pytfa Documentation

Release 0.9.2

pytfa Team

Nov 01, 2021

CONTENTS:

TFA is a method that builds on FBA to improve its solution space. Specifically, it includes thermodynamics and explicit formulation of Gibbs energies and metabolite concentrations, which enables straightforward integration of metabolite concentration measurements.

If you use our work, please cite us[1] !

[1] Salvy, P., Fengos, G., Ataman, M., Pathier, T., Soh, K. C., & Hatzimanikatis, V. (2018). pyTFA and matTFA: a Python package and a Matlab toolbox for Thermodynamics-based Flux Analysis. Bioinformatics, 35(1), 167-169.

CHAPTER

ONE

QUICK START

Three tutorial files detail thoroughly normal usages of the pytfa package. They can be found at:

```
pytfa
└── tutorials
    ├── figure_paper.py
    ├── tutorial_basics.py
    └── tutorial_sampling.py
```

figure_paper.py details how to get the figure from our paper[1], a simple use case for TFA on a reduced *Escherichia coli*. We show that adding thermodyamics constraints and simple concentration data allow to substantially reduce the flux space.

tutorial_basics.py shows a more realistic case with two models (reduced or full genome-scale) of *Escherichia coli*. It also cycles through several solvers (if more are installed), to show how simple it is to change your solver (thanks to optlang).

tutorial_sampling.py shows how to sample a variable, for example thermodynamic displacement, and generate plots to visualize the results.

If you plan to run the tutorials with full genome-scale models, we recommend you to get a commercial solver, as it has been seen that GLPK's lack of parallelism significantly increases solving time

The next sections give more details on how the thermodynamic model is structured, and how data is managed.

Cheers,

The py.TFA team

[1] Salvy, P., Fengos, G., Ataman, M., Pathier, T., Soh, K. C., & Hatzimanikatis, V. (2018). pyTFA and matTFA: a Python package and a Matlab toolbox for Thermodynamics-based Flux Analysis. Bioinformatics, 35(1), 167-169.

CHAPTER

TWO

PYTFA MODELS

pyTFA models are based on `COBRApy models`, with additional values.

2.1 Compartment data

This is the `compartments` attribute of the model. It is a `dict` where each key is the symbol of a compartment, and the value is another `dict` with the following keys :

c_min	`float` The minimum concentration for each metabolite in the compartment, in mol.L-1
c_max	`float` The maximum concentration for each metabolite in the compartment, in mol.L-1
ionicStr	`float` The ionic strength of the compartment (mV)
membranePot	`dict` A dictionnary representing the membrane potential between this compartment (which is the source compartment) and the others. Each key is the symbol of another compartment (which is the destination compartment), and the value is the potential (in mV) from the source to the destination.
name	`string` The name of the compartment
pH	`float` The pH in the compartment
symbol	`string` The symbol of the compartment (which is the key of this dictionnary)

Here is an example:

```
cobra_model.compartments['c'] = {
  'c_max': 0.01,
  'c_min': 9.9999999999999995e-08,
  'ionicStr': 0.25,
  'membranePot': {
    'c': 0,
    'e': 60,
    'g': 0,
    'm': -180,
    'n': 0,
    'p': 0,
    'r': 0,
    't': 0,
    'v': 0,
    'x': 0
  },
```

(continues on next page)

(continued from previous page)

```
  'name': 'Cytosol',
  'pH': 7.0,
  'symbol': 'c'
}
```

2.2 Metabolites

Each metabolite must be annotated with its `SeedID`, which will be used to get the thermodynamic values from the *Thermodynamic Databases*. In order to do this, use the `annotation` attribute of each `metabolite`. Here is an example:

```
cobra_model.metabolites[0].annotation = {
  'SeedID': 'cpd00018'
}
```

pyTFA will also define a `thermo` a thermo attribute for each metabolite, which is a *pytfa.thermo.MetaboliteThermo*.

2.3 Reactions

pyTFA will define a `thermo` attribute for each reaction. It is a `dict` with the following attributes:

computed	`bool` Whether the thermodynamic values were computed or not.
deltaGR	`float` Sum of the non-concentration terms for the reaction. Used as the right-hand side value of a constraint. If the thermodynamic values were not computed, this is `10**7`.
deltaGRerr	`float` Error on `deltaGR` If the thermodynamic values were not computed, this is `10**7`.
deltaGrxn	`float` Sum of the deltaGF of all the metabolites in the reaction. **Not defined if computed is False !**
isTrans	`bool` Whether the reaction is a transport reaction or not

Here are some examples:

```
cobra_model.reactions[0].thermo = {
  'computed': False,
  'deltaGR': 10000000,
  'deltaGRerr': 10000000,
  'isTrans': False
}

cobra_model.reactions[99].thermo = {
  'computed': True,
  'deltaGR': 1.161097833014658,
  'deltaGRerr': 2,
  'deltaGrxn': 0,
```

(continues on next page)

(continued from previous page)

```
  'isTrans': True,
}
```

CHAPTER

THREE

THERMODYNAMIC DATABASES

3.1 Converting a Matlab thermodynamic database

If you have a Matlab thermodynamic database, you can easily convert it to a Python database thanks to the script `thermoDBconverter.py`:

```
python thermoDBconverter.py database.mat converted_database.thermodb
```

3.2 Loading a thermodynamic database

Thermodynamic databases are compressed through `zlib` and binary-encoded with `pickle`. In order to load them, you need first to uncompress them with `zlib.decompress` then load the result into memory with `pickle.loads`:

```
import pickle
import zlib
with open('thermoDatabases/DB_AlbertyUpdate.thermodb', 'rb') as file:
    ReactionDB = pickle.loads(zlib.decompress(file.read()))
```

Warning: Since the file is compressed, you **MUST** load it as a binary file by calling `open` with the `b` flag, otherwise Python will try to decode it as unicode and raise an exception !

3.3 Structure of a thermodynamic database

A thermodynamic database is a `dict` with the following fields:

- `name` : `string` The name of the database
- `units` : `string` The unit of the energies in the database. Can be `kcal/mol` or `kJ/mol`.
- `metabolites` : `dict` A dictionnary containing the metabolites' thermodynamic data. See *Metabolites* for more information.
- `cues` : `dict` A dictionnary containing the cues' thermodynamic data. See *Cues* for more information.

3.3.1 Metabolites

This is a dictionnary storing various thermodynamic data about metabolites. It is stored as a `dict` where each key is a `SeedID`. The values are others `dict` with the following keys.

id	`string` SeedID of the metabolite
charge_std	`float` Charge of the metabolite (mV) in standard conditions
deltaGf_std	`float` Transformed Gibbs energy of formation of the metabolite, in standard conditions.
deltaGf_err	`float` Error on the transformed Gibbs energy of formation of the metabolite, in standard conditions
mass_std	`float` Mass of the metabolite (g.mol-1)
nH_std	`int` Number of protons of the metabolite, in standard conditions
error	`string` Error on the metabolite's thermodynamic data. Thermodynamic values will be computed only if this equals to 'Nil'.
formula	`string` Formula of the metabolite.
nH_std	`int` Number of protons in the metabolite's formula
name	`string` Name of the metabolite
other_names	`list` (`string`) Other names of the metabolite
pKa	`list` (`float`) pKas of the metabolite
struct_cues	`dict` (`int`) cues of the metabolite The keys of the array are the names of the cues, and the values the number of cues of this type that are part of the structure.

Here is an example:

```
ReactionDB['metabolites']['cpd00001'] = {
  'charge_std': 0,
  'deltaGf_err': 0.5,
  'deltaGf_std': -56.686999999999998,
  'error': 'Nil',
  'formula': 'H2O',
  'id': 'cpd00001',
  'mass_std': 18.0,
  'nH_std': 2,
  'name': 'H2O',
  'other_names': ['H2O', 'Water', 'HO-', 'OH-', 'h2o'],
  'pKa': [15.7],
  'struct_cues': {'H2O': 1}
}
```

3.3.2 Cues

This is a dictionnary storing various thermodynamic data about cues. It is stored as a `dict` where each key is the cue ID, as referrenced in the `struct_cues` attribute of *Metabolites*. The values are others `dict` with the following keys.

id	`string` ID of the cue
charge	`float` The charge (mV) of the cue in standard conditions
datfile	`string` The dat file from which the data was imported. *Optional*
energy	`float` Transformed Gibbs energy of formation of the cue, in standard conditions.
error	`float` The error on the transformed Gibbs energy of formation of the cue, in standard conditions.
formula	`string` Formula of the cue
names	`list` (`string`) Other names of the cue
small	`bool` Whether this is a small cue or not

Here is an example:

```
ReactionDB['cues']['H2O'] = {
  'charge': 0,
  'datfile': 'H2O.gds',
  'energy': -56.686999999999998,
  'error': 0.5,
  'formula': 'H2O',
  'id': 'H2O',
  'names': ['H2O', 'OH-', 'HO-'],
  'small': True
}
```

CHAPTER

FOUR

SOLVER SETUP

This document is written assuming a Docker container installation. However, you can easily adapt the content to other types of Linux-based installations.

4.1 GPLK

GLPK should be directly available from the requirements.

4.2 CPLEX

You will need to first install CPLEX on a Linux machine.

Place in *etfl/docker/solvers/* the folder */ibm* that is installed by CPLEX (usually in */opt*). You actually only need the following substructure (makes the container lighter):

```
.
└── ibm
    └── ILOG
        └── CPLEX_StudioXXXX
            └── cplex
                ├── bin
                ├── include
                ├── lib
                └── python
```

4.3 Gurobi

Place in *etfl/docker/solvers/* the tarball you downloaded from the website, and modify accordingly the files:

```
../utils/install_gurobi.sh
../utils/activate_gurobi.sh
```

Make sure you change the paths and filenames to reflect the actual version of Gurobi you are running.

Gurobi needs a floating license for Docker instances, (see http://www.gurobi.com/documentation/7.5/quickstart_windows/setting_up_and_using_a_flo.html#subsection:tokenserver) Once your system administrator set it up, you will need to add your gurobi license server to ../utils/gurobi.lic.template, and rename it to gurobi.lic

CHAPTER

FIVE

INTEGRATING METABOLOMICS

In this short example we will go through a simple case of integration of absolute metabolite concentrations.

Let us imagine we got absolute concentration values for cytosolic ATP:

$$5.10^{-3}\ mol.L^{-1} \leq [X] \leq 3.10^{-2}\ mol.L^{-1}$$

Then you can tell the model that your (log) concentration is limited in range:

```
from math import log

mymodel.log_concentration.atp_c.variable.lb = log(5e-3)
mymodel.log_concentration.atp_c.variable.ub = log(3e-2)
```

This will constrain the dG according to your concentration measurements for cytosolic ATP. As a reminder, the dG (not the dGo) takes activity (here, concentrations) into account for its calculation. You can find a more detailed explanation in those papers:

- Henry, Christopher S., Linda J. Broadbelt, and Vassily Hatzimanikatis. "Thermodynamics-based metabolic flux analysis." Biophysical journal 92.5 (2007): 1792-1805.
- Soh, Keng Cher, Ljubisa Miskovic, and Vassily Hatzimanikatis. "From network models to network responses: integration of thermodynamic and kinetic properties of yeast genome-scale metabolic networks." FEMS yeast research 12.2 (2012): 129-143.

CHAPTER

SIX

API REFERENCE

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

6.1 pytfa

Thermodynamic analysis for Flux-Based Analysis

6.1.1 Subpackages

pytfa.analysis

Submodules

pytfa.analysis.chebyshev

Variability analysis

Module Contents

Classes

ChebyshevRadius	Variable representing a Chebyshev radius

Functions

is_inequality(constraint)	
chebyshev_center(model, variables, inplace=False, big_m=BIGM, include=list(), exclude=list())	Computes the chebyshev center of a problem with respect to given variables,
chebyshev_transform(model, vars, include_list=list(), exclude_list=list(), radius_id='radius', scaling_factor=1, big_m=BIGM)	Adds a Chebyshev radius variable and edits accordingly the selected

continues on next page

[1] Created with sphinx-autoapi

Table 2 – continued from previous page

`get_cons_var_classes`(model, elements, type)
`get_variables`(model, variables)

Attributes

`BIGM`

pytfa.**BIGM = 1000**

class pytfa.**ChebyshevRadius**(*model*, *id_*, ***kwargs*)

Bases: `pytfa.optim.variables.ModelVariable`

Variable representing a Chebyshev radius

prefix = CR_

pytfa.**is_inequality**(*constraint*)

pytfa.**chebyshev_center**(*model*, *variables*, *inplace=False*, *big_m=BIGM*, *include=list()*, *exclude=list()*)

Computes the chebyshev center of a problem with respect to given variables, including `include' constraints and excluding `exclude' constraints. *Warning: Only works with pyTFA variables so far*

Parameters

- **model** –
- **variables** –
- **inplace** –
- **big_m** –

Returns

pytfa.**chebyshev_transform**(*model*, *vars*, *include_list=list()*, *exclude_list=list()*, *radius_id='radius'*, *scaling_factor=1*, *big_m=BIGM*)

Adds a Chebyshev radius variable and edits accordingly the selected constraints

Parameters

- **model** –
- **vars** – variables with respect to which to perform the Chebyshev centering. If none is supplied, all of the variables in the equation will be considered
- **include_list** –
- **exclude_list** –
- **radius_id** –
- **big_m** –

Returns

pytfa.**get_cons_var_classes**(*model*, *elements*, *type*)

pytfa.**get_variables**(*model*, *variables*)

pytfa.analysis.manipulation

Module Contents

Functions

apply_reaction_variability(tmodel, va, inplace=True)	Applies the VA results as bounds for the reactions of a cobra_model
apply_generic_variability(tmodel, va, inplace=True)	Reactions a dealt with cobra, but the other variables added use pytfa's
apply_directionality(tmodel, solution, inplace=True)	Takes a flux solution and transfers its reaction directionality as

pytfa.analysis.manipulation.**apply_reaction_variability**(*tmodel*, *va*, *inplace=True*)
Applies the VA results as bounds for the reactions of a cobra_model :param inplace: :param tmodel: :param va: :return:

pytfa.analysis.manipulation.**apply_generic_variability**(*tmodel*, *va*, *inplace=True*)
Reactions a dealt with cobra, but the other variables added use pytfa's interface: the class GenericVariable. We use a different method to apply variability directly in the solver

Parameters

- **tmodel** –
- **va** –
- **inplace** –

Returns

pytfa.analysis.manipulation.**apply_directionality**(*tmodel*, *solution*, *inplace=True*)
Takes a flux solution and transfers its reaction directionality as constraints for the cobra_model

Parameters

- **inplace** –
- **tmodel** –
- **solution** –

Returns

pytfa.analysis.sampling

Sampling wrappers for pytfa models

Module Contents

Classes

GeneralizedHRSampler	The abstract base class for hit-and-run samplers.
GeneralizedACHRSampler	The abstract base class for hit-and-run samplers.
GeneralizedOptGPSampler	The abstract base class for hit-and-run samplers.

Functions

sample(model, n, method='optgp', thinning=100, processes=1, seed=None)	Sample valid flux distributions from a thermo cobra_model.

class pytfa.**GeneralizedHRSampler**(*model*, *thinning*, *nproj=None*, *seed=None*)

Bases: cobra.sampling.HRSampler

The abstract base class for hit-and-run samplers.

New samplers should derive from this class where possible to provide a uniform interface.

model [cobra.Model] The cobra model from which to generate samples.

thinning [int] The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

nproj [int > 0, optional] How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number (default None).

seed [int > 0, optional] Sets the random number seed. Initialized to the current time stamp if None (default None).

feasibility_tol: float The tolerance used for checking equalities feasibility.

bounds_tol: float The tolerance used for checking bounds feasibility.

n_samples [int] The total number of samples that have been generated by this sampler instance.

retries [int] The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

problem [Problem] A NamedTuple whose attributes define the entire sampling problem in matrix form.

warmup [numpy.matrix] A numpy matrix with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

fwd_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective forward variable.

rev_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective reverse variable.

generate_fva_warmup(*self*)

Generate the warmup points for the sampler.

Generates warmup points by setting each flux as the sole objective and minimizing/maximizing it. Also caches the projection of the warmup points into the nullspace for non-homogeneous problems (only if necessary).

class pytfa.**GeneralizedACHRSampler**(*model*, *thinning=100*, *seed=None*)

Bases: *GeneralizedHRSampler*, cobra.sampling.ACHRSampler

The abstract base class for hit-and-run samplers.

New samplers should derive from this class where possible to provide a uniform interface.

model [cobra.Model] The cobra model from which to generate samples.

thinning [int] The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

nproj [int > 0, optional] How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number (default None).

seed [int > 0, optional] Sets the random number seed. Initialized to the current time stamp if None (default None).

feasibility_tol: float The tolerance used for checking equalities feasibility.

bounds_tol: float The tolerance used for checking bounds feasibility.

n_samples [int] The total number of samples that have been generated by this sampler instance.

retries [int] The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

problem [Problem] A NamedTuple whose attributes define the entire sampling problem in matrix form.

warmup [numpy.matrix] A numpy matrix with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

fwd_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective forward variable.

rev_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective reverse variable.

class pytfa.**GeneralizedOptGPSampler**(*model*, *processes*, *thinning=100*, *seed=None*)

Bases: *GeneralizedHRSampler*, cobra.sampling.OptGPSampler

The abstract base class for hit-and-run samplers.

New samplers should derive from this class where possible to provide a uniform interface.

model [cobra.Model] The cobra model from which to generate samples.

thinning [int] The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

nproj [int > 0, optional] How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number (default None).

seed [int > 0, optional] Sets the random number seed. Initialized to the current time stamp if None (default None).

feasibility_tol: float The tolerance used for checking equalities feasibility.

bounds_tol: float The tolerance used for checking bounds feasibility.

n_samples [int] The total number of samples that have been generated by this sampler instance.

retries [int] The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

problem [Problem] A NamedTuple whose attributes define the entire sampling problem in matrix form.

warmup [numpy.matrix] A numpy matrix with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

fwd_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective forward variable.

rev_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective reverse variable.

pytfa.**sample**(*model*, *n*, *method='optgp'*, *thinning=100*, *processes=1*, *seed=None*)

Sample valid flux distributions from a thermo cobra_model.

