.constraints.constraint.Constraint

Gets the upper bound constraint for fluxes.

property steady_state_constraint: cvxpy.constraints.constraint.Constraint

Gets the steady state constraint.

property confidence_constraint: cvxpy.constraints.constraint.Constraint

Gets the constraint for the selected confidence level.

property sign_constraints: cvxpy.constraints.constraint.Constraint

Gets the reaction direction constraints. The constraints at indices 0 and 1 are constraints on the reaction energies, while 2 and 3 are constraints on the fluxes.

solve(*verbose=False*) → str

Solves the PMO problem. The result of the optimization is stored inside the class.

Parameters

verbose (*bool, optional*) – True is the solver log should be printed to the console, false otherwise. By default False.

Returns

The CVXPY result of the optimization.

Return type
str

`rebuild_for_directions`(*directions: numpy.ndarray*) → *PmoProblem*

Construct a copy of this PMO problem constrained to the given reaction directions.

Parameters
`directions` (*`np.ndarray`*) – Vector containing the directions (0: backward, 1: forward) of the reactions in T.

Returns
A copy of this problem, constrained to the given directions.

Return type
PmoProblem

`pta.thermodynamic_space`

Description of the space of thermodynamics-related quantities of a metabolic network.

Module Contents

`class pta.thermodynamic_space.ThermodynamicSpace`(*S_constraints: numpy.ndarray*, *reaction_ids: List[str]*, *metabolites: List[enkie.Metabolite]*, *parameters: enkie.CompartmentParameters = None*, *concentrations: pta.concentrations_prior.ConcentrationsPrior = None*, *estimator: enkie.estimators.GibbsEstimatorInterface = None*, *dfg0_estimate: Optional[Tuple[pta.commons.Q, pta.commons.Q]] = None*)

Construction, description and manipulation of the thermodynamic space of a metabolic network.

Parameters

- **`S_constraints`** (*`np.ndarray`*) – Stoichiometric matrix of the reactions covered by thermodynamic constraints.
- **`reaction_ids`** (*`List[str]`*) – Identifiers of the reactions covered by thermodynamic constraints.
- **`metabolites`** (*`List[Metabolite]`*) – List describing the metabolites in the network.
- **`parameters`** (*`CompartmentParameters, optional`*) – The physiological parameters (pH, ionic strength, . . .) of each compartment.
- **`concentrations`** (*`ConcentrationsPrior, optional`*) – Prior distributions for the metabolite concentrations.
- **`estimator`** (*`GibbsEstimatorInterface, optional`*) – Object used to estimate Gibbs free energies.
- **`dfg0_estimate`** (*`Optional[Tuple[Q, Q]], optional`*) – Estimate of formation energies (mean and a square root of the covariance matrix) in case the user wants to specify them manually. This is only used if `estimator` is `None`.

property dfg0_prime_mean: pta.commons.Q

Gets the mean of the corrected standard formation energies.

property dfg0_prime_cov: pta.commons.Q

Gets the covariance of the corrected standard formation energies.

property dfg0_prime_cov_sqrt: pta.commons.Q

Gets a square root of the covariance of the corrected standard formation energies.

property drg0_prime_mean: pta.commons.Q

Gets the mean of the corrected standard reaction energies.

property drg0_prime_cov: pta.commons.Q

Gets the covariance of the corrected standard reaction energies.

property drg0_prime_cov_sqrt: pta.commons.Q

Gets a square root of the covariance of the corrected standard reaction energies.

property log_conc_mean

Gets the mean of the log-concentrations.

property log_conc_cov

Gets the covariance of the log-concentrations.

property dfg_prime_mean

Gets the mean of the formation energies.

property dfg_prime_cov

Gets the covariance of the formation energies.

property drg_prime_mean

Gets the mean of the reaction energies.

property drg_prime_cov

Gets the covariance of the reaction energies.

property parameters: enkie.CompartmentParameters

Gets the compartment parameters of the system.

property metabolites: List[enkie.Metabolite]

Gets the list of metabolites in the thermodynamic space.

property S_constraints: numpy.ndarray

Gets stoichiometric matrix of the reactions with thermodynamic constraints.

property metabolite_ids: List[str]

Gets the IDs of the metabolites in the thermodynamic space.

property reaction_ids: List[str]

Gets the IDs of the reactions with thermodynamic constraints.

subspace(*reaction_ids: Set[str], metabolite_ids: Set[str]*) → *ThermodynamicSpace*

static from_cobrapy_model(*model: cobra.Model, metabolites_namespace: str = None, constrained_rxns: Union[List[int], List[str], cobra.DictList] = None, estimator: enkie.estimators.GibbsEstimatorInterface = None, parameters: enkie.CompartmentParameters = None, concentrations: pta.concentrations_prior.ConcentrationsPrior = None, dfg0_estimate: Optional[Tuple[pta.commons.Q, pta.commons.Q]] = None*) → *ThermodynamicSpace*

Constructs a thermodynamic space from a cobrapy model.

Parameters

- **model** (*cobra.Model*) – Cobra model describing the metabolic network.
- **metabolites_namespace** (*str, optional*) – Specifies the name to use when reading metabolite identifiers from the SBML model annotations.
- **constrained_rxns** (*Union[List[int], List[str], cobra.DictList], optional*) – The reactions that should be modeled with thermodynamic constraints. Usually this list should contain all reactions except biomass and boundary reactions. The list can contain either the reactions themselves, their indices or they identifiers.
- **estimator** (*GibbsEstimatorInterface, optional*) – Object used to estimate Gibbs free energies.
- **parameters** (*CompartmentParameters, optional*) – The physiological parameters (pH, ionic strength, …) of each compartment.
- **concentrations** (*ConcentrationsPrior, optional*) – Prior distributions for the metabolite concentrations.

Returns

The thermodynamic space for the specified model.

Return type

ThermodynamicSpace

`class pta.thermodynamic_space.`**`ThermodynamicSpaceBasis`**(*thermodynamic_space:* ThermodynamicSpace, *explicit_log_conc:* bool *= True*, *explicit_drg0:* bool *= True*, *explicit_drg:* bool *= True*, *min_eigenvalue:* float *= default_min_eigenvalue_tds_basis*)

Full-dimensional basis of the thermodynamic space. It is possible to make only selected variables explicit in the basis, reducing its dimensionality.

Parameters

- **thermodynamic_space** (ThermodynamicSpace) – The target thermodynamic space.
- **explicit_log_conc** (*bool, optional*) – True if log-concentrations should be represented explicitly in the basis, false otherwise. By default True.
- **explicit_drg0** (*bool, optional*) – True if standard reaction energies should be represented explicitly in the basis, false otherwise. By default True.
- **explicit_drg** (*bool, optional*) – True if reaction energies should be represented explicitly in the basis, false otherwise. By default True.
- **min_eigenvalue** (*float, optional*) – Minimum eigenvalue of for a vector to be part of the basis.

`property to_log_conc_transform: Optional[Tuple[numpy.ndarray, numpy.ndarray]]`

Gets the transformation from this space to log-concentrations. `None` if concentrations are not represented explicitly.

`property to_drg0_transform: Optional[Tuple[numpy.ndarray, numpy.ndarray]]`

Gets the transformation from this space to DrG'°. `None` if standard reaction energies are not represented explicitly.

property to_drg_transform: Optional[Tuple[numpy.ndarray, numpy.ndarray]]

Gets the transformation from this space to DrG'. `None` if reaction energies are not represented explicitly.

property to_observables_transform: Tuple[numpy.ndarray, numpy.ndarray]

Gets the transformation from this space to the selected observables.

property observables_ranges: Dict[str, List[int]]

The ranges of the different variables in a vector of observables.

property sigmas: numpy.ndarray

Gets a vector containing the standard deviation of each variable in the minimal basis.

property dimensionality: int

Gets the dimensionality of the basis.

to_log_conc(*basis_vars: numpy.ndarray*) → numpy.ndarray

Transform a vector or matrix in the basis to a vector or matrix of log-concentrations.

Parameters
basis_vars (*np.ndarray*) – The input vector or matrix.

Returns
The transformed vector or matrix. `None` if concentrations are not represented explicitly.

Return type
np.ndarray

to_drg0(*basis_vars: numpy.ndarray*) → numpy.ndarray

Transform a vector or matrix in the basis to a vector or matrix of standard reaction energies.

Parameters
basis_vars (*np.ndarray*) – The input vector or matrix.

Returns
The transformed vector or matrix. `None` if standard reaction energies are not represented explicitly.

Return type
np.ndarray

to_drg(*basis_vars: numpy.ndarray*) → numpy.ndarray

Transform a vector or matrix in the basis to a vector or matrix of reaction energies.

Parameters
basis_vars (*np.ndarray*) – The input vector or matrix.

Returns
The transformed vector or matrix. `None` if reaction energies are not represented explicitly.