Function adapted from cobra.flux_analysis.sample to display all solver variables

Documentation adapted from cobra.flux_analysis.sample

1. **'optgp' (default) which uses the OptGPSampler that supports parallel** analysis[1]. Requires large numbers of samples to be performant (n < 1000). For smaller samples 'achr' might be better suited.

or

2. 'achr' which uses artificial centering hit-and-run. This is a single process method with good convergence[2].

model [pytfa.core.ThermoModel] The cobra_model from which to sample variables.

n [int] The number of samples to obtain. When using 'optgp' this must be a multiple of *processes*, otherwise a larger number of samples will be returned.

method [str, optional] The analysis algorithm to use.

thinning [int, optional] The thinning factor of the generated analysis chain. A thinning of 10 means samples are returned every 10 steps. Defaults to 100 which in benchmarks gives approximately uncorrelated samples. If set to one will return all iterates.

processes [int, optional] Only used for 'optgp'. The number of processes used to generate samples.

seed [positive integer, optional] The random number seed to be used. Initialized to current time stamp if None.

pandas.DataFrame The generated flux samples. Each row corresponds to a sample of the fluxes and the columns are the reactions.

The samplers have a correction method to ensure equality feasibility for long-running chains, however this will only work for homogeneous models, meaning models with no non-zero fixed variables or constraints (right-hand side of the equalities are zero).

[1] Megchelenbrink W, Huynen M, Marchiori E (2014) optGpSampler: An Improved Tool for Uniformly Sampling the Solution-Space of Genome-Scale Metabolic Networks. PLoS ONE 9(2): e86587.

[2] Direction Choice for Accelerated Convergence in Hit-and-Run Sampling David E. Kaufman Robert L. Smith Operations Research 199846:1 , 84-95

pytfa.analysis.variability

Variability analysis

Module Contents

Functions

find_bidirectional_reactions(va, tolerance=1e-08)	Returns the ids of reactions that can both carry net flux in the forward or
find_directionality_profiles(tmodel, bidirectional, max_iter=10000.0, solver='optlang-glpk', tolerance=1e-09)	Takes a ThermoModel and performs enumeration of the directionality profiles
_bool2str(bool_list)	turns a list of booleans into a string
_variability_analysis_element(tmodel, var, sense)	
variability_analysis(tmodel, kind='reactions', proc_num=BEST_THREAD_RATIO)	Performs variability analysis, gicven a variable type
parallel_variability_analysis(tmodel, kind='reactions', proc_num=BEST_THREAD_RATIO)	WIP.
calculate_dissipation(tmodel, solution=None)	

Attributes

CPU_COUNT
BEST_THREAD_RATIO

pytfa.**CPU_COUNT**

pytfa.**BEST_THREAD_RATIO**

pytfa.**find_bidirectional_reactions**(*va*, *tolerance=1e-08*)

Returns the ids of reactions that can both carry net flux in the forward or backward direction.

Parameters **va** –

A variability analysis, pandas Dataframe like so: maximum minimum

6PGLter -8.330667e-04 -8.330667e-04 ABUTt2r 0.000000e+00 0.000000e+00 ACALDt 0.000000e+00 0.000000e+00

Returns

pytfa.**find_directionality_profiles**(*tmodel*, *bidirectional*, *max_iter=10000.0*, *solver='optlang-glpk'*, *tolerance=1e-09*)

Takes a ThermoModel and performs enumeration of the directionality profiles

Parameters

- **tmodel** –

- **max_iter** –

Returns

pytfa.**_bool2str**(*bool_list*)
turns a list of booleans into a string

Parameters **bool_list** – ex: '[False True False False True]'

Returns '01001'

pytfa.**_variability_analysis_element**(*tmodel*, *var*, *sense*)

pytfa.**variability_analysis**(*tmodel*, *kind='reactions'*, *proc_num=BEST_THREAD_RATIO*)
Performs variability analysis, gicven a variable type

Parameters

- **tmodel** –
- **kind** –
- **proc_num** –

Returns

pytfa.**parallel_variability_analysis**(*tmodel*, *kind='reactions'*, *proc_num=BEST_THREAD_RATIO*)
WIP.

Parameters

- **tmodel** –
- **kind** –
- **proc_num** –

Returns

pytfa.**calculate_dissipation**(*tmodel*, *solution=None*)

Package Contents

Classes

GeneralizedHRSampler	The abstract base class for hit-and-run samplers.
GeneralizedACHRSampler	The abstract base class for hit-and-run samplers.
GeneralizedOptGPSampler	The abstract base class for hit-and-run samplers.
DeltaG	Class to represent a DeltaG
ForbiddenProfile	Class to represent a forbidden net flux directionality profile
ForwardUseVariable	Class to represent a forward use variable, a type of binary variable used to

Functions

sample(model, n, method='optgp', thinning=100, processes=1, seed=None)	Sample valid flux distributions from a thermo cobra_model.
get_direction_use_variables(tmodel, solution)	Returns the active use variables in a solution. Use Variables are binary
get_active_use_variables(tmodel, solution)	Returns the active use variables in a solution. Use Variables are binary
get_bistream_logger(name)	Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages
find_bidirectional_reactions(va, tolerance=1e-08)	Returns the ids of reactions that can both carry net flux in the forward or
find_directionality_profiles(tmodel, bidirectional, max_iter=10000.0, solver='optlang-glpk', tolerance=1e-09)	Takes a ThermoModel and performs enumeration of the directionality profiles
_bool2str(bool_list)	turns a list of booleans into a string
_variability_analysis_element(tmodel, var, sense)	
variability_analysis(tmodel, kind='reactions', proc_num=BEST_THREAD_RATIO)	Performs variability analysis, gicven a variable type
parallel_variability_analysis(tmodel, kind='reactions', proc_num=BEST_THREAD_RATIO)	WIP.
calculate_dissipation(tmodel, solution=None)	
apply_reaction_variability(tmodel, va, inplace=True)	Applies the VA results as bounds for the reactions of a cobra_model
apply_generic_variability(tmodel, va, inplace=True)	Reactions a dealt with cobra, but the other variables added use pytfa's
apply_directionality(tmodel, solution, inplace=True)	Takes a flux solution and transfers its reaction directionality as

Attributes

CPU_COUNT
BEST_THREAD_RATIO

class pytfa.analysis.**GeneralizedHRSampler**(*model*, *thinning*, *nproj=None*, *seed=None*)

Bases: cobra.sampling.HRSampler

The abstract base class for hit-and-run samplers.

New samplers should derive from this class where possible to provide a uniform interface.

model [cobra.Model] The cobra model from which to generate samples.

thinning [int] The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

nproj [int > 0, optional] How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate*

you should lower this number (default None).

seed [int > 0, optional] Sets the random number seed. Initialized to the current time stamp if None (default None).

feasibility_tol: float The tolerance used for checking equalities feasibility.

bounds_tol: float The tolerance used for checking bounds feasibility.

n_samples [int] The total number of samples that have been generated by this sampler instance.

retries [int] The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

problem [Problem] A NamedTuple whose attributes define the entire sampling problem in matrix form.

warmup [numpy.matrix] A numpy matrix with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

fwd_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective forward variable.

rev_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective reverse variable.

`generate_fva_warmup`(*self*)

Generate the warmup points for the sampler.

Generates warmup points by setting each flux as the sole objective and minimizing/maximizing it. Also caches the projection of the warmup points into the nullspace for non-homogeneous problems (only if necessary).

`class pytfa.analysis.GeneralizedACHRSampler`(*model*, *thinning=100*, *seed=None*)

Bases: *GeneralizedHRSampler*, `cobra.sampling.ACHRSampler`

The abstract base class for hit-and-run samplers.

New samplers should derive from this class where possible to provide a uniform interface.

model [cobra.Model] The cobra model from which to generate samples.

thinning [int] The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

nproj [int > 0, optional] How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number (default None).

seed [int > 0, optional] Sets the random number seed. Initialized to the current time stamp if None (default None).

feasibility_tol: float The tolerance used for checking equalities feasibility.

bounds_tol: float The tolerance used for checking bounds feasibility.

n_samples [int] The total number of samples that have been generated by this sampler instance.

retries [int] The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

problem [Problem] A NamedTuple whose attributes define the entire sampling problem in matrix form.

warmup [numpy.matrix] A numpy matrix with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

fwd_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective forward variable.

rev_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective reverse variable.

class pytfa.analysis.**GeneralizedOptGPSampler**(*model*, *processes*, *thinning=100*, *seed=None*)

Bases: *GeneralizedHRSampler*, cobra.sampling.OptGPSampler

The abstract base class for hit-and-run samplers.

New samplers should derive from this class where possible to provide a uniform interface.

model [cobra.Model] The cobra model from which to generate samples.

thinning [int] The thinning factor of the generated sampling chain. A thinning of 10 means samples are returned every 10 steps.

nproj [int > 0, optional] How often to reproject the sampling point into the feasibility space. Avoids numerical issues at the cost of lower sampling. If you observe many equality constraint violations with *sampler.validate* you should lower this number (default None).

seed [int > 0, optional] Sets the random number seed. Initialized to the current time stamp if None (default None).

feasibility_tol: float The tolerance used for checking equalities feasibility.

bounds_tol: float The tolerance used for checking bounds feasibility.

n_samples [int] The total number of samples that have been generated by this sampler instance.

retries [int] The overall of sampling retries the sampler has observed. Larger values indicate numerical instabilities.

problem [Problem] A NamedTuple whose attributes define the entire sampling problem in matrix form.

warmup [numpy.matrix] A numpy matrix with as many columns as reactions in the model and more than 3 rows containing a warmup sample in each row. None if no warmup points have been generated yet.

fwd_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective forward variable.

rev_idx [numpy.array] A numpy array having one entry for each reaction in the model, containing the index of the respective reverse variable.

pytfa.analysis.**sample**(*model*, *n*, *method='optgp'*, *thinning=100*, *processes=1*, *seed=None*)

Sample valid flux distributions from a thermo cobra_model.

Function adapted from cobra.flux_analysis.sample to display all solver variables

Documentation adapted from cobra.flux_analysis.sample

1. **'optgp' (default) which uses the OptGPSampler that supports parallel** analysis[1]. Requires large numbers of samples to be performant (n < 1000). For smaller samples 'achr' might be better suited.

or

2. 'achr' which uses artificial centering hit-and-run. This is a single process method with good convergence[2].

[1] Megchelenbrink W, Huynen M, Marchiori E (2014) optGpSampler: An Improved Tool for Uniformly Sampling the Solution-Space of Genome-Scale Metabolic Networks. PLoS ONE 9(2): e86587.

[2] Direction Choice for Accelerated Convergence in Hit-and-Run Sampling David E. Kaufman Robert L. Smith Operations Research 199846:1 , 84-95

model [pytfa.core.ThermoModel] The cobra_model from which to sample variables.

n [int] The number of samples to obtain. When using 'optgp' this must be a multiple of *processes*, otherwise a larger number of samples will be returned.

method [str, optional] The analysis algorithm to use.

thinning [int, optional] The thinning factor of the generated analysis chain. A thinning of 10 means samples are returned every 10 steps. Defaults to 100 which in benchmarks gives approximately uncorrelated samples. If set to one will return all iterates.

processes [int, optional] Only used for 'optgp'. The number of processes used to generate samples.

seed [positive integer, optional] The random number seed to be used. Initialized to current time stamp if None.

pandas.DataFrame The generated flux samples. Each row corresponds to a sample of the fluxes and the columns are the reactions.

The samplers have a correction method to ensure equality feasibility for long-running chains, however this will only work for homogeneous models, meaning models with no non-zero fixed variables or constraints (right-hand side of the equalities are zero).

class pytfa.analysis.**DeltaG**(*reaction*, ***kwargs*)
Bases: ReactionVariable

Class to represent a DeltaG

prefix = DG_

class pytfa.analysis.**ForbiddenProfile**(*model*, *expr*, *id_*, ***kwargs*)
Bases: GenericConstraint

Class to represent a forbidden net flux directionality profile Looks like: FU_rxn_1 + BU_rxn_2 + … + FU_rxn_n <= n-1

prefix = FP_

pytfa.analysis.**get_direction_use_variables**(*tmodel*, *solution*)
Returns the active use variables in a solution. Use Variables are binary variables that control the directionality of the reaction The difference with get_active_use_variables is that variables with both UseVariables at 0 will return as going forwards. This is to ensure that the output size of the function is equal to the number of FDPs

ex: FU_ACALDt BU_PFK

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **solution** –

Returns

pytfa.analysis.**get_active_use_variables**(*tmodel*, *solution*)
Returns the active use variables in a solution. Use Variables are binary variables that control the directionality of the reaction

ex: FU_ACALDt BU_PFK

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **solution** –

Returns

class pytfa.analysis.**ForwardUseVariable**(*reaction*, ***kwargs*)
Bases: ReactionVariable, BinaryVariable

Class to represent a forward use variable, a type of binary variable used to enforce forward directionality in reaction net fluxes

prefix = FU_

pytfa.analysis.**get_bistream_logger**(*name*)
Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages in the log file

Parameters **name** – a class __name__ attribute

Returns

pytfa.analysis.**CPU_COUNT**

pytfa.analysis.**BEST_THREAD_RATIO**

pytfa.analysis.**find_bidirectional_reactions**(*va*, *tolerance=1e-08*)
Returns the ids of reactions that can both carry net flux in the forward or backward direction.

Parameters **va** –

A variability analysis, pandas Dataframe like so: maximum minimum

6PGLter -8.330667e-04 -8.330667e-04 ABUTt2r 0.000000e+00 0.000000e+00 ACALDt 0.000000e+00 0.000000e+00

Returns

pytfa.analysis.**find_directionality_profiles**(*tmodel*, *bidirectional*, *max_iter=10000.0*, *solver='optlang-glpk'*, *tolerance=1e-09*)
Takes a ThermoModel and performs enumeration of the directionality profiles

Parameters

- **tmodel** –
- **max_iter** –

Returns

pytfa.analysis.**_bool2str**(*bool_list*)
turns a list of booleans into a string

Parameters **bool_list** – ex: '[False True False False True]'

Returns '01001'

pytfa.analysis.**_variability_analysis_element**(*tmodel*, *var*, *sense*)

pytfa.analysis.**variability_analysis**(*tmodel*, *kind='reactions'*, *proc_num=BEST_THREAD_RATIO*)
Performs variability analysis, gicven a variable type

Parameters

- **tmodel** –
- **kind** –
- **proc_num** –

Returns

pytfa.analysis.**parallel_variability_analysis**(*tmodel*, *kind='reactions'*, *proc_num=BEST_THREAD_RATIO*)

WIP.

Parameters

- **tmodel** –
- **kind** –
- **proc_num** –

Returns

pytfa.analysis.**calculate_dissipation**(*tmodel*, *solution=None*)

pytfa.analysis.**apply_reaction_variability**(*tmodel*, *va*, *inplace=True*)

Applies the VA results as bounds for the reactions of a cobra_model :param inplace: :param tmodel: :param va: :return:

pytfa.analysis.**apply_generic_variability**(*tmodel*, *va*, *inplace=True*)

Reactions a dealt with cobra, but the other variables added use pytfa's interface: the class GenericVariable. We use a different method to apply variability directly in the solver

Parameters

- **tmodel** –
- **va** –
- **inplace** –

Returns

pytfa.analysis.**apply_directionality**(*tmodel*, *solution*, *inplace=True*)

Takes a flux solution and transfers its reaction directionality as constraints for the cobra_model

Parameters

- **inplace** –
- **tmodel** –
- **solution** –

Returns

pytfa.core

Submodules

pytfa.core.model

Model class

Module Contents

Classes

LCSBModel	Helper class that provides a standard way to create an ABC using

Functions

timeit(method)	Adapted from Andreas Jung's `blog

pytfa.**timeit**(*method*)
Adapted from Andreas Jung's blog

Parameters **method** – The method to time

Returns

class pytfa.**LCSBModel**(*model*, *name*, *sloppy=False*)
Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

abstract copy(*self*)
Needs to be reimplemented, as our objects have complicated hierarchy :return:

print_info(*self*)
Print information and counts for the cobra_model :return:

add_variable(*self*, *kind*, *hook*, *queue=False*, ***kwargs*)
Add a new variable to a COBRApy cobra_model.

Parameters

- **kind** –
- **hook** (*string, cobra.Reaction*) – Either a string representing the name of the variable to add to the cobra_model, or a reaction object if the kind allows it

Returns The created variable

Return type optlang.interface.Variable

add_constraint(*self*, *kind*, *hook*, *expr*, *queue=False*, ***kwargs*)
Add a new constraint to a COBRApy cobra_model

Parameters

- **kind** –
- **hook** (*string, cobra.Reaction*) – Either a string representing the name of the variable to add to the cobra_model, or a reaction object if the kind allows it
- **expr** (*sympy.thermo.expr.Expr*) – The expression of the constraint

Returns The created constraint

Return type optlang.interface.Constraint

remove_reactions(*self*, *reactions*, *remove_orphans=False*)

remove_metabolites(*self*, *metabolite_list*, *destructive=False*)

_remove_associated_consvar(*self*, *all_cons_subclasses*, *all_var_subclasses*, *collection*)
Removes both the constraints and variables associated to an element, as long as it was used as a hook in the cons/var declaration. For example, upon removing a reaction, also removes its associated deltaG variables and coupling constraints

remove_variable(*self*, *var*)
Removes a variable

Parameters **var** –

Returns

remove_constraint(*self*, *cons*)
Removes a constraint

Parameters **cons** –

Returns

_push_queue(*self*)
updates the constraints and variables of the model with what's in the queue :return:

regenerate_variables(*self*)
Generates references to the cobra_model's constraints in self._var_dict as tab-searchable attributes of the thermo cobra_model :return:

regenerate_constraints(*self*)
Generates references to the cobra_model's constraints in self._cons_dict as tab-searchable attributes of the thermo cobra_model :return:

repair(*self*)
Updates references to variables and constraints :return:

get_primal(*self*, *vartype*, *index_by_reactions=False*)
Returns the primal value of the cobra_model for variables of a given type

Parameters

- **index_by_reactions** –
- **vartype** – Class of variable. Ex: pytfa.optim.variables.ThermoDisplacement

Returns

get_solution(*self*)
Overrides the cobra.thermo.solution method, to also get the supplementary variables we added to the cobra_model

- `solution.fluxes` in *cobrapy* is a transformed version of the solver output, as it actually calculates the _net_ flux of each reaction by substracting the reverse variable value to the forward variable value. This should be used anytime one needs the actual flux value
- `solution.raw` is a clear copy of the solver output. From there one can access the value at solution for all the variables of the problem. However, looking for a reaction ID in there will only give the _forward_ flux. This should be used for any other variable than fluxes.
- `solution.values` yields variables multiplied by their scaling factor (1 by default). Useful if you operated scaling on your equations for numerical reasons. This does _not_ include fluxes

Returns

optimize(*self*, *objective_sense=None*, ***kwargs*)
Call the Model.optimize function (which really is but an interface to the solver's. Catches SolverError in the case of no solutions. Passes down supplementary keyword arguments (see cobra.thermo.Model.optimize) :type objective_sense: 'min' or 'max'

slim_optimize(*self*, **args*, ***kwargs*)

get_constraints_of_type(*self*, *constraint_type*)
Convenience function that takes as input a constraint class and returns all its instances within the cobra_model

Parameters **constraint_type** –

Returns

get_variables_of_type(*self*, *variable_type*)
Convenience function that takes as input a variable class and returns all its instances within the cobra_model

Parameters **variable_type** –

Returns

Package Contents

Classes

LCSBModel	Helper class that provides a standard way to create an ABC using

class pytfa.core.**LCSBModel**(*model*, *name*, *sloppy=False*)
Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

abstract copy(*self*)
Needs to be reimplemented, as our objects have complicated hierarchy :return:

print_info(*self*)
Print information and counts for the cobra_model :return:

add_variable(*self*, *kind*, *hook*, *queue=False*, ***kwargs*)
Add a new variable to a COBRApy cobra_model.

Parameters

- **kind** –
- **hook** (*string,cobra.Reaction*) – Either a string representing the name of the variable to add to the cobra_model, or a reaction object if the kind allows it

Returns The created variable

Return type optlang.interface.Variable

add_constraint(*self*, *kind*, *hook*, *expr*, *queue=False*, ***kwargs*)
Add a new constraint to a COBRApy cobra_model

Parameters

- **kind** –

- **hook** (*string,cobra.Reaction*) – Either a string representing the name of the variable to add to the cobra_model, or a reaction object if the kind allows it
- **expr** (*sympy.thermo.expr.Expr*) – The expression of the constraint

Returns The created constraint

Return type optlang.interface.Constraint

remove_reactions(*self*, *reactions*, *remove_orphans=False*)

remove_metabolites(*self*, *metabolite_list*, *destructive=False*)

_remove_associated_consvar(*self*, *all_cons_subclasses*, *all_var_subclasses*, *collection*)
Removes both the constraints and variables associated to an element, as long as it was used as a hook in the cons/var declaration. For example, upon removing a reaction, also removes its associated deltaG variables and coupling constraints

remove_variable(*self*, *var*)
Removes a variable

Parameters **var** –

Returns

remove_constraint(*self*, *cons*)
Removes a constraint

Parameters **cons** –

Returns

_push_queue(*self*)
updates the constraints and variables of the model with what's in the queue :return:

regenerate_variables(*self*)
Generates references to the cobra_model's constraints in self._var_dict as tab-searchable attributes of the thermo cobra_model :return:

regenerate_constraints(*self*)
Generates references to the cobra_model's constraints in self._cons_dict as tab-searchable attributes of the thermo cobra_model :return:

repair(*self*)
Updates references to variables and constraints :return:

get_primal(*self*, *vartype*, *index_by_reactions=False*)
Returns the primal value of the cobra_model for variables of a given type

Parameters

- **index_by_reactions** –
- **vartype** – Class of variable. Ex: pytfa.optim.variables.ThermoDisplacement

Returns

get_solution(*self*)
Overrides the cobra.thermo.solution method, to also get the supplementary variables we added to the cobra_model

- `solution.fluxes` in *cobrapy* is a transformed version of the solver output, as it actually calculates the _net_ flux of each reaction by substracting the reverse variable value to the forward variable value. This should be used anytime one needs the actual flux value

- `solution.raw` is a clear copy of the solver output. From there one can access the value at solution for all the variables of the problem. However, looking for a reaction ID in there will only give the _forward_ flux. This should be used for any other variable than fluxes.
- `solution.values` yields variables multiplied by their scaling factor (1 by default). Useful if you operated scaling on your equations for numerical reasons. This does _not_ include fluxes

Returns

optimize(*self*, *objective_sense=None*, ***kwargs*)
Call the Model.optimize function (which really is but an interface to the solver's. Catches SolverError in the case of no solutions. Passes down supplementary keyword arguments (see cobra.thermo.Model.optimize) :type objective_sense: 'min' or 'max'

slim_optimize(*self*, **args*, ***kwargs*)

get_constraints_of_type(*self*, *constraint_type*)
Convenience function that takes as input a constraint class and returns all its instances within the cobra_model

Parameters **constraint_type** –

Returns

get_variables_of_type(*self*, *variable_type*)
Convenience function that takes as input a variable class and returns all its instances within the cobra_model

Parameters **variable_type** –

Returns

pytfa.io

Submodules

pytfa.io.base

Input/Output tools to import or export pytfa models

Module Contents

Functions

import_matlab_model(path, variable_name=None)	Convert at matlab cobra_model to a pyTFA cobra_model, with Thermodynamic values
recover_compartments(model, compartments_list)	
write_matlab_model(tmodel, path, varname='tmodel')	Writes the Thermo Model to a Matlab-compatible structure
create_thermo_dict(tmodel)	Dumps the thermodynamic information in a mat-compatible dictionary
varnames2matlab(name, tmodel)	Transforms reaction variable pairs from *('ACALD','ACALD_reverse_xxxxx')* to

continues on next page

Table 15 – continued from previous page

create_problem_dict(tmodel)	Dumps the the MILP formulation for TFA in a mat-compatible dictionary
create_generalized_matrix(tmodel, array_type='dense')	Returns the generalized stoichiomatric matrix used for TFA
load_thermoDB(path)	Load a thermodynamic database
printLP(model)	Print the LP file corresponding to the cobra_model
writeLP(model, path=None)	Write the LP file of the specified cobra_model to the file indicated by path.

pytfa.**import_matlab_model**(*path*, *variable_name=None*)

Convert at matlab cobra_model to a pyTFA cobra_model, with Thermodynamic values

Parameters

- **variable_name** –
- **path** (*string*) – The path of the file to import

Returns The converted cobra_model

Return type cobra.thermo.model.Model

pytfa.**recover_compartments**(*model*, *compartments_list*)

pytfa.**write_matlab_model**(*tmodel*, *path*, *varname='tmodel'*)

Writes the Thermo Model to a Matlab-compatible structure

Parameters

- **varname** –
- **tmodel** –
- **path** –

Returns None

pytfa.**create_thermo_dict**(*tmodel*)

Dumps the thermodynamic information in a mat-compatible dictionary (similar to the output of cobra.io.mat.create_mat_dict)

Parameters **tmodel** – pytfa.thermo.tmodel.ThermoModel

Returns dict object

pytfa.**varnames2matlab**(*name*, *tmodel*)

Transforms reaction variable pairs from *('ACALD','ACALD_reverse_xxxxx')* to *('F_ACALD','B_ACALD')* if it is a reaction, else leaves is as is

Returns

pytfa.**create_problem_dict**(*tmodel*)

Dumps the the MILP formulation for TFA in a mat-compatible dictionary (similar to the output of cobra.io.mat.create_mat_dict)

Parameters **tmodel** – pytfa.thermo.tmodel.ThermoModel

:ret

pytfa.**create_generalized_matrix**(*tmodel*, *array_type='dense'*)

Returns the generalized stoichiomatric matrix used for TFA

Parameters

- **array_type** –
- **tmodel** – pytfa.ThermoModel

Returns matrix.

pytfa.**load_thermoDB**(*path*)
Load a thermodynamic database

Parameters **path** (*string*) – The path of the file to load

Returns The thermodynamic database

Return type dict

pytfa.**printLP**(*model*)
Print the LP file corresponding to the cobra_model

Parameters **model** (*cobra.thermo.model.Model*) – The cobra_model to output the LP file for

Returns The content of the LP file

Return type str

Usually, you pass the result of this function to `file.write()` to write it on disk. If you prefer, you can use *pytfa.io.writeLP()* to write the result directly to a file.

pytfa.**writeLP**(*model*, *path=None*)
Write the LP file of the specified cobra_model to the file indicated by path.

Parameters

- **model** (*cobra.thermo.model.Model*) – The COBRApy cobra_model to write the LP file for
- **path** (*string*) – *Optional* The path of the file to be written. If not specified, the name of the COBRApy cobra_model will be used.

pytfa.io.enrichment

Tools to import or export enrichment to and from pytfa models

Module Contents

Functions

write_lexicon(tmodel, filepath)	Writes a csv file in the format :
annotate_from_lexicon(model, lexicon)	Converts a lexicon into annotation for the metabolites
read_lexicon(filepath)	
write_compartment_data(tmodel, filepath)	**param filepath**
read_compartment_data(filepath)	
apply_compartment_data(tmodel, compartment_data)	

pytfa.**write_lexicon**(*tmodel*, *filepath*)

Writes a csv file in the format : seed_id

13BDglcn_c cpd11791 13dpg_c cpd00203 2pg_c cpd00482 3pg_c cpd00169 4abut_c cpd00281

Useful for exporting an annotation

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **filepath** –

Returns

pytfa.**annotate_from_lexicon**(*model*, *lexicon*)

Converts a lexicon into annotation for the metabolites

Parameters

- **model** (*cobra.Model*) –
- **lexicon** –

Returns

pytfa.**read_lexicon**(*filepath*)

pytfa.**write_compartment_data**(*tmodel*, *filepath*)

Parameters

- **filepath** –
- **tmodel** (*pytfa.core.ThermoModel*) –

Returns

pytfa.**read_compartment_data**(*filepath*)

pytfa.**apply_compartment_data**(*tmodel*, *compartment_data*)

pytfa.io.json

JSON serialization

Module Contents

Classes

MyEncoder	We define an encoder that takes care of the serialization of numpy types,

Functions

check_json_extension(filepath)	
save_json_model(model, filepath)	
load_json_model(filepath)	
json_dumps_model(model)	Returns a JSON dump as a string
json_loads_model(s)	Loads a model from a string JSON dump

class pytfa.**MyEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: json.JSONEncoder

We define an encoder that takes care of the serialization of numpy types, which are not handled by json by default

default(*self*, *obj*)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

pytfa.**check_json_extension**(*filepath*)

pytfa.**save_json_model**(*model*, *filepath*)

pytfa.**load_json_model**(*filepath*)

pytfa.**json_dumps_model**(*model*)

Returns a JSON dump as a string

Parameters **model** –

Returns

pytfa.**json_loads_model**(*s*)

Loads a model from a string JSON dump

Parameters **s** – JSON string

Returns

pytfa.io.plotting

Plotting results

Module Contents

Functions

plot_fva_tva_comparison(fva, tva)	

plot_thermo_displacement_histogram(tmodel, solution=None)	Plot a histogram of the thermodynamic displacement. if no solution is
plot_histogram(values, **kwargs)	Convenience function. Plots a histogram of flat 1D data.

pytfa.**plot_fva_tva_comparison**(*fva*, *tva*)

pytfa.**plot_thermo_displacement_histogram**(*tmodel*, *solution=None*)

Plot a histogram of the thermodynamic displacement. if no solution is provided, will look at the cobra_model's own solution

Parameters

- **tmodel** –
- **solution** –

Returns

pytfa.**plot_histogram**(*values*, ***kwargs*)

Convenience function. Plots a histogram of flat 1D data.

Parameters **values** –

Returns

pytfa.io.viz

Input/Output tools to vizualize results

Module Contents

Functions

export_variable_for_escher(tmodel, variable_type, data, filename)	Exports all the variables of a given type into a csv file, indexed by
get_reaction_data(tmodel, data)	Exports values indexed by reaction ids. Reconciles Forward and Backwards
export_reactions_for_escher(tmodel, data, filename)	Exports values indexed by reaction ids. Reconciles Forward and Backwards

pytfa.**export_variable_for_escher**(*tmodel*, *variable_type*, *data*, *filename*)

Exports all the variables of a given type into a csv file, indexed by variable.id. This format is read-

able by escher if the variable_type is a subclass of **:pytfa:`pytfa.optim.variables.ReactionVariable`** or **:pytfa:`pytfa.optim.variables.MetaboliteVariable`**

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **variable_type** (ReactionVariable / MetaboliteVariable) –
- **data** (*pandas.Series*) – indexed by variable name
- **filename** (*string*) –

Returns

pytfa.**get_reaction_data**(*tmodel*, *data*)

Exports values indexed by reaction ids. Reconciles Forward and Backwards variables.

pytfa.**export_reactions_for_escher**(*tmodel*, *data*, *filename*)

Exports values indexed by reaction ids. Reconciles Forward and Backwards variables. Writes it in a csv file. This format is readable by escher

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **variable_type** (ReactionVariable / MetaboliteVariable) –
- **data** (*pandas.Series*) – indexed by variable name
- **filename** (*string*) –

Returns

Package Contents

Classes

MyEncoder	We define an encoder that takes care of the serialization of numpy types,

Functions

import_matlab_model(path, variable_name=None)	Convert at matlab cobra_model to a pyTFA cobra_model, with Thermodynamic values
recover_compartments(model, compartments_list)	
write_matlab_model(tmodel, path, varname='tmodel')	Writes the Thermo Model to a Matlab-compatible structure
create_thermo_dict(tmodel)	Dumps the thermodynamic information in a mat-compatible dictionary
varnames2matlab(name, tmodel)	Transforms reaction variable pairs from *('ACALD','ACALD_reverse_xxxxx')* to
create_problem_dict(tmodel)	Dumps the the MILP formulation for TFA in a mat-compatible dictionary

continues on next page

Table 22 – continued from previous page

create_generalized_matrix(tmodel, array_type='dense')	Returns the generalized stoichiomatric matrix used for TFA
load_thermoDB(path)	Load a thermodynamic database
printLP(model)	Print the LP file corresponding to the cobra_model
writeLP(model, path=None)	Write the LP file of the specified cobra_model to the file indicated by path.
write_lexicon(tmodel, filepath)	Writes a csv file in the format :
annotate_from_lexicon(model, lexicon)	Converts a lexicon into annotation for the metabolites
read_lexicon(filepath)	
write_compartment_data(tmodel, filepath)	**param filepath**
read_compartment_data(filepath)	
apply_compartment_data(tmodel, compartment_data)	

Attributes

BIGM_DG

pytfa.io.**BIGM_DG = 1000.0**

pytfa.io.**import_matlab_model**(*path*, *variable_name=None*)

Convert at matlab cobra_model to a pyTFA cobra_model, with Thermodynamic values

Parameters

- **variable_name** –
- **path** (*string*) – The path of the file to import

Returns The converted cobra_model

Return type cobra.thermo.model.Model

pytfa.io.**recover_compartments**(*model*, *compartments_list*)

pytfa.io.**write_matlab_model**(*tmodel*, *path*, *varname='tmodel'*)

Writes the Thermo Model to a Matlab-compatible structure

Parameters

- **varname** –
- **tmodel** –
- **path** –

Returns None

pytfa.io.**create_thermo_dict**(*tmodel*)

Dumps the thermodynamic information in a mat-compatible dictionary (similar to the output of cobra.io.mat.create_mat_dict)

Parameters **tmodel** – pytfa.thermo.tmodel.ThermoModel

Returns dict object

pytfa.io.**varnames2matlab**(*name*, *tmodel*)
Transforms reaction variable pairs from *('ACALD','ACALD_reverse_xxxxx')* to *('F_ACALD','B_ACALD')* if it is a reaction, else leaves is as is

Returns

pytfa.io.**create_problem_dict**(*tmodel*)
Dumps the the MILP formulation for TFA in a mat-compatible dictionary (similar to the output of cobra.io.mat.create_mat_dict)

Parameters **tmodel** – pytfa.thermo.tmodel.ThermoModel

:ret

pytfa.io.**create_generalized_matrix**(*tmodel*, *array_type='dense'*)
Returns the generalized stoichiomatric matrix used for TFA

Parameters

- **array_type** –
- **tmodel** – pytfa.ThermoModel

Returns matrix.

pytfa.io.**load_thermoDB**(*path*)
Load a thermodynamic database

Parameters **path** (*string*) – The path of the file to load

Returns The thermodynamic database

Return type dict

pytfa.io.**printLP**(*model*)
Print the LP file corresponding to the cobra_model

Parameters **model** (*cobra.thermo.model.Model*) – The cobra_model to output the LP file for

Returns The content of the LP file

Return type str

Usually, you pass the result of this function to `file.write()` to write it on disk. If you prefer, you can use *pytfa.io.writeLP()* to write the result directly to a file.

pytfa.io.**writeLP**(*model*, *path=None*)
Write the LP file of the specified cobra_model to the file indicated by path.