Return type
np.ndarray

pta.utils

General utility functions.

Module Contents

pta.utils.**WATER_OR_PROTON_IDS**

pta.utils.**enable_all_logging**(*level: int = logging.INFO*)

Enable detailed logging in PTA and its dependencies. This is useful to debug possible problems.

Parameters
level (*int, optional*) – The desired logging level, by default logging.INFO

pta.utils.**floor**(*value: float*, *decimals: int = 0*) → float

Same as `math.floor`, except that it rounds to a given number of decimals.

Parameters

- **value** (*float*) – Value to round.
- **decimals** (*int, optional*) – Number of decimals to round to, by default 0.

Returns
The rounded value.

Return type
float

pta.utils.**ceil**(*value: float*, *decimals: int = 0*)

Same as `math.ceil`, except that it rounds to a given number of decimals.

Parameters

- **value** (*float*) – Value to round.
- **decimals** (*int, optional*) – Number of decimals to round to, by default 0.

Returns
The rounded value.

Return type
float

pta.utils.**get_internal_reactions**(*model: cobra.Model*) → List[str]

Gets the identifiers of all internal (non boundary) reactions in the model.

Parameters
model (*cobra.Model*) – The target model.

Returns
List of identifiers.

Return type
List[str]

pta.utils.**tighten_model_bounds**(*model: cobra.Model*, *margin: float = 0.0001*, *round_to_digits: int = 6*, *prevent_loops: bool = False*, *fva_result: pandas.DataFrame = None*)

Apply FVA to a model to reduce the range of the flux bounds. The FVA result (plus a margin) is applied to each reaction if it is tighter than the bounds in the model.

Parameters

- **model** (*cobra.Model*) – The model to analyze.
- **margin** (*float, optional*) – Margin to be applied to the FVA result when the computed bounds are tighter than the original ones. This avoids overconstraining the model in case of numerical inaccuracies in the solution. By default 1e-4.
- **round_to_digits** (*int, optional*) – Number of digits to which tighter bounds should be rounded to. This limits the range of the coefficients for optimization problems. By default 6.
- **prevent_loops** (*bool, optional*) – If no FVA solution is provided, determines whether regular or loopless FVA will be used.
- **fva_result** (*pd.DataFrame, optional*) – Precomputed FVA result. If specified, FVA will not be run again, by default None.

pta.utils.**find_blocked_reactions_from_fva_solution**(*model: cobra.Model*, *prevent_loops: bool = False*, *zero_cutoff: Optional[float] = None*, *fva_result: pandas.DataFrame = None*) → List[str]

Similar to cobrapy's find_blocked_reactions(), but optionally takes a precomputed FVA solution as input.

Parameters

- **model** (*cobra.Model*) – The model to analyze.
- **prevent_loops** (*bool, optional*) – If no FVA solution is provided, determines whether regular or loopless FVA will be used.
- **zero_cutoff** (*Optional[float], optional*) – Flux value which is considered to effectively be zero. The default is set to use *model.tolerance* (default None).
- **fva_result** (*pd.DataFrame, optional*) – Precomputed FVA result. If specified, FVA will not be run again, by default None.

Returns
List of identifiers of the blocked reactions.

Return type
List[str]

pta.utils.**get_candidate_thermodynamic_constraints**(*model: cobra.Model*, *metabolites_namespace: str = None*, *exclude_compartments: List[str] = None*) → List[str]

Selects reactions in the flux space that are suitable to be used as thermodynamic constraints. By default this method selects all internal reactions, excluding water transport all boundary and exchange reactions. Optionally, it can exclude reactions where at least one metabolite belongs to certain compartments.

Parameters

- **model** (*Union[FluxSpace, cobra.Model]*) – The target model.
- **metabolite_interpreter** (*MetaboliteInterpreter, optional*) – Specifies how to parse metabolite identifiers.
- **exclude_compartments** (*List[str], optional*) – List of identifiers of the compartment to exclude from the candidates.
- **ccache** (*CompoundCache, optional*) – eQuilibrator's `CompoundCache` object, used to identify compounds using identifiers from different namespaces. For performance reasons, it is recommended to use a single instance of *CompoundCache* for all functions in PTA.

Returns
The identifiers of the candidate reactions.

Return type
List[str]

`pta.utils.`**`get_reactions_in_internal_cycles`**(*model: cobra.Model*, *biomass_name: Optional[str] = None*, *atpm_name: str = 'ATPM'*) → List[str]

Find all the reactions in a model involved in one or more internal cycles.