Parameters

- **model** (*cobra.thermo.model.Model*) – The COBRApy cobra_model to write the LP file for
- **path** (*string*) – *Optional* The path of the file to be written. If not specified, the name of the COBRApy cobra_model will be used.

class pytfa.io.**MyEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)
Bases: json.JSONEncoder

We define an encoder that takes care of the serialization of numpy types, which are not handled by json by default

default(*self*, *obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

pytfa.io.**write_lexicon**(*tmodel*, *filepath*)

Writes a csv file in the format : seed_id

13BDglcn_c cpd11791 13dpg_c cpd00203 2pg_c cpd00482 3pg_c cpd00169 4abut_c cpd00281

Useful for exporting an annotation

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **filepath** –

Returns

pytfa.io.**annotate_from_lexicon**(*model*, *lexicon*)

Converts a lexicon into annotation for the metabolites

Parameters

- **model** (*cobra.Model*) –
- **lexicon** –

Returns

pytfa.io.**read_lexicon**(*filepath*)

pytfa.io.**write_compartment_data**(*tmodel*, *filepath*)

Parameters

- **filepath** –
- **tmodel** (*pytfa.core.ThermoModel*) –

Returns

pytfa.io.**read_compartment_data**(*filepath*)

pytfa.io.**apply_compartment_data**(*tmodel*, *compartment_data*)

pytfa.optim

Submodules

pytfa.optim.config

Pre-tuned configurations for faster solving

Module Contents

Functions

dg_relax_config(model)	**param model**

pytfa.**dg_relax_config**(*model*)

Parameters **model** –

Returns

pytfa.optim.constraints

Constraints declarations

Module Contents

Classes

GenericConstraint	Class to represent a generic constraint. The purpose is that the interface
ModelConstraint	Class to represent a variable attached to the model
GeneConstraint	Class to represent a variable attached to a enzyme
ReactionConstraint	Class to represent a variable attached to a reaction
MetaboliteConstraint	Class to represent a variable attached to a enzyme
NegativeDeltaG	Class to represent thermodynamics constraints.
NegativeDeltaGIneq	Class to represent thermodynamics constraints.
ForwardDeltaGCoupling	Class to represent thermodynamics coupling: DeltaG of reactions has to be
BackwardDeltaGCoupling	Class to represent thermodynamics coupling: DeltaG of reactions has to be
ForwardDirectionCoupling	Class to represent a forward directionality coupling with thermodynamics on
BackwardDirectionCoupling	Class to represent a backward directionality coupling with thermodynamics on

continues on next page

Table 25 – continued from previous page

SimultaneousUse	Class to represent a simultaneous use constraint on reaction variables
DisplacementCoupling	Class to represent the coupling to the thermodynamic displacement
ForbiddenProfile	Class to represent a forbidden net flux directionality profile
LinearizationConstraint	Class to represent a variable attached to a reaction
LowerBoundLogConcentration	Class to represent a forbidden net flux directionality profile
UpperBoundLogConcentration	Class to represent a forbidden net flux directionality profile
NullspaceConstraint	Class to represent a forbidden net flux directionality profile
PotentialConstraint	Class to represent a forbidden net flux directionality profile
PotentialCoupling	Class to rep couple DG to potential variable

class pytfa.**GenericConstraint**(*expr*, *id_=''*, *model=None*, *hook=None*, *queue=False*, ***kwargs*)

Class to represent a generic constraint. The purpose is that the interface is instantiated on initialization, to follow the type of interface used by the problem, and avoid incompatibilities in optlang

Attributes:

id Used for DictList comprehension. Usually points back at a

enzyme or reaction id for ease of linking. Should be unique given a constraint type. :name: Should be a concatenation of the id and a prefix that is specific to the variable type. will be used to address the constraint at the solver level, and hence should be unique in the whole cobra_model :expr: the expression of the constraint (sympy.Expression subtype) :cobra_model: the cobra_model hook. :constraint: links directly to the cobra_model representation of tbe constraint

prefix

property __attrname__(*self*)
Name the attribute the instances will have Example: GenericConstraint -> generic_constraint :return:

get_interface(*self*, *expr*, *queue*)
Called upon completion of __init__, initializes the value of self.var, which is returned upon call, and stores the actual interfaced variable.

Returns instance of Variable from the problem

make_name(*self*)

Needs to be overridden by the subclass, concats the id with a prefix

Returns None

change_expr(*self*, *new_expr*, *sloppy=False*)

property expr(*self*)

property name(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property constraint(*self*)

property model(*self*)

__repr__(*self*)
Return repr(self).

class pytfa.ModelConstraint(*model*, *expr*, *id_*, ***kwargs*)
Bases: GenericConstraint

Class to represent a variable attached to the model

prefix = MODC_

class pytfa.GeneConstraint(*gene*, *expr*, ***kwargs*)
Bases: GenericConstraint

Class to represent a variable attached to a enzyme

prefix = GC_

property gene(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.ReactionConstraint(*reaction*, *expr*, ***kwargs*)
Bases: GenericConstraint

Class to represent a variable attached to a reaction

prefix = RC_

property reaction(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.MetaboliteConstraint(*metabolite*, *expr*, ***kwargs*)
Bases: GenericConstraint

Class to represent a variable attached to a enzyme

prefix = MC_

property metabolite(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.NegativeDeltaG(*reaction*, *expr*, ***kwargs*)
Bases: ReactionConstraint

Class to represent thermodynamics constraints.

G: - DGR_rxn + DGoRerr_Rxn + RT * StoichCoefProd1 * LC_prod1

- RT * StoichCoefProd2 * LC_prod2
- RT * StoichCoefSub1 * LC_subs1
- RT * StoichCoefSub2 * LC_subs2

- …

= 0

prefix = G_

class pytfa.**NegativeDeltaGIneq**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics constraints.

G: DGoRerr_Rxn + RT * StoichCoefProd1 * LC_prod1

- RT * StoichCoefProd2 * LC_prod2
- RT * StoichCoefSub1 * LC_subs1
- RT * StoichCoefSub2 * LC_subs2
- …

< or > 0

prefix = GI_

class pytfa.**ForwardDeltaGCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics coupling: DeltaG of reactions has to be DGR < 0 for the reaction to proceed forwards Looks like: FU_rxn: 1000 FU_rxn + DGR_rxn < 1000

prefix = FU_

class pytfa.**BackwardDeltaGCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics coupling: DeltaG of reactions has to be DGR > 0 for the reaction to proceed backwards Looks like: BU_rxn: 1000 BU_rxn - DGR_rxn < 1000

prefix = BU_

class pytfa.**ForwardDirectionCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a forward directionality coupling with thermodynamics on reaction variables Looks like : UF_rxn: F_rxn - M FU_rxn < 0

prefix = UF_

class pytfa.**BackwardDirectionCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a backward directionality coupling with thermodynamics on reaction variables Looks like : UR_rxn: R_rxn - M RU_rxn < 0

prefix = UR_

class pytfa.**SimultaneousUse**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a simultaneous use constraint on reaction variables Looks like: SU_rxn: FU_rxn + BU_rxn <= 1

prefix = SU_

class pytfa.**DisplacementCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent the coupling to the thermodynamic displacement Looks like: Ln(Gamma) - (1/RT)*DGR_rxn = 0

prefix = DC_

class pytfa.**ForbiddenProfile**(*model*, *expr*, *id_*, ***kwargs*)
Bases: *GenericConstraint*

Class to represent a forbidden net flux directionality profile Looks like: FU_rxn_1 + BU_rxn_2 + … + FU_rxn_n <= n-1

prefix = FP_

class pytfa.**LinearizationConstraint**(*model*, *expr*, *id_*, ***kwargs*)
Bases: *ModelConstraint*

Class to represent a variable attached to a reaction

prefix = LC_

static from_constraints(*cons*, *model*)

class pytfa.**LowerBoundLogConcentration**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = LB_

class pytfa.**UpperBoundLogConcentration**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = UB_

class pytfa.**NullspaceConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = NS_

class pytfa.**PotentialConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = PC_

class pytfa.**PotentialCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to rep couple DG to potential variable

prefix = PPC_

pytfa.optim.debugging

Debugging of models

Module Contents

Functions

debug_iis(model)	Performs reduction to an Irreducible Inconsistent Subsystem (IIS)
find_extreme_coeffs(model, n=5)	
find_maxed_vars(model, ub=1000, epsilon=0.01)	

pytfa.**debug_iis**(*model*)
Performs reduction to an Irreducible Inconsistent Subsystem (IIS)

Parameters **model** –

Returns

pytfa.**find_extreme_coeffs**(*model*, *n=5*)

pytfa.**find_maxed_vars**(*model*, *ub=1000*, *epsilon=0.01*)

pytfa.optim.meta

Metaclass declarations to force the definition of prefixes in GenericVariable and GeneriConstraint subclasses

Based on SethMMorton's answer on StackOverflow https://stackoverflow.com/questions/45248243/most-pythonic-way-to-declare-an-abstract-class-property https://stackoverflow.com/a/45250114

Module Contents

Classes

RequirePrefixMeta	Metaclass that enforces child classes define prefix.

Attributes

ABCRequirePrefixMeta

class pytfa.**RequirePrefixMeta**(*cls*, *name*, *bases*, *attrs*)
Bases: type

Metaclass that enforces child classes define prefix.

pytfa.**ABCRequirePrefixMeta**

pytfa.optim.reformulation

MILP-fu to reformulate problems

Module Contents

Functions

subs_bilinear(expr)	Substitutes bilinear forms from an expression with dedicated variables
glovers_linearization(b, fy, z=None, L=0, U=1000)	Glover, Fred.
petersen_linearization(b, x, z=None, M=1000)	PETERSEN, C,,
linearize_product(model, b, x, queue=False)	**param model**

Attributes

ConstraintTuple
OPTLANG_BINARY

pytfa.**ConstraintTuple**

pytfa.**OPTLANG_BINARY = binary**

pytfa.**subs_bilinear**(*expr*)
Substitutes bilinear forms from an expression with dedicated variables :param expr: :return:

pytfa.**glovers_linearization**(*b*, *fy*, *z=None*, *L=0*, *U=1000*)
Glover, Fred. "Improved linear integer programming formulations of nonlinear integer problems." Management Science 22.4 (1975): 455-460.

Performs Glovers Linearization of a product z = b*f(y) <=> z - b*f(y) = 0 <=> { L*b <= z <= U*b { f(y) - U*(1-b) <= z <= f(y) - L*(1-b)

where : * b is a binary variable * f a linear combination of continuous or integer variables y

Parameters

- **b** – Must be a binary optlang variable
- **z** – Must be an optlang variable. Will be mapped to the product so that z = b*f(y)
- **fy** – Must be an expression or variable
- **L** – minimal value for fy
- **U** – maximal value for fy

Returns

pytfa.**petersen_linearization**(*b*, *x*, *z=None*, *M=1000*)

PETERSEN, C,, "A Note on Transforming the Product of Variables to Linear Form in Linear CLIFFORD Programs," Working Paper, Purdue University, 1971.

Performs Petersen Linearization of a product z = b*x <=> z - b*x = 0 <=> { x + M*b - M <= z <= M*b { z <= x

where : * b is a binary variable * f a linear combination of continuous or integer variables y

Parameters

- **x** – Must be an expression or variable
- **b** – Must be a binary optlang variable
- **z** – Must be an optlang variable. Will be mapped to the product so that z = b*f(y)
- **M** – big-M constraint

Returns

pytfa.**linearize_product**(*model*, *b*, *x*, *queue=False*)

Parameters

- **model** –
- **b** – the binary variable
- **x** – the continuous variable
- **queue** – whether to queue the variables and constraints made

Returns

pytfa.optim.relaxation

Relaxation of models with constraint too tight

Module Contents

Functions

relax_dgo_gurobi(model, relax_obj_type=0)	
relax_dgo(tmodel, reactions_to_ignore=(), solver=None, in_place=False)	**param t_tmodel**
relax_dgo_err(tmodel, reactions_to_ignore=(), max_sigma=3, solver=None, in_place=False)	**param t_tmodel**
relax_lc(tmodel, metabolites_to_ignore=(), solver=None)	**param metabolites_to_ignore**

Attributes

BIGM
BIGM_THERMO
BIGM_DG
BIGM_P
EPSILON

pytfa.**BIGM**

pytfa.**BIGM_THERMO**

pytfa.**BIGM_DG**

pytfa.**BIGM_P**

pytfa.**EPSILON**

pytfa.**relax_dgo_gurobi**(*model*, *relax_obj_type=0*)

pytfa.**relax_dgo**(*tmodel*, *reactions_to_ignore=()*, *solver=None*, *in_place=False*)

Parameters

- **t_tmodel** (*pytfa.thermo.ThermoModel:*) –
- **reactions_to_ignore** – Iterable of reactions that should not be relaxed
- **solver** – solver to use (e.g. 'optlang-glpk', 'optlang-cplex', 'optlang-gurobi'

Returns a cobra_model with relaxed bounds on standard Gibbs free energy

pytfa.**relax_dgo_err**(*tmodel*, *reactions_to_ignore=()*, *max_sigma=3*, *solver=None*, *in_place=False*)

Parameters

- **t_tmodel** (*pytfa.thermo.ThermoModel:*) –
- **reactions_to_ignore** – Iterable of reactions that should not be relaxed
- **solver** – solver to use (e.g. 'optlang-glpk', 'optlang-cplex', 'optlang-gurobi'

Returns a cobra_model with relaxed bounds on standard Gibbs free energy

pytfa.**relax_lc**(*tmodel*, *metabolites_to_ignore=()*, *solver=None*)

Parameters

- **metabolites_to_ignore** –
- **in_tmodel** (*pytfa.thermo.ThermoModel:*) –
- **min_objective_value** –

Returns

pytfa.optim.utils

Relaxation of models with constraint too tight

Module Contents

Functions

get_all_subclasses(cls)	Given a variable or constraint class, get all the subclassses
chunk_sum(variables)	This functions handles the sum of many sympy variables by chunks, which
symbol_sum(variables)	``` python
get_solution_value_for_variables(solution, these_vars, index_by_reaction=False)	
compare_solutions(models)	returns the solution dictionnary for each cobra_model
evaluate_constraint_at_solution(constraint, solution)	**param expression**
get_active_use_variables(tmodel, solution)	Returns the active use variables in a solution. Use Variables are binary
get_direction_use_variables(tmodel, solution)	Returns the active use variables in a solution. Use Variables are binary
get_primal(tmodel, vartype, index_by_reactions=False)	Returns the primal value of the cobra_model for variables of a given type
strip_from_integer_variables(tmodel)	Removes all integer and binary variables of a cobra_model, to make it sample-able
copy_solver_configuration(source, target)	Copies the solver configuration from a source model to a target model

Attributes

SYMPY_ADD_CHUNKSIZE
INTEGER_VARIABLE_TYPES

pytfa.**SYMPY_ADD_CHUNKSIZE = 100**

pytfa.**INTEGER_VARIABLE_TYPES = ['binary', 'integer']**

pytfa.**get_all_subclasses**(*cls*)

Given a variable or constraint class, get all the subclassses that inherit from it

Parameters **cls** –

Returns

pytfa.**chunk_sum**(*variables*)

This functions handles the sum of many sympy variables by chunks, which somehow increases the speed of the computation

You can test it in IPython: ```python a = sympy.symbols('a0:100') %timeit (sum(a)) # >>> 198 µs ± 11.4 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

b = sympy.symbols('b0:1000') %timeit (sum(b)) # >>> 1.85 ms ± 356 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

c = sympy.symbols('c0:3000') %timeit (sum(c)) # >>> 5min 7s ± 2.57 s per loop (mean ± std. dev. of 7 runs, 1 loop each) ```

See the github thread

Parameters **variables** –

Returns

pytfa.**symbol_sum**(*variables*)

``` python a = symbols('a0:100')

%timeit Add(*a) # >>> 10000 loops, best of 3: 34.1 µs per loop

b = symbols('b0:1000')

%timeit Add(*b) # >>> 1000 loops, best of 3: 343 µs per loop

c = symbols('c0:3000')

%timeit Add(*c) # >>> 1 loops, best of 3: 1.03 ms per loop ```

See the github thread :param variables: :return:

pytfa.**get_solution_value_for_variables**(*solution*, *these_vars*, *index_by_reaction=False*)

pytfa.**compare_solutions**(*models*)

returns the solution dictionnary for each cobra_model :param (iterable (pytfa.thermo.ThermoModel)) models: :return:

pytfa.**evaluate_constraint_at_solution**(*constraint*, *solution*)

**Parameters**

- **expression** –
- **solution** – pandas.DataFrame, with index as variable names

**Returns**

pytfa.**get_active_use_variables**(*tmodel*, *solution*)

Returns the active use variables in a solution. Use Variables are binary variables that control the directionality of the reaction

ex: FU_ACALDt BU_PFK

**Parameters**

- **tmodel** (*pytfa.core.ThermoModel*) –
- **solution** –

**Returns**

pytfa.**get_direction_use_variables**(*tmodel*, *solution*)

Returns the active use variables in a solution. Use Variables are binary variables that control the directionality of the reaction The difference with get_active_use_variables is that variables with both UseVariables at 0 will return as going forwards. This is to ensure that the output size of the function is equal to the number of FDPs
```

ex: FU_ACALDt BU_PFK

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **solution** –

Returns

pytfa.**get_primal**(*tmodel*, *vartype*, *index_by_reactions=False*)
Returns the primal value of the cobra_model for variables of a given type :param tmodel: :param vartype: Class of variable. Ex: pytfa.optim.variables.ThermoDisplacement :param index_by_reactions: Set to true to get reaction names as index instead of

variables. Useful for Escher export

Returns

pytfa.**strip_from_integer_variables**(*tmodel*)
Removes all integer and binary variables of a cobra_model, to make it sample-able :param tmodel: :return:

pytfa.**copy_solver_configuration**(*source*, *target*)
Copies the solver configuration from a source model to a target model :param source: :param target: :return:

pytfa.optim.variables

Variable declarations

Module Contents

Classes

GenericVariable	Class to represent a generic variable. The purpose is that the interface
ModelVariable	Class to represent a variable attached to the model
GeneVariable	Class to represent a gene variable
BinaryVariable	Class to represent a generic binary variable
IntegerVariable	Class to represent a generic binary variable
ReactionVariable	Class to represent a variable attached to a reaction
MetaboliteVariable	Class to represent a variable attached to a enzyme
ForwardUseVariable	Class to represent a forward use variable, a type of binary variable used to
BackwardUseVariable	Class to represent a backward use variable, a type of binary variable used
ForwardBackwardUseVariable	Class to represent a type of binary variable used to tell whether the
LogConcentration	Class to represent a log concentration of a enzyme
DeltaGErr	Class to represent a DeltaGErr
DeltaG	Class to represent a DeltaG
ThermoPotential	Class to represent a DeltaG
DeltaGFormstd	Class to represent a DeltaGstf of formation

continues on next page

Table 35 – continued from previous page

DeltaGstd	Class to represent a DeltaG^o (naught) - standard conditions
ThermoDisplacement	Class to represent the thermodynamic displacement of a reaction
PosSlackVariable	Class to represent a positive slack variable for relaxation problems
NegSlackVariable	Class to represent a negative slack variable for relaxation problems
PosSlackVariableInt	Class to represent a postive slack variable for relaxation problems using integer
NegSlackVariableInt	Class to represent a postive slack variable for relaxation problems using integer
PosSlackLC	Class to represent a variable attached to a enzyme
NegSlackLC	Class to represent a variable attached to a enzyme
LinearizationVariable	Class to represent a product A*B when performing linearization of the

Functions

get_binary_type()	FIX : We enforce type to be integer instead of binary, else optlang does

Attributes

op_replace_dict

pytfa.**op_replace_dict**

class pytfa.**GenericVariable**(*id_=''*, *model=None*, *hook=None*, *queue=False*, *scaling_factor=1*, ***kwargs*)

Class to represent a generic variable. The purpose is that the interface is instantiated on initialization, to follow the type of interface used by the problem, and avoid incompatibilities in optlang

Attributes:

id Used for DictList comprehension. Usually points back at a

enzyme or reaction id for ease of linking. Should be unique given a variable type. :name: Should be a concatenation of the id and a prefix that is specific to the variable type. will be used to address the variable at the solver level, and hence should be unique in the whole cobra_model :cobra_model: the cobra_model hook. :variable: links directly to the cobra_model representation of tbe variable

prefix

property __attrname__(*self*)

Name the attribute the instances will have Example: GenericVariable -> generic_variable :return:

get_interface(*self*, *queue*)

Called upon completion of __init__, initializes the value of self.var, which is returned upon call, and stores the actual interfaced variable.

Returns instance of Variable from the problem

make_name(*self*)

Needs to be overridden by the subclass, concats the id with a prefix

Returns None

property name(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property variable(*self*)

property scaling_factor(*self*)

property unscaled(*self*)
If the scaling factor of quantity X is a, it is represented by the variable X_hat = X/a. This returns X = a.X_hat Useful for nondimensionalisation of variables and constraints

Returns The variable divided by its scaling factor

property value(*self*)

property unscaled_value(*self*)

property model(*self*)

property type(*self*)

test_consistency(*self*, *other*)
Tests whether a candidate to an operation is of the right type and is from the same problem

Parameters **other** – an object

Returns None

get_operand(*self*, *other*)
For operations, choose if the operand is a GenericVariable, in which we return its optlang variable, or something else (presumably a numeric) and we let optlang decide what to do

Parameters **other** –

Returns

__add__(*self*, *other*)
Adding either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__radd__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__sub__(*self*, *other*)
Substracting either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rsub__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__mul__(*self*, *other*)
Multiplying either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rmul__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__truediv__(*self*, *other*)
Dividing either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rtruediv__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

make_result(*self*, *new_variable*)
Returns a Sympy expression :param new_variable: :return:

__repr__(*self*)
Return repr(self).

pytfa.**get_binary_type**()
FIX : We enforce type to be integer instead of binary, else optlang does not allow to set the binary variable bounds to anything other than (0,1) You might want to set it at (0,0) to enforce directionality for example

class pytfa.**ModelVariable**(*model*, *id_*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a variable attached to the model

prefix = MODV_

class pytfa.**GeneVariable**(*gene*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a gene variable

prefix = GV_

property gene(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.**BinaryVariable**(*id_*, *model*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a generic binary variable

prefix = B_

class pytfa.**IntegerVariable**(*id_*, *model*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a generic binary variable

prefix = I_

class pytfa.**ReactionVariable**(*reaction*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a variable attached to a reaction

prefix = RV_

property reaction(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.**MetaboliteVariable**(*metabolite*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a variable attached to a enzyme

prefix = MV_

property metabolite(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.**ForwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a forward use variable, a type of binary variable used to enforce forward directionality in reaction net fluxes

prefix = FU_

class pytfa.**BackwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a backward use variable, a type of binary variable used to enforce backward directionality in reaction net fluxes

prefix = BU_

class pytfa.**ForwardBackwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a type of binary variable used to tell whether the reaction is active or not such that:

FU + BU + BFUSE = 1

prefix = BFUSE_

class pytfa.**LogConcentration**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a log concentration of a enzyme

prefix = LC_

class pytfa.**DeltaGErr**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaGErr

prefix = DGE_

class pytfa.**DeltaG**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaG

prefix = DG_

class pytfa.**ThermoPotential**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a DeltaG

prefix = P_

class pytfa.**DeltaGFormstd**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a DeltaGstf of formation

prefix = DGoF_

class pytfa.**DeltaGstd**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaG^o (naught) - standard conditions

prefix = DGo_

class pytfa.**ThermoDisplacement**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent the thermodynamic displacement of a reaction Gamma = -DeltaG/RT

prefix = LnGamma_

class pytfa.**PosSlackVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a positive slack variable for relaxation problems

prefix = PosSlack_

class pytfa.**NegSlackVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a negative slack variable for relaxation problems

prefix = NegSlack_

class pytfa.**PosSlackVariableInt**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *IntegerVariable*

Class to represent a postive slack variable for relaxation problems using integer enforce forward directionality in reaction net fluxes

prefix = PosSlackInt_

class pytfa.**NegSlackVariableInt**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *IntegerVariable*

Class to represent a postive slack variable for relaxation problems using integer enforce forward directionality in reaction net fluxes

prefix = NegSlackInt_

class pytfa.**PosSlackLC**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a variable attached to a enzyme

prefix = PosSlackLC_

class pytfa.**NegSlackLC**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a variable attached to a enzyme

prefix = NegSlackLC_

class pytfa.**LinearizationVariable**(*model*, *id_*, ***kwargs*)

Bases: *ModelVariable*

Class to represent a product A*B when performing linearization of the model

prefix = LZ_

Package Contents

Classes

GenericConstraint	Class to represent a generic constraint. The purpose is that the interface
ModelConstraint	Class to represent a variable attached to the model
GeneConstraint	Class to represent a variable attached to a enzyme
ReactionConstraint	Class to represent a variable attached to a reaction
MetaboliteConstraint	Class to represent a variable attached to a enzyme
NegativeDeltaG	Class to represent thermodynamics constraints.
NegativeDeltaGIneq	Class to represent thermodynamics constraints.
ForwardDeltaGCoupling	Class to represent thermodynamics coupling: DeltaG of reactions has to be
BackwardDeltaGCoupling	Class to represent thermodynamics coupling: DeltaG of reactions has to be
ForwardDirectionCoupling	Class to represent a forward directionality coupling with thermodynamics on
BackwardDirectionCoupling	Class to represent a backward directionality coupling with thermodynamics on
SimultaneousUse	Class to represent a simultaneous use constraint on reaction variables
DisplacementCoupling	Class to represent the coupling to the thermodynamic displacement
ForbiddenProfile	Class to represent a forbidden net flux directionality profile
LinearizationConstraint	Class to represent a variable attached to a reaction
LowerBoundLogConcentration	Class to represent a forbidden net flux directionality profile
UpperBoundLogConcentration	Class to represent a forbidden net flux directionality profile
NullspaceConstraint	Class to represent a forbidden net flux directionality profile
PotentialConstraint	Class to represent a forbidden net flux directionality profile
PotentialCoupling	Class to rep couple DG to potential variable
GenericVariable	Class to represent a generic variable. The purpose is that the interface
ModelVariable	Class to represent a variable attached to the model
GeneVariable	Class to represent a gene variable
BinaryVariable	Class to represent a generic binary variable
IntegerVariable	Class to represent a generic binary variable
ReactionVariable	Class to represent a variable attached to a reaction
MetaboliteVariable	Class to represent a variable attached to a enzyme

continues on next page

Table 38 – continued from previous page

ForwardUseVariable	Class to represent a forward use variable, a type of binary variable used to
BackwardUseVariable	Class to represent a backward use variable, a type of binary variable used
ForwardBackwardUseVariable	Class to represent a type of binary variable used to tell whether the
LogConcentration	Class to represent a log concentration of a enzyme
DeltaGErr	Class to represent a DeltaGErr
DeltaG	Class to represent a DeltaG
ThermoPotential	Class to represent a DeltaG
DeltaGFormstd	Class to represent a DeltaGstf of formation
DeltaGstd	Class to represent a DeltaG^o (naught) - standard conditions
ThermoDisplacement	Class to represent the thermodynamic displacement of a reaction
PosSlackVariable	Class to represent a positive slack variable for relaxation problems
NegSlackVariable	Class to represent a negative slack variable for relaxation problems
PosSlackVariableInt	Class to represent a postive slack variable for relaxation problems using integer
NegSlackVariableInt	Class to represent a postive slack variable for relaxation problems using integer
PosSlackLC	Class to represent a variable attached to a enzyme
NegSlackLC	Class to represent a variable attached to a enzyme
LinearizationVariable	Class to represent a product A*B when performing linearization of the
NegativeDeltaG	Class to represent thermodynamics constraints.
PosSlackVariable	Class to represent a positive slack variable for relaxation problems
NegSlackVariable	Class to represent a negative slack variable for relaxation problems
DeltaGstd	Class to represent a DeltaG^o (naught) - standard conditions
LogConcentration	Class to represent a log concentration of a enzyme
NegSlackLC	Class to represent a variable attached to a enzyme
PosSlackLC	Class to represent a variable attached to a enzyme
PosSlackVariableInt	Class to represent a postive slack variable for relaxation problems using integer
NegSlackVariableInt	Class to represent a postive slack variable for relaxation problems using integer
GenericConstraint	Class to represent a generic constraint. The purpose is that the interface
ForwardUseVariable	Class to represent a forward use variable, a type of binary variable used to
BackwardUseVariable	Class to represent a backward use variable, a type of binary variable used
GenericVariable	Class to represent a generic variable. The purpose is that the interface

Functions

camel2underscores(name)	
camel2underscores(name)	
get_binary_type()	FIX : We enforce type to be integer instead of binary, else optlang does
dg_relax_config(model)	**param model**
get_solution_value_for_variables(solution, these_vars, index_by_reaction=False)	
chunk_sum(variables)	This functions handles the sum of many sympy variables by chunks, which
symbol_sum(variables)	``` python
relax_dgo_gurobi(model, relax_obj_type=0)	
relax_dgo(tmodel, reactions_to_ignore=(), solver=None, in_place=False)	**param t_tmodel**
relax_dgo_err(tmodel, reactions_to_ignore=(), max_sigma=3, solver=None, in_place=False)	**param t_tmodel**
relax_lc(tmodel, metabolites_to_ignore=(), solver=None)	**param metabolites_to_ignore**
get_all_subclasses(cls)	Given a variable or constraint class, get all the subclassses
chunk_sum(variables)	This functions handles the sum of many sympy variables by chunks, which
symbol_sum(variables)	``` python
get_solution_value_for_variables(solution, these_vars, index_by_reaction=False)	
compare_solutions(models)	returns the solution dictionnary for each cobra_model
evaluate_constraint_at_solution(constraint, solution)	**param expression**
get_active_use_variables(tmodel, solution)	Returns the active use variables in a solution. Use Variables are binary
get_direction_use_variables(tmodel, solution)	Returns the active use variables in a solution. Use Variables are binary
get_primal(tmodel, vartype, index_by_reactions=False)	Returns the primal value of the cobra_model for variables of a given type
strip_from_integer_variables(tmodel)	Removes all integer and binary variables of a cobra_model, to make it sample-able
copy_solver_configuration(source, target)	Copies the solver configuration from a source model to a target model

Attributes

ABCRequirePrefixMeta
ABCRequirePrefixMeta
op_replace_dict
BIGM
BIGM_THERMO
BIGM_DG
BIGM_P
EPSILON
SYMPY_ADD_CHUNKSIZE
INTEGER_VARIABLE_TYPES

pytfa.optim.**camel2underscores**(*name*)

pytfa.optim.**ABCRequirePrefixMeta**

class pytfa.optim.**GenericConstraint**(*expr*, *id_=''*, *model=None*, *hook=None*, *queue=False*, ***kwargs*)

Class to represent a generic constraint. The purpose is that the interface is instantiated on initialization, to follow the type of interface used by the problem, and avoid incompatibilities in optlang

Attributes:

id Used for DictList comprehension. Usually points back at a

enzyme or reaction id for ease of linking. Should be unique given a constraint type. :name: Should be a concatenation of the id and a prefix that is specific to the variable type. will be used to address the constraint at the solver level, and hence should be unique in the whole cobra_model :expr: the expression of the constraint (sympy.Expression subtype) :cobra_model: the cobra_model hook. :constraint: links directly to the cobra_model representation of tbe constraint

prefix

property __attrname__(*self*)
Name the attribute the instances will have Example: GenericConstraint -> generic_constraint :return:

get_interface(*self*, *expr*, *queue*)
Called upon completion of __init__, initializes the value of self.var, which is returned upon call, and stores the actual interfaced variable.

Returns instance of Variable from the problem

make_name(*self*)

Needs to be overridden by the subclass, concats the id with a prefix

Returns None

change_expr(*self*, *new_expr*, *sloppy=False*)

property expr(*self*)

property name(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property constraint(*self*)

property model(*self*)

__repr__(*self*)
Return repr(self).

class pytfa.optim.**ModelConstraint**(*model*, *expr*, *id_*, ***kwargs*)
Bases: *GenericConstraint*

Class to represent a variable attached to the model

prefix = MODC_

class pytfa.optim.**GeneConstraint**(*gene*, *expr*, ***kwargs*)
Bases: *GenericConstraint*

Class to represent a variable attached to a enzyme

prefix = GC_

property gene(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.optim.**ReactionConstraint**(*reaction*, *expr*, ***kwargs*)
Bases: *GenericConstraint*

Class to represent a variable attached to a reaction

prefix = RC_

property reaction(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.optim.**MetaboliteConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: *GenericConstraint*

Class to represent a variable attached to a enzyme

prefix = MC_

property metabolite(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.optim.**NegativeDeltaG**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics constraints.

G: - DGR_rxn + DGoRerr_Rxn + RT * StoichCoefProd1 * LC_prod1

- RT * StoichCoefProd2 * LC_prod2
- RT * StoichCoefSub1 * LC_subs1
- RT * StoichCoefSub2 * LC_subs2
- …

= 0

prefix = G_

class pytfa.optim.**NegativeDeltaGIneq**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics constraints.

G: DGoRerr_Rxn + RT * StoichCoefProd1 * LC_prod1

- RT * StoichCoefProd2 * LC_prod2
- RT * StoichCoefSub1 * LC_subs1
- RT * StoichCoefSub2 * LC_subs2
- …

< or > 0

prefix = GI_

class pytfa.optim.**ForwardDeltaGCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics coupling: DeltaG of reactions has to be DGR < 0 for the reaction to proceed forwards Looks like: FU_rxn: 1000 FU_rxn + DGR_rxn < 1000

prefix = FU_

class pytfa.optim.**BackwardDeltaGCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics coupling: DeltaG of reactions has to be DGR > 0 for the reaction to proceed backwards Looks like: BU_rxn: 1000 BU_rxn - DGR_rxn < 1000

prefix = BU_

class pytfa.optim.**ForwardDirectionCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a forward directionality coupling with thermodynamics on reaction variables Looks like : UF_rxn: F_rxn - M FU_rxn < 0

prefix = UF_

class pytfa.optim.**BackwardDirectionCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a backward directionality coupling with thermodynamics on reaction variables Looks like : UR_rxn: R_rxn - M RU_rxn < 0

prefix = UR_

class pytfa.optim.**SimultaneousUse**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a simultaneous use constraint on reaction variables Looks like: SU_rxn: FU_rxn + BU_rxn <= 1

prefix = SU_

class pytfa.optim.**DisplacementCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent the coupling to the thermodynamic displacement Looks like: Ln(Gamma) - (1/RT)*DGR_rxn = 0

prefix = DC_

class pytfa.optim.**ForbiddenProfile**(*model*, *expr*, *id_*, ***kwargs*)
Bases: *GenericConstraint*

Class to represent a forbidden net flux directionality profile Looks like: FU_rxn_1 + BU_rxn_2 + … + FU_rxn_n <= n-1

prefix = FP_

class pytfa.optim.**LinearizationConstraint**(*model*, *expr*, *id_*, ***kwargs*)
Bases: *ModelConstraint*

Class to represent a variable attached to a reaction

prefix = LC_

static from_constraints(*cons*, *model*)

class pytfa.optim.**LowerBoundLogConcentration**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = LB_

class pytfa.optim.**UpperBoundLogConcentration**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = UB_

class pytfa.optim.**NullspaceConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = NS_

class pytfa.optim.**PotentialConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = PC_

class pytfa.optim.**PotentialCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to rep couple DG to potential variable

prefix = PPC_

pytfa.optim.**camel2underscores**(*name*)

pytfa.optim.**ABCRequirePrefixMeta**

pytfa.optim.**op_replace_dict**

class pytfa.optim.**GenericVariable**(*id_=''*, *model=None*, *hook=None*, *queue=False*, *scaling_factor=1*, ***kwargs*)
Class to represent a generic variable. The purpose is that the interface is instantiated on initialization, to follow the type of interface used by the problem, and avoid incompatibilities in optlang

Attributes:

id Used for DictList comprehension. Usually points back at a

enzyme or reaction id for ease of linking. Should be unique given a variable type. :name: Should be a concatenation of the id and a prefix that is specific to the variable type. will be used to address the variable at the solver level, and hence should be unique in the whole cobra_model :cobra_model: the cobra_model hook. :variable: links directly to the cobra_model representation of tbe variable

prefix

property __attrname__(*self*)
Name the attribute the instances will have Example: GenericVariable -> generic_variable :return:

get_interface(*self*, *queue*)
Called upon completion of __init__, initializes the value of self.var, which is returned upon call, and stores the actual interfaced variable.