Parameters

- **`model`** (*`cobra.Model`*) – The target model.
- **`biomass_name`** (*`Optional[str], optional`*) – Identifier of the biomass reaction.
- **`atpm_name`** (*`str, optional`*) – Identifier of the ATP maintenance reaction.

Returns
The identifiers of the reactions involved in internal cycles.

Return type
List[str]

`pta.utils.`**`get_internal_cycles`**(*model: cobra.Model*, *biomass_name: Optional[str] = None*, *atpm_name: str = 'ATPM'*) → numpy.ndarray

Uses `efmtool` to enumerate all the internal cycles in the network.

model
[cobra.Model] The target model.

biomass_name
[Optional[str], optional] Identifier of the biomass reaction.

atpm_name
[str, optional] Identifier of the ATP maintenance reaction.

Returns
A n-by-e matrix, where e in the number of EFMs and n is the number of reactions, where each column is an EFM representing an internal cycle.

Return type
np.ndarray

`pta.utils.`**`to_reactions_idxs`**(*reactions: Union[List[int], List[str], cobra.DictList, List[cobra.Reaction]]*, *model: cobra.Model*) → List[int]

Utility function to obtain a list of reaction indices from different representations.

Parameters

- **`reactions`** (*`Union[List[int], List[str], cobra.DictList, List[cobra.Reaction]]`*) – Input list of reactions. Reactions can be defined through their index in the model, their identifiers, or with the reactions themselves.
- **`model`** (*`cobra.Model`*) – The model in which the reactions are defined.

Returns
List of reaction indices.

Return type
List[int]

Raises
`Exception` – If the list is not of one of the expected formats.

`pta.utils.`**`to_reactions_ids`**(*reactions: Union[List[int], List[str], cobra.DictList, List[cobra.Reaction]], model: cobra.Model*) → List[str]

Utility function to obtain a list of reaction identifiers from different representations.

Parameters

- **`reactions`** (*`Union[List[int], List[str], cobra.DictList, List[cobra.Reaction]]`*) – Input list of reactions. Reactions can be defined through their index in the model, their identifiers, or with the reactions themselves.
- **`model`** (*`cobra.Model`*) – The model in which the reactions are defined.

Returns
List of reaction identifiers.

Return type
List[str]

Raises
`Exception` – If the list is not of one of the expected formats.

`pta.utils.`**`apply_transform`**(*value: numpy.ndarray, transform: Tuple[numpy.ndarray, numpy.ndarray]*) → numpy.ndarray

Applies an affine transform to a matrix.

Parameters

- **`value`** (*`np.ndarray`*) – The matrix to be transformed.
- **`transform`** (*`Tuple[np.ndarray, np.ndarray]`*) – Tuple describing the linear (transformation matrix) and affine (translation vector) of the transform.

Returns
The transformed matrix.

Return type
np.ndarray

`pta.utils.`**`get_path`**(*path: Union[pathlib.Path, str]*) → pathlib.Path

Gets a `Path` object from different representations.

Parameters
`path` (*`Union[Path, str]`*) – A `Path` object or a string describing the path.

Returns
A `Path` object.

Return type
Path

Raises
`Exception` – If the type of the input is not supported.

`pta.utils.`**`covariance_square_root`**(*covariance: numpy.ndarray, min_eigenvalue: float = default_min_eigenvalue_tds_basis*) → numpy.ndarray

Gets a full-rank square root of a covariance matrix.

Parameters

- **`covariance`** (*`np.ndarray`*) – The input covariance matrix.

- `min_eigenvalue` (*float, optional*) – Minimum eigenvalue to keep during the truncated EVD.

Returns
A full-rank square root of the covariance.

Return type
np.ndarray

`pta.utils.`**`load_example_model`**`(`*model_name: str*`)` → cobra.Model

Load an example SBML model in cobrapy.

Parameters
`model_name` (*str*) – Name of the model file (without extension). This can be any of the models in pta/data/models.

Returns
The loaded model in cobrapy format.

Return type
cobra.Model

`pta.utils.`**`find_rotation_matrix`**`(`*x: numpy.ndarray*, *y: numpy.ndarray*`)` → numpy.ndarray

Compute a matrix that rotates the vector x onto vector y (without scaling).

Parameters

- **`x`** (*np.ndarray*) – The starting vector.
- **`y`** (*np.ndarray*) – The destination vector.

Returns
The computed rotation matrix.

Return type
np.ndarray

CHAPTER

TWO

REFERENCES

PYTHON MODULE INDEX

INDEX