Returns instance of Variable from the problem

make_name(*self*)

Needs to be overridden by the subclass, concats the id with a prefix

Returns None

property name(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property variable(*self*)

property scaling_factor(*self*)

property unscaled(*self*)
If the scaling factor of quantity X is a, it is represented by the variable X_hat = X/a. This returns X = a.X_hat Useful for nondimensionalisation of variables and constraints

Returns The variable divided by its scaling factor

property value(*self*)

property unscaled_value(*self*)

property model(*self*)

property type(*self*)

test_consistency(*self*, *other*)
Tests whether a candidate to an operation is of the right type and is from the same problem

Parameters **other** – an object

Returns None

get_operand(*self*, *other*)
For operations, choose if the operand is a GenericVariable, in which we return its optlang variable, or something else (presumably a numeric) and we let optlang decide what to do

Parameters **other** –

Returns

__add__(*self*, *other*)
Adding either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__radd__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__sub__(*self*, *other*)
Substracting either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rsub__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__mul__(*self*, *other*)
Multiplying either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rmul__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__truediv__(*self*, *other*)
Dividing either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rtruediv__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

make_result(*self*, *new_variable*)
Returns a Sympy expression :param new_variable: :return:

__repr__(*self*)
Return repr(self).

pytfa.optim.**get_binary_type**()
FIX : We enforce type to be integer instead of binary, else optlang does not allow to set the binary variable bounds to anything other than (0,1) You might want to set it at (0,0) to enforce directionality for example

class pytfa.optim.**ModelVariable**(*model*, *id_*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a variable attached to the model

prefix = MODV_

class pytfa.optim.**GeneVariable**(*gene*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a gene variable

prefix = GV_

property gene(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.optim.**BinaryVariable**(*id_*, *model*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a generic binary variable

prefix = B_

class pytfa.optim.**IntegerVariable**(*id_*, *model*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a generic binary variable

prefix = I_

class pytfa.optim.**ReactionVariable**(*reaction*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a variable attached to a reaction

prefix = RV_

property reaction(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.optim.**MetaboliteVariable**(*metabolite*, ***kwargs*)
Bases: *GenericVariable*

Class to represent a variable attached to a enzyme

prefix = MV_

property metabolite(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.optim.**ForwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a forward use variable, a type of binary variable used to enforce forward directionality in reaction net fluxes

prefix = FU_

class pytfa.optim.**BackwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a backward use variable, a type of binary variable used to enforce backward directionality in reaction net fluxes

prefix = BU_

class pytfa.optim.**ForwardBackwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a type of binary variable used to tell whether the reaction is active or not such that:

FU + BU + BFUSE = 1

prefix = BFUSE_

class pytfa.optim.**LogConcentration**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a log concentration of a enzyme

prefix = LC_

class pytfa.optim.**DeltaGErr**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaGErr

prefix = DGE_

class pytfa.optim.**DeltaG**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaG

prefix = DG_

class pytfa.optim.**ThermoPotential**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a DeltaG

prefix = P_

class pytfa.optim.**DeltaGFormstd**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a DeltaGstf of formation

prefix = DGoF_

class pytfa.optim.**DeltaGstd**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaG^o (naught) - standard conditions

prefix = DGo_

class pytfa.optim.**ThermoDisplacement**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent the thermodynamic displacement of a reaction Gamma = -DeltaG/RT

prefix = LnGamma_

class pytfa.optim.**PosSlackVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a positive slack variable for relaxation problems

prefix = PosSlack_

class pytfa.optim.**NegSlackVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a negative slack variable for relaxation problems

prefix = NegSlack_

class pytfa.optim.**PosSlackVariableInt**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *IntegerVariable*

Class to represent a postive slack variable for relaxation problems using integer enforce forward directionality in reaction net fluxes

prefix = PosSlackInt_

class pytfa.optim.**NegSlackVariableInt**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *IntegerVariable*

Class to represent a postive slack variable for relaxation problems using integer enforce forward directionality in reaction net fluxes

prefix = NegSlackInt_

class pytfa.optim.**PosSlackLC**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a variable attached to a enzyme

prefix = PosSlackLC_

class pytfa.optim.**NegSlackLC**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a variable attached to a enzyme

prefix = NegSlackLC_

class pytfa.optim.**LinearizationVariable**(*model*, *id_*, ***kwargs*)
Bases: *ModelVariable*

Class to represent a product A*B when performing linearization of the model

prefix = LZ_

class pytfa.optim.**NegativeDeltaG**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics constraints.

G: - DGR_rxn + DGoRerr_Rxn + RT * StoichCoefProd1 * LC_prod1

- RT * StoichCoefProd2 * LC_prod2
- RT * StoichCoefSub1 * LC_subs1
- RT * StoichCoefSub2 * LC_subs2
- …

= 0

prefix = G_

pytfa.optim.**dg_relax_config**(*model*)

Parameters **model** –

Returns

pytfa.optim.**get_solution_value_for_variables**(*solution*, *these_vars*, *index_by_reaction=False*)

pytfa.optim.**chunk_sum**(*variables*)

This functions handles the sum of many sympy variables by chunks, which somehow increases the speed of the computation

You can test it in IPython: ```python a = sympy.symbols('a0:100') %timeit (sum(a)) # >>> 198 µs ± 11.4 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

b = sympy.symbols('b0:1000') %timeit (sum(b)) # >>> 1.85 ms ± 356 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

c = sympy.symbols('c0:3000') %timeit (sum(c)) # >>> 5min 7s ± 2.57 s per loop (mean ± std. dev. of 7 runs, 1 loop each) ```

See the github thread

Parameters **variables** –

Returns

pytfa.optim.**symbol_sum**(*variables*)

``` python a = symbols('a0:100')

%timeit Add(*a) # >>> 10000 loops, best of 3: 34.1 µs per loop

b = symbols('b0:1000')

%timeit Add(*b) # >>> 1000 loops, best of 3: 343 µs per loop

c = symbols('c0:3000')

%timeit Add(*c) # >>> 1 loops, best of 3: 1.03 ms per loop ```

See the github thread :param variables: :return:

**class** pytfa.optim.**PosSlackVariable**(*reaction*, *\*\*kwargs*)

Bases: *ReactionVariable*

Class to represent a positive slack variable for relaxation problems

**prefix = PosSlack_**

**class** pytfa.optim.**NegSlackVariable**(*reaction*, *\*\*kwargs*)

Bases: *ReactionVariable*

Class to represent a negative slack variable for relaxation problems

**prefix = NegSlack_**

**class** pytfa.optim.**DeltaGstd**(*reaction*, *\*\*kwargs*)

Bases: *ReactionVariable*

Class to represent a DeltaG^o (naught) - standard conditions

**prefix = DGo_**
```

class pytfa.optim.**LogConcentration**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a log concentration of a enzyme

prefix = LC_

class pytfa.optim.**NegSlackLC**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a variable attached to a enzyme

prefix = NegSlackLC_

class pytfa.optim.**PosSlackLC**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a variable attached to a enzyme

prefix = PosSlackLC_

class pytfa.optim.**PosSlackVariableInt**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *IntegerVariable*

Class to represent a postive slack variable for relaxation problems using integer enforce forward directionality in reaction net fluxes

prefix = PosSlackInt_

class pytfa.optim.**NegSlackVariableInt**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *IntegerVariable*

Class to represent a postive slack variable for relaxation problems using integer enforce forward directionality in reaction net fluxes

prefix = NegSlackInt_

pytfa.optim.**BIGM**

pytfa.optim.**BIGM_THERMO**

pytfa.optim.**BIGM_DG**

pytfa.optim.**BIGM_P**

pytfa.optim.**EPSILON**

pytfa.optim.**relax_dgo_gurobi**(*model*, *relax_obj_type=0*)

pytfa.optim.**relax_dgo**(*tmodel*, *reactions_to_ignore=()*, *solver=None*, *in_place=False*)

Parameters

- **t_tmodel** (*pytfa.thermo.ThermoModel:*) –
- **reactions_to_ignore** – Iterable of reactions that should not be relaxed
- **solver** – solver to use (e.g. 'optlang-glpk', 'optlang-cplex', 'optlang-gurobi'

Returns a cobra_model with relaxed bounds on standard Gibbs free energy

pytfa.optim.**relax_dgo_err**(*tmodel*, *reactions_to_ignore=()*, *max_sigma=3*, *solver=None*, *in_place=False*)

Parameters

- **t_tmodel** (*pytfa.thermo.ThermoModel:*) –

- **reactions_to_ignore** – Iterable of reactions that should not be relaxed
- **solver** – solver to use (e.g. 'optlang-glpk', 'optlang-cplex', 'optlang-gurobi'

Returns a cobra_model with relaxed bounds on standard Gibbs free energy

pytfa.optim.**relax_lc**(*tmodel*, *metabolites_to_ignore=()*, *solver=None*)

Parameters

- **metabolites_to_ignore** –
- **in_tmodel** (*pytfa.thermo.ThermoModel:*) –
- **min_objective_value** –

Returns

class pytfa.optim.**GenericConstraint**(*expr*, *id_=''*, *model=None*, *hook=None*, *queue=False*, ***kwargs*)

Class to represent a generic constraint. The purpose is that the interface is instantiated on initialization, to follow the type of interface used by the problem, and avoid incompatibilities in optlang

Attributes:

id Used for DictList comprehension. Usually points back at a

enzyme or reaction id for ease of linking. Should be unique given a constraint type. :name: Should be a concatenation of the id and a prefix that is specific to the variable type. will be used to address the constraint at the solver level, and hence should be unique in the whole cobra_model :expr: the expression of the constraint (sympy.Expression subtype) :cobra_model: the cobra_model hook. :constraint: links directly to the cobra_model representation of tbe constraint

prefix

property __attrname__(*self*)
Name the attribute the instances will have Example: GenericConstraint -> generic_constraint :return:

get_interface(*self*, *expr*, *queue*)
Called upon completion of __init__, initializes the value of self.var, which is returned upon call, and stores the actual interfaced variable.

Returns instance of Variable from the problem

make_name(*self*)

Needs to be overridden by the subclass, concats the id with a prefix

Returns None

change_expr(*self*, *new_expr*, *sloppy=False*)

property expr(*self*)

property name(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property constraint(*self*)

property model(*self*)

__repr__(*self*)
Return repr(self).

class pytfa.optim.**ForwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a forward use variable, a type of binary variable used to enforce forward directionality in reaction net fluxes

prefix = FU_

class pytfa.optim.**BackwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, *BinaryVariable*

Class to represent a backward use variable, a type of binary variable used to enforce backward directionality in reaction net fluxes

prefix = BU_

class pytfa.optim.**GenericVariable**(*id_=''*, *model=None*, *hook=None*, *queue=False*, *scaling_factor=1*, ***kwargs*)
Class to represent a generic variable. The purpose is that the interface is instantiated on initialization, to follow the type of interface used by the problem, and avoid incompatibilities in optlang

Attributes:

id Used for DictList comprehension. Usually points back at a

enzyme or reaction id for ease of linking. Should be unique given a variable type. :name: Should be a concatenation of the id and a prefix that is specific to the variable type. will be used to address the variable at the solver level, and hence should be unique in the whole cobra_model :cobra_model: the cobra_model hook. :variable: links directly to the cobra_model representation of tbe variable

prefix

property __attrname__(*self*)
Name the attribute the instances will have Example: GenericVariable -> generic_variable :return:

get_interface(*self*, *queue*)
Called upon completion of __init__, initializes the value of self.var, which is returned upon call, and stores the actual interfaced variable.

Returns instance of Variable from the problem

make_name(*self*)

Needs to be overridden by the subclass, concats the id with a prefix

Returns None

property name(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property variable(*self*)

property scaling_factor(*self*)

property unscaled(*self*)
If the scaling factor of quantity X is a, it is represented by the variable X_hat = X/a. This returns X = a.X_hat Useful for nondimensionalisation of variables and constraints

Returns The variable divided by its scaling factor

property value(*self*)

property unscaled_value(*self*)

property model(*self*)

property type(*self*)

test_consistency(*self*, *other*)
Tests whether a candidate to an operation is of the right type and is from the same problem

Parameters **other** – an object

Returns None

get_operand(*self*, *other*)
For operations, choose if the operand is a GenericVariable, in which we return its optlang variable, or something else (presumably a numeric) and we let optlang decide what to do

Parameters **other** –

Returns

__add__(*self*, *other*)
Adding either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__radd__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__sub__(*self*, *other*)
Substracting either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rsub__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__mul__(*self*, *other*)
Multiplying either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rmul__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

__truediv__(*self*, *other*)
Dividing either two variables together or a variable and a numeric results in a new variable :param other: :return: a new Generic Variable

__rtruediv__(*self*, *other*)
Take priority on symmetric arithmetic operation :param other: :return:

make_result(*self*, *new_variable*)
Returns a Sympy expression :param new_variable: :return:

__repr__(*self*)
Return repr(self).

pytfa.optim.**SYMPY_ADD_CHUNKSIZE = 100**

pytfa.optim.**INTEGER_VARIABLE_TYPES = ['binary', 'integer']**

pytfa.optim.**get_all_subclasses**(*cls*)
Given a variable or constraint class, get all the subclassses that inherit from it

Parameters **cls** –

Returns

pytfa.optim.**chunk_sum**(*variables*)

This functions handles the sum of many sympy variables by chunks, which somehow increases the speed of the computation

You can test it in IPython: ```python a = sympy.symbols('a0:100') %timeit (sum(a)) # >>> 198 µs ± 11.4 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

b = sympy.symbols('b0:1000') %timeit (sum(b)) # >>> 1.85 ms ± 356 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

c = sympy.symbols('c0:3000') %timeit (sum(c)) # >>> 5min 7s ± 2.57 s per loop (mean ± std. dev. of 7 runs, 1 loop each) ```

See the github thread

Parameters **variables** –

Returns

pytfa.optim.**symbol_sum**(*variables*)

``` python a = symbols('a0:100')

%timeit Add(*a) # >>> 10000 loops, best of 3: 34.1 µs per loop

b = symbols('b0:1000')

%timeit Add(*b) # >>> 1000 loops, best of 3: 343 µs per loop

c = symbols('c0:3000')

%timeit Add(*c) # >>> 1 loops, best of 3: 1.03 ms per loop ```

See the github thread :param variables: :return:

pytfa.optim.**get_solution_value_for_variables**(*solution*, *these_vars*, *index_by_reaction=False*)

pytfa.optim.**compare_solutions**(*models*)

returns the solution dictionnary for each cobra_model :param (iterable (pytfa.thermo.ThermoModel)) models: :return:

pytfa.optim.**evaluate_constraint_at_solution**(*constraint*, *solution*)

**Parameters**

- **expression** –
- **solution** – pandas.DataFrame, with index as variable names

**Returns**

pytfa.optim.**get_active_use_variables**(*tmodel*, *solution*)

Returns the active use variables in a solution. Use Variables are binary variables that control the directionality of the reaction

ex: FU_ACALDt BU_PFK

**Parameters**

- **tmodel** (*pytfa.core.ThermoModel*) –
- **solution** –
```

Returns

pytfa.optim.**get_direction_use_variables**(*tmodel*, *solution*)
Returns the active use variables in a solution. Use Variables are binary variables that control the directionality of the reaction The difference with get_active_use_variables is that variables with both UseVariables at 0 will return as going forwards. This is to ensure that the output size of the function is equal to the number of FDPs

ex: FU_ACALDt BU_PFK

Parameters

- **tmodel** (*pytfa.core.ThermoModel*) –
- **solution** –

Returns

pytfa.optim.**get_primal**(*tmodel*, *vartype*, *index_by_reactions=False*)
Returns the primal value of the cobra_model for variables of a given type :param tmodel: :param vartype: Class of variable. Ex: pytfa.optim.variables.ThermoDisplacement :param index_by_reactions: Set to true to get reaction names as index instead of

variables. Useful for Escher export

Returns

pytfa.optim.**strip_from_integer_variables**(*tmodel*)
Removes all integer and binary variables of a cobra_model, to make it sample-able :param tmodel: :return:

pytfa.optim.**copy_solver_configuration**(*source*, *target*)
Copies the solver configuration from a source model to a target model :param source: :param target: :return:

pytfa.redgem

Submodules

pytfa.redgem.debugging

Debugging

Module Contents

Functions

make_sink(met, ub=100, lb=0)	
add_BBB_sinks(model, biomass_rxn_id, ub=100, lb=0)	
check_BBB_production(model, biomass_rxn_id, verbose=False)	
min_BBB_uptake(model, biomass_rxn_id, min_growth_value, verbose=False)	

redgem.**make_sink**(*met*, *ub=100*, *lb=0*)

redgem.**add_BBB_sinks**(*model*, *biomass_rxn_id*, *ub=100*, *lb=0*)

redgem.**check_BBB_production**(*model*, *biomass_rxn_id*, *verbose=False*)

redgem.**min_BBB_uptake**(*model*, *biomass_rxn_id*, *min_growth_value*, *verbose=False*)

pytfa.redgem.lumpgem

Module Contents

Classes

FluxKO	Class to represent a variable attached to a reaction
UseOrKOInt	Class to represent a variable attached to a reaction
UseOrKOFlux	Class to represent a variable attached to a reaction
LumpGEM	A class encapsulating the LumpGEM algorithm

Functions

sum_reactions(rxn_dict, epsilon=1e-09, exclude=())	id_='summed_reaction',	Keys are reactions

Attributes

CPLEX
GUROBI
GLPK
DEFAULT_EPS
disambiguate
Lump

pytfa.redgem.lumpgem.**CPLEX = optlang-cplex**

pytfa.redgem.lumpgem.**GUROBI = optlang-gurobi**

pytfa.redgem.lumpgem.**GLPK = optlang-glpk**

pytfa.redgem.lumpgem.**DEFAULT_EPS = 1e-05**

pytfa.redgem.lumpgem.**disambiguate**

pytfa.redgem.lumpgem.**Lump**

exception pytfa.redgem.lumpgem.**InfeasibleExcept**(*status*, *feasibility*)
Bases: Exception

Common base class for all non-exit exceptions.

exception pytfa.redgem.lumpgem.**TimeoutExcept**(*time_limit*)
Bases: Exception

Common base class for all non-exit exceptions.

class pytfa.redgem.lumpgem.**FluxKO**(*reaction*, ***kwargs*)
Bases: pytfa.optim.variables.ReactionVariable, pytfa.optim.variables.BinaryVariable

Class to represent a variable attached to a reaction

prefix = KO_

class pytfa.redgem.lumpgem.**UseOrKOInt**(*reaction*, *expr*, ***kwargs*)
Bases: pytfa.optim.constraints.ReactionConstraint

Class to represent a variable attached to a reaction

prefix = UKI_

class pytfa.redgem.lumpgem.**UseOrKOFlux**(*reaction*, *expr*, ***kwargs*)
Bases: pytfa.optim.constraints.ReactionConstraint

Class to represent a variable attached to a reaction

prefix = UKF_

class pytfa.redgem.lumpgem.**LumpGEM**(*tfa_model*, *additional_core_reactions*, *params*)
A class encapsulating the LumpGEM algorithm

init_params(*self*)

_generate_usage_constraints(*self*)
Generate carbon intake related constraints for each non-core reaction For each reaction rxn : rxn.forward_variable + rxn.reverse_variable + activation_var * C_uptake < C_uptake

get_cofactor_adjusted_stoich(*self*, *rxn*)

_prepare_sinks(*self*)
For each BBB (reactant of the biomass reactions), generate a sink, i.e an unbalanced reaction BBB -> of which purpose is to enable the BBB to be output of the GEM :return: the dict {BBB: sink} containing every BBB (keys) and their associated sinks

_generate_objective(*self*)
Generate and add the maximization objective : set as many activation variables as possible to 1 When an activation variable is set to 1, the corresponding non-core reaction is deactivated

compute_lumps(*self*, *force_solve=False*, *method='OnePerBBB'*)
For each BBB (reactant of the biomass reaction), add the corresponding sink to the model, then optimize and lump the result into one lumped reaction :param force_solve: Indicates whether the computations must continue when one lumping yields a status "infeasible" :return: The dict {BBB: lump} containing every lumped reactions, associated to their BBBs

_lump_one_per_bbb(*self*, *met_BBB*, *sink*, *force_solve*, *minimize_subnet_fluxes=False*)

Parameters

- **met_BBB** –
- **sink** –

- **force_solve** –

Returns

_lump_min_plus_p(*self*, *met_BBB*, *sink*, *p*, *force_solve*)

Parameters

- **met_BBB** –
- **sink** –
- **force_solve** –

Returns

_build_lump(*self*, *met_BBB*, *sink*)
This function uses the current solution of self._tfa_model

Parameters

- **met_BBB** –
- **sink** –

Returns

pytfa.redgem.lumpgem.**sum_reactions**(*rxn_dict*, *id_='summed_reaction'*, *epsilon=1e-09*, *exclude=()*)
Keys are reactions Values are their multiplicative coefficient

pytfa.redgem.network_expansion

Model class

Module Contents

Classes

NetworkExpansion

class redgem.**NetworkExpansion**(*gem*, *core_subsystems*, *extracellular_system*, *cofactors*, *small_metabolites*, *inorganics*, *d*, *n*)

extract_subsystem_reactions(*self*, *subsystem*)
Extracts all reactions of a subsystem and stores them and their id in the corresponding dictionary.

Parameters **subsystem** – Name of the subsystem

Returns Extracted reactions

extract_subsystem_metabolites(*self*, *subsystem*)
Extracts all metabolites of a subsystem and stores them and their id in the corresponding dictionary.

Parameters **subsystem** – Name of the subsystem

Returns Extracted metabolites

create_new_stoichiometric_matrix(*self*)
Extracts the new graph without the small metabolites, inorganics and cofactor pairs.

Returns Networkx graph of the new network

breadth_search_subsystems_paths_length_d(*self*, *subsystem_i*, *subsystem_j*, *d*)
Breadth first search from each metabolite in subsystem i with special stop conditions during exploration for paths of length d.

This function explores the graph through allowed paths only : this path can't go through subsystem i or j but must start in i and end in j. The length of each path found is d.

Parameters

- **subsystem_i** – Source subsystem
- **subsystem_j** – Destination subsystem
- **d** – Path length desired

Returns None

is_node_allowed(*self*, *node*, *i*, *explored*, *subsystem_i*, *subsystem_j*, *d*)
Checks whether or not a metabolite is allowed for the current path.

The new node is added if it is not already explored, if it is not in the source subsystem, and if it is not in the destination subsystem, except if it is the last round of exploration

Parameters

- **node** – Metabolite id
- **i** – Current step
- **explored** – Explored node for this path
- **subsystem_i** – Source subsystem
- **subsystem_j** – Destination subsystem
- **d** – Path length desired

Returns Boolean answering the question

retrieve_all_paths(*self*, *dest_node*, *src_node*, *ancestors*, *init_dict=True*)
Retrieves all paths between a source metabolite and a destination metabolite after a breadth first search.

This function is a recursive function, which makes use of dynamic programming to reduce its complexity. It uses self._path_dict to store already computed data.

Parameters

- **dest_node** – Destination metabolite
- **src_node** – Source metabolite
- **ancestors** – Dictionary with ancestors found during the search
- **init_dict** – Boolean, for function initialisation

Returns A list of all paths as tuples

retrieve_intermediate_metabolites_and_reactions(*self*, *paths*, *subsystem_i*, *subsystem_j*, *d*)
Retrieves and stores intermediate metabolites and reactions (i.e. M_{i,j}, R_{i,j}, M_{i,i} and R_{i,i}).

This function adds all reactions contained in these paths, and all metabolites between

Parameters

- **paths** – List of paths between subsystems
- **subsystem_i** – Source subsystem
- **subsystem_j** – Destination subsystem
- **d** – Path length

Returns None

find_min_distance_between_subsystems(*self*)
Find minimal distance between each subsystems in both directions

Returns Dict with distances

breadth_search_extracellular_system_paths(*self*, *subsystem*, *n*)
Breadth first search from each metabolite in the extracellular system with special stop conditions during exploration for paths of length n.

This function explores the graph through allowed paths only : this path can't go through the extracellular system or the subsystem but must start in the extracellular system and end in the subsystem. The length of each path found is n.

Parameters

- **subsystem** – Destination subsystem
- **n** – Path length desired

Returns None

is_node_allowed_extracellular(*self*, *node*, *i*, *explored*, *subsystem*, *n*)
Checks whether or not a metabolite is allowed for the current path.

The new node is added if it is not already explored, if it is not in the extracellular system, and if it is not in the destination subsystem except if it is the last round of exploration

Parameters

- **node** – Metabolite id
- **i** – Current step
- **explored** – Explored node for this path
- **subsystem** – Destination subsystem
- **n** – Path length desired

Returns Boolean answering the question

retrieve_intermediate_extracellular_metabolites_and_reactions(*self*, *paths*, *subsystem*, *n*)
Retrieves and stores intermediate metabolites and reactions for the extracellular system

This function adds all reactions contained in these paths, and all metabolites between

Parameters

- **paths** – List of paths
- **subsystem** – Destination subsystem
- **n** – Path length

Returns None

run_between_all_subsystems(*self*)
Retrieve subsystem and intermediate reactions and metabolites.

Returns None

run_extracellular_system(*self*)
Retrieve intermediate reactions and metabolites for the extracellular system

Returns None

extract_sub_network(*self*, *additional_reactions*)
Extracts the reduced gem.

Returns None

run(*self*, *additional_reactions=[]*)
Runs RedGEM.

Returns None

pytfa.redgem.redgem

Model class

Module Contents

Classes

RedGEM

Functions

add_lump(model, lump_object, id_suffix='')

class redgem.**RedGEM**(*gem*, *parameters_path*, *inplace=False*)

read_parameters(*self*, *parameters_path*)

fill_default_params(*self*)

set_solver(*self*)

run(*self*)

_extract_inorganics(*self*)
Extract inorganics from self._gem based on their formula

Returns list of inorganics metabolites

redgem.**add_lump**(*model*, *lump_object*, *id_suffix=''*)

pytfa.redgem.utils

Module Contents

Functions

remove_blocked_reactions(model)	
round(value, epsilon)	
trim_epsilon_mets(met_dict, epsilon, model=None)	
set_medium(model, medium_dict, inplace)	

pytfa.redgem.utils.**remove_blocked_reactions**(*model*)

pytfa.redgem.utils.**round**(*value*, *epsilon*)

pytfa.redgem.utils.**trim_epsilon_mets**(*met_dict*, *epsilon*, *model=None*)

pytfa.redgem.utils.**set_medium**(*model*, *medium_dict*, *inplace*)

pytfa.thermo

Thermodynamic analysis for Flux-Based Analysis

Submodules

pytfa.thermo.equilibrator

Thermodynamic information for metabolites from eQuilibrator.

Module Contents

Functions

build_thermo_from_equilibrator(model, T=TEMPERATURE_0)	Build *thermo_data* structure from a cobra Model.
compute_dGf(compound, cc)	Get Gf from equilibrator *compound*.
compound_to_entry(compound, cc)	Build thermo structure entry from a *equilibrator_cache.Compound*.

Attributes

ccache
logger

pytfa.**ccache**

pytfa.**logger**

pytfa.**build_thermo_from_equilibrator**(*model*, *T=TEMPERATURE_0*)
Build *thermo_data* structure from a cobra Model.

The structure of the returned dictionary is specified in the pyTFA [documentation](https://pytfa.readthedocs.io/en/latest/thermoDB.html).

Parameters **model** – cobra.Model

Return thermo_data dict to be passed as argument to initialize a *ThermoModel*.

pytfa.**compute_dGf**(*compound*, *cc*)
Get Gf from equilibrator *compound*.

pytfa.**compound_to_entry**(*compound*, *cc*)
Build thermo structure entry from a *equilibrator_cache.Compound*.

eQuilibrator works with Component Contribution instead of groups, so it is not possible to generate cues from it.

Parameters **compound** – equilibrator_cache.Compound

Returns dict with keys ['deltaGf_std', 'deltaGf_err', 'error', 'struct_cues', 'id', 'pKa', 'mass_std', 'charge_std', 'nH_std', 'name', 'formula', 'other_names']

pytfa.thermo.metabolite

Thermodynamic computations for metabolites

Module Contents

Classes

MetaboliteThermo	A class representing the thermodynamic values of a enzyme

Attributes

CPD_PROTON
DEFAULT_VAL

pytfa.**CPD_PROTON = cpd00067**

pytfa.**DEFAULT_VAL**

class pytfa.**MetaboliteThermo**(*metData*, *pH*, *ionicStr*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*, *debye_huckel_b=std.DEBYE_HUCKEL_B_0*, *thermo_unit='kJ/mol'*, *debug=False*)

A class representing the thermodynamic values of a enzyme

Parameters

- **metData** (*dict*) – A dictionary containing the values for the enzyme, from the thermodynamic database
- **pH** (*float*) – The pH of the enzyme's compartment
- **ionicStr** (*float*) – The ionic strength of the enzyme's compartment
- **temperature** –
- **min_ph** –
- **max_ph** –
- **debye_huckel_b** –
- **thermo_unit** (*string*) – The unit used in *metData*'s values
- **debug** (*bool*) – *Optional* If set to `True`, some debugging values will be printed. This is only useful for development or debugging purposes.

Note: The values are automatically computed on class creation. Usually you don't have to call any methods defined by this class, but only to access the attributes you need.

The available attributes are :

Since the reactions expose similar values through a dictionnary, it is better to access the attributes aforementionned of this class as if it was a dictionnary : `enzyme.thermo['pH']`.

__getitem__(*self*, *key*)

__repr__(*self*)
Return repr(self).

keys(*self*)

values(*self*)

items(*self*)

__cmp__(*self*, *dict_*)

__contains__(*self*, *item*)

__iter__(*self*)

__unicode__(*self*)

calcDGis(*self*)
Calculate the transformed Gibbs energy of formation of specie with given pH and ionic strength using formula given by Goldberg and Tewari, 1991

Equation 4.5-6 in Alberty's book

Returns DG_is for the enzyme

Return type float

calcDGsp(*self*)
Calculate the transformed Gibbs energy of formation of specie with given pH and ionic strength using formula given by Goldberg and Tewari, 1991

Equation 4.4-10 in Alberty's book

Returns DG_sp for the enzyme

Return type float

calc_potential(*self*)
Calculate the binding polynomial of a specie, with pK values

Returns The potential of the enzyme

Return type float

get_pka(*self*)
Get the pKas of the enzyme

Returns The pKas of the enzyme

Return type list(float)

_calc_pka(*self*, *pka*, *sigmanusq*)

calcDGspA(*self*)
Calculates deltaGf, charge and nH of the specie when it is at least protonated state based on MFAToolkit compound data for the pKa values within the range considered (MIN_pH to MAX_pH).

These values are used as the starting point for Alberty's calculations.

Returns deltaGspA, sp_charge and sp_nH

Return type tuple(float, float, int)

pytfa.thermo.reaction

Thermodynamic computations for reactions

Module Contents

Functions

calcDGtpt_rhs(reaction, compartmentsData, thermo_units)	Calculates the RHS of the deltaG constraint, i.e. the sum of the
calcDGR_cues(reaction, reaction_cues_data)	Calculates the deltaG reaction and error of the reaction using the
calcDGF_cues(cues, reaction_cues_data)	Calculates the deltaG formation and error of the compound using its
get_debye_huckel_b(T)	The Debye-Huckel A and B do depend on the temperature

pytfa.**calcDGtpt_rhs**(*reaction*, *compartmentsData*, *thermo_units*)

Calculates the RHS of the deltaG constraint, i.e. the sum of the non-concentration terms

Parameters

- **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to compute the data for
- **compartmentsData** (*dict(float)*) – Data of the compartments of the cobra_model
- **thermo_units** (*str*) – The thermodynamic database of the cobra_model

Returns deltaG_tpt and the breakdown of deltaG_tpt

Return type tuple(float, dict(float))

Example: ATP Synthase reaction:

```
reaction = cpd00008 + 4 cpd00067 + cpd00009 <=> cpd00002 + 3 cpd00067 + cpd00001
compartments =  'c'       'e'        'c'           'c'         'c'         'c'
```

If there are any metabolites with unknown energies then returns `(0, None)`.

pytfa.**calcDGR_cues**(*reaction*, *reaction_cues_data*)

Calculates the deltaG reaction and error of the reaction using the constituent structural cues changes and returns also the error if any.

Parameters

- **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to compute deltaG for
- **reaction_cues_data** (*dict*) –

Returns

deltaGR, error on deltaGR, the cues in the reaction (keys of the dictionnary) and their indices (values of the dictionnary), and the error code if any.

If everything went right, the error code is an empty string

Return type tuple(float, float, dict(float), str)

pytfa.**calcDGF_cues**(*cues*, *reaction_cues_data*)

Calculates the deltaG formation and error of the compound using its constituent structural cues.

Parameters

- **cues** (*list(str)*) – A list of cues' names

- **reaction_cues_data** (*dict*) –

Returns deltaG formation, the error on deltaG formation, and a dictionnary with the cues' names as key and their coefficient as value

Return type tuple(float, float, dict(float))

pytfa.**get_debye_huckel_b**(*T*)

The Debye-Huckel A and B do depend on the temperature As for now though they are returned as a constant (value at 298.15K)

Parameters **T** – Temperature in Kelvin

Returns Debye_Huckel_B

pytfa.thermo.std

Standard constants definitions

Module Contents

pytfa.thermo.std.**TEMPERATURE_0 = 298.15**

pytfa.thermo.std.**MIN_PH = 3**

pytfa.thermo.std.**MAX_PH = 9**

pytfa.thermo.std.**DEBYE_HUCKEL_B_0 = 1.6**

pytfa.thermo.std.**DEBYE_HUCKEL_A**

pytfa.thermo.std.**A_LOT = 5000**

pytfa.thermo.std.**A_LITTLE = 0.5**

pytfa.thermo.std.**A_COUPLE = 2.5**

pytfa.thermo.std.**MANY = 100**

pytfa.thermo.tmodel

Thermodynamic cobra_model class and methods definition

Module Contents

Classes

ThermoModel	A class representing a cobra_model with thermodynamics information

Attributes

BIGM
BIGM_THERMO
BIGM_DG
BIGM_P
EPSILON
MAX_STOICH

pytfa.**BIGM**

pytfa.**BIGM_THERMO**

pytfa.**BIGM_DG**

pytfa.**BIGM_P**

pytfa.**EPSILON**

pytfa.**MAX_STOICH = 10**

class pytfa.**ThermoModel**(*thermo_data=None*, *model=Model()*, *name=None*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*)

Bases: pytfa.core.model.LCSBModel, cobra.Model

A class representing a cobra_model with thermodynamics information

_init_thermo(*self*)

normalize_reactions(*self*)

Find reactions with important stoichiometry and normalizes them :return:

_prepare_metabolite(*self*, *met*)

Parameters **met** –

Returns

_prepare_reaction(*self*, *reaction*)

prepare(*self*)

Prepares a COBRA toolbox cobra_model for TFBA analysis by doing the following:

1. checks if a reaction is a transport reaction
2. checks the ReactionDB for Gibbs energies of formation of metabolites
3. computes the Gibbs energies of reactions

_convert_metabolite(*self*, *met*, *add_potentials*, *verbose*)

Given a enzyme, proceeds to create the necessary variables and constraints for thermodynamics-based modeling

Parameters **met** –

Returns

_convert_reaction(*self*, *rxn*, *add_potentials*, *add_displacement*, *verbose*)

Parameters

- **rxn** –
- **add_potentials** –
- **add_displacement** –
- **verbose** –

Returns

convert(*self*, *add_potentials=False*, *add_displacement=False*, *verbose=True*)
Converts a cobra_model into a tFBA ready cobra_model by adding the thermodynamic constraints required

> **Warning:** This function requires you to have already called `prepare()`, otherwise it will raise an Exception !

print_info(*self*, *specific=False*)
Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memo*)

Parameters **memo** –

Returns

copy(*self*)
Needs to be reimplemented, as our objects have complicated hierarchy :return:

pytfa.thermo.utils

Some tools around COBRApy models used by pyTFA

Module Contents

Functions

check_reaction_balance(reaction, proton=None)	Check the balance of a reaction, and eventually add protons to balance
find_transported_mets(reaction)	Get a list of the transported metabolites of the reaction.
check_transport_reaction(reaction)	Check if a reaction is a transport reaction
is_same_stoichiometry(this_reaction, that_reaction)	
is_exchange(rxn)	
get_reaction_compartment(reaction)	Get the compartment of a reaction to then prepare it for conversion.

Attributes

Formula_regex

pytfa.**Formula_regex**

pytfa.**check_reaction_balance**(*reaction*, *proton=None*)

Check the balance of a reaction, and eventually add protons to balance it

Parameters

- **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to check the balance of.
- **proton** (*cobra.thermo.metabolite.Metabolite*) – *Optional* The proton to add to the reaction to balance it.

Returns

The balance of the reaction :

- `drain flux`
- `missing structures`
- `balanced`
- `N protons added to reactants` with `N` a `float`
- `N protons added to products` with `N` a `float`
- `missing atoms`

Return type str

If `proton` is provided, this function will try to balance the equation with it, and return the result.

If no `proton` is provided, this function will not try to balance the equation.

> **Warning:** This function does not verify if `proton` is in the correct compartment, so make sure you provide the `proton` belonging to the correct compartment !

pytfa.**find_transported_mets**(*reaction*)

Get a list of the transported metabolites of the reaction.

Parameters **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to get the transported metabolites of

Returns

A dictionnary of the transported metabolites. The index corresponds to the seed_id of the transported enzyme

The value is a dictionnairy with the following values:

- **coeff (float):** The stoechiomectric coefficient of the enzyme
- **reactant (cobra.thermo.enzyme.Metabolite):** The reactant of the reaction corresponding to the transported enzyme
- **product (cobra.thermo.enzyme.Metabolite):** The product of the reaction corresponding to the transported enzyme

A transported enzyme is defined as a enzyme which is a product and a reactant of a reaction. We can distinguish them thanks to their seed_ids.

pytfa.**check_transport_reaction**(*reaction*)
Check if a reaction is a transport reaction

Parameters **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to check

Returns Whether the reaction is a transport reaction or not

Return type bool

A transport reaction is defined as a reaction that has the same compound as a reactant and a product. We can distinguish them thanks to their seed_ids. If they have one If not, use met_ids and check if they are the same, minus compartment

pytfa.**is_same_stoichiometry**(*this_reaction*, *that_reaction*)

pytfa.**is_exchange**(*rxn*)

pytfa.**get_reaction_compartment**(*reaction*)
Get the compartment of a reaction to then prepare it for conversion.

Package Contents

Classes

LCSBModel	Helper class that provides a standard way to create an ABC using
MetaboliteThermo	A class representing the thermodynamic values of a enzyme
SimultaneousUse	Class to represent a simultaneous use constraint on reaction variables
NegativeDeltaG	Class to represent thermodynamics constraints.
BackwardDeltaGCoupling	Class to represent thermodynamics coupling: DeltaG of reactions has to be
ForwardDeltaGCoupling	Class to represent thermodynamics coupling: DeltaG of reactions has to be
BackwardDirectionCoupling	Class to represent a backward directionality coupling with thermodynamics on
ForwardDirectionCoupling	Class to represent a forward directionality coupling with thermodynamics on
ReactionConstraint	Class to represent a variable attached to a reaction
MetaboliteConstraint	Class to represent a variable attached to a enzyme
DisplacementCoupling	Class to represent the coupling to the thermodynamic displacement
PotentialConstraint	Class to represent a forbidden net flux directionality profile
PotentialCoupling	Class to rep couple DG to potential variable
ThermoDisplacement	Class to represent the thermodynamic displacement of a reaction
DeltaGstd	Class to represent a DeltaG^o (naught) - standard conditions
DeltaG	Class to represent a DeltaG

continues on next page

Table 58 – continued from previous page

ForwardUseVariable	Class to represent a forward use variable, a type of binary variable used to
BackwardUseVariable	Class to represent a backward use variable, a type of binary variable used
LogConcentration	Class to represent a log concentration of a enzyme
ReactionVariable	Class to represent a variable attached to a reaction
MetaboliteVariable	Class to represent a variable attached to a enzyme
ThermoPotential	Class to represent a DeltaG
DeltaGFormstd	Class to represent a DeltaGstf of formation
ThermoModel	A class representing a cobra_model with thermodynamics information
MetaboliteThermo	A class representing the thermodynamic values of a enzyme

Functions

calcDGtpt_rhs(reaction, compartmentsData, thermo_units)	Calculates the RHS of the deltaG constraint, i.e. the sum of the
calcDGR_cues(reaction, reaction_cues_data)	Calculates the deltaG reaction and error of the reaction using the
get_debye_huckel_b(T)	The Debye-Huckel A and B do depend on the temperature
check_reaction_balance(reaction, proton=None)	Check the balance of a reaction, and eventually add protons to balance
check_transport_reaction(reaction)	Check if a reaction is a transport reaction
find_transported_mets(reaction)	Get a list of the transported metabolites of the reaction.
get_bistream_logger(name)	Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages

Attributes

BIGM
BIGM_THERMO
BIGM_DG
BIGM_P
EPSILON
MAX_STOICH

class pytfa.thermo.**LCSBModel**(*model*, *name*, *sloppy=False*)

Bases: abc.ABC

Helper class that provides a standard way to create an ABC using inheritance.

abstract copy(*self*)
Needs to be reimplemented, as our objects have complicated hierarchy :return:

print_info(*self*)
Print information and counts for the cobra_model :return:

add_variable(*self*, *kind*, *hook*, *queue=False*, ***kwargs*)
Add a new variable to a COBRApy cobra_model.

Parameters

- **kind** –
- **hook** (*string,cobra.Reaction*) – Either a string representing the name of the variable to add to the cobra_model, or a reaction object if the kind allows it

Returns The created variable

Return type optlang.interface.Variable

add_constraint(*self*, *kind*, *hook*, *expr*, *queue=False*, ***kwargs*)
Add a new constraint to a COBRApy cobra_model

Parameters

- **kind** –
- **hook** (*string,cobra.Reaction*) – Either a string representing the name of the variable to add to the cobra_model, or a reaction object if the kind allows it
- **expr** (*sympy.thermo.expr.Expr*) – The expression of the constraint

Returns The created constraint

Return type optlang.interface.Constraint

remove_reactions(*self*, *reactions*, *remove_orphans=False*)

remove_metabolites(*self*, *metabolite_list*, *destructive=False*)

_remove_associated_consvar(*self*, *all_cons_subclasses*, *all_var_subclasses*, *collection*)
Removes both the constraints and variables associated to an element, as long as it was used as a hook in the cons/var declaration. For example, upon removing a reaction, also removes its associated deltaG variables and coupling constraints

remove_variable(*self*, *var*)
Removes a variable

Parameters **var** –

Returns

remove_constraint(*self*, *cons*)
Removes a constraint

Parameters **cons** –

Returns

_push_queue(*self*)
updates the constraints and variables of the model with what's in the queue :return:

regenerate_variables(*self*)
Generates references to the cobra_model's constraints in self._var_dict as tab-searchable attributes of the thermo cobra_model :return:

regenerate_constraints(*self*)
Generates references to the cobra_model's constraints in self._cons_dict as tab-searchable attributes of the thermo cobra_model :return:

repair(*self*)
Updates references to variables and constraints :return:

get_primal(*self*, *vartype*, *index_by_reactions=False*)
Returns the primal value of the cobra_model for variables of a given type

Parameters

- **index_by_reactions** –
- **vartype** – Class of variable. Ex: pytfa.optim.variables.ThermoDisplacement

Returns

get_solution(*self*)
Overrides the cobra.thermo.solution method, to also get the supplementary variables we added to the cobra_model

- `solution.fluxes` in *cobrapy* is a transformed version of the solver output, as it actually calculates the _net_ flux of each reaction by substracting the reverse variable value to the forward variable value. This should be used anytime one needs the actual flux value
- `solution.raw` is a clear copy of the solver output. From there one can access the value at solution for all the variables of the problem. However, looking for a reaction ID in there will only give the _forward_ flux. This should be used for any other variable than fluxes.
- `solution.values` yields variables multiplied by their scaling factor (1 by default). Useful if you operated scaling on your equations for numerical reasons. This does _not_ include fluxes

Returns

optimize(*self*, *objective_sense=None*, ***kwargs*)
Call the Model.optimize function (which really is but an interface to the solver's. Catches SolverError in the case of no solutions. Passes down supplementary keyword arguments (see cobra.thermo.Model.optimize) :type objective_sense: 'min' or 'max'

slim_optimize(*self*, **args*, ***kwargs*)

get_constraints_of_type(*self*, *constraint_type*)
Convenience function that takes as input a constraint class and returns all its instances within the cobra_model

Parameters **constraint_type** –

Returns

get_variables_of_type(*self*, *variable_type*)
Convenience function that takes as input a variable class and returns all its instances within the cobra_model

Parameters **variable_type** –

Returns

class pytfa.thermo.**MetaboliteThermo**(*metData*, *pH*, *ionicStr*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*, *debye_huckel_b=std.DEBYE_HUCKEL_B_0*, *thermo_unit='kJ/mol'*, *debug=False*)
A class representing the thermodynamic values of a enzyme

Parameters

- **metData** (*dict*) – A dictionary containing the values for the enzyme, from the thermodynamic database
- **pH** (*float*) – The pH of the enzyme's compartment
- **ionicStr** (*float*) – The ionic strength of the enzyme's compartment
- **temperature** –
- **min_ph** –
- **max_ph** –
- **debye_huckel_b** –
- **thermo_unit** (*string*) – The unit used in *metData*'s values
- **debug** (*bool*) – *Optional* If set to `True`, some debugging values will be printed. This is only useful for development or debugging purposes.

Note: The values are automatically computed on class creation. Usually you don't have to call any methods defined by this class, but only to access the attributes you need.

The available attributes are :

Since the reactions expose similar values through a dictionnary, it is better to access the attributes aforementionned of this class as if it was a dictionnary : `enzyme.thermo['pH']`.

__getitem__(*self*, *key*)

__repr__(*self*)
Return repr(self).

keys(*self*)

values(*self*)

items(*self*)

__cmp__(*self*, *dict_*)

__contains__(*self*, *item*)

__iter__(*self*)

__unicode__(*self*)

calcDGis(*self*)
Calculate the transformed Gibbs energy of formation of specie with given pH and ionic strength using formula given by Goldberg and Tewari, 1991

Equation 4.5-6 in Alberty's book

Returns DG_is for the enzyme

Return type float

calcDGsp(*self*)
Calculate the transformed Gibbs energy of formation of specie with given pH and ionic strength using formula given by Goldberg and Tewari, 1991

Equation 4.4-10 in Alberty's book

Returns DG_sp for the enzyme

Return type float

calc_potential(*self*)
Calculate the binding polynomial of a specie, with pK values

Returns The potential of the enzyme

Return type float

get_pka(*self*)
Get the pKas of the enzyme

Returns The pKas of the enzyme

Return type list(float)

_calc_pka(*self*, *pka*, *sigmanusq*)

calcDGspA(*self*)
Calculates deltaGf, charge and nH of the specie when it is at least protonated state based on MFAToolkit compound data for the pKa values within the range considered (MIN_pH to MAX_pH).

These values are used as the starting point for Alberty's calculations.

Returns deltaGspA, sp_charge and sp_nH

Return type tuple(float, float, int)

pytfa.thermo.**calcDGtpt_rhs**(*reaction*, *compartmentsData*, *thermo_units*)
Calculates the RHS of the deltaG constraint, i.e. the sum of the non-concentration terms

Parameters

- **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to compute the data for
- **compartmentsData** (*dict(float)*) – Data of the compartments of the cobra_model
- **thermo_units** (*str*) – The thermodynamic database of the cobra_model

Returns deltaG_tpt and the breakdown of deltaG_tpt

Return type tuple(float, dict(float))

Example: ATP Synthase reaction:

```
reaction = cpd00008 + 4 cpd00067 + cpd00009 <=> cpd00002 + 3 cpd00067 + cpd00001
compartments =  'c'       'e'        'c'           'c'         'c'         'c'
```

If there are any metabolites with unknown energies then returns `(0, None)`.

pytfa.thermo.**calcDGR_cues**(*reaction*, *reaction_cues_data*)
Calculates the deltaG reaction and error of the reaction using the constituent structural cues changes and returns also the error if any.

Parameters

- **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to compute deltaG for
- **reaction_cues_data** (*dict*) –

Returns

deltaGR, error on deltaGR, the cues in the reaction (keys of the dictionnary) and their indices (values of the dictionnary), and the error code if any.

If everything went right, the error code is an empty string

Return type tuple(float, float, dict(float), str)

pytfa.thermo.**get_debye_huckel_b**(*T*)

The Debye-Huckel A and B do depend on the temperature As for now though they are returned as a constant (value at 298.15K)

Parameters **T** – Temperature in Kelvin

Returns Debye_Huckel_B

pytfa.thermo.**check_reaction_balance**(*reaction*, *proton=None*)

Check the balance of a reaction, and eventually add protons to balance it

Parameters

- **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to check the balance of.
- **proton** (*cobra.thermo.metabolite.Metabolite*) – *Optional* The proton to add to the reaction to balance it.

Returns

The balance of the reaction :

- `drain flux`
- `missing structures`
- `balanced`
- `N protons added to reactants` with `N` a `float`
- `N protons added to products` with `N` a `float`
- `missing atoms`

Return type str

If `proton` is provided, this function will try to balance the equation with it, and return the result.

If no `proton` is provided, this function will not try to balance the equation.

> **Warning:** This function does not verify if `proton` is in the correct compartment, so make sure you provide the `proton` belonging to the correct compartment !

pytfa.thermo.**check_transport_reaction**(*reaction*)

Check if a reaction is a transport reaction

Parameters **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to check

Returns Whether the reaction is a transport reaction or not

Return type bool

A transport reaction is defined as a reaction that has the same compound as a reactant and a product. We can distinguish them thanks to their seed_ids. If they have one If not, use met_ids and check if they are the same, minus compartment

pytfa.thermo.**find_transported_mets**(*reaction*)

Get a list of the transported metabolites of the reaction.

Parameters **reaction** (*cobra.thermo.reaction.Reaction*) – The reaction to get the transported metabolites of

Returns

A dictionnary of the transported metabolites. The index corresponds to the seed_id of the transported enzyme

The value is a dictionnairy with the following values:

- **coeff (float):** The stoechiomectric coefficient of the enzyme
- **reactant (cobra.thermo.enzyme.Metabolite):** The reactant of the reaction corresponding to the transported enzyme
- **product (cobra.thermo.enzyme.Metabolite):** The product of the reaction corresponding to the transported enzyme

A transported enzyme is defined as a enzyme which is a product and a reactant of a reaction. We can distinguish them thanks to their seed_ids.

class pytfa.thermo.**SimultaneousUse**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a simultaneous use constraint on reaction variables Looks like: SU_rxn: FU_rxn + BU_rxn <= 1

prefix = SU_

class pytfa.thermo.**NegativeDeltaG**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics constraints.

G: - DGR_rxn + DGoRerr_Rxn + RT * StoichCoefProd1 * LC_prod1

- RT * StoichCoefProd2 * LC_prod2
- RT * StoichCoefSub1 * LC_subs1
- RT * StoichCoefSub2 * LC_subs2
- ...

= 0

prefix = G_

class pytfa.thermo.**BackwardDeltaGCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics coupling: DeltaG of reactions has to be DGR > 0 for the reaction to proceed backwards Looks like: BU_rxn: 1000 BU_rxn - DGR_rxn < 1000

prefix = BU_

class pytfa.thermo.**ForwardDeltaGCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent thermodynamics coupling: DeltaG of reactions has to be DGR < 0 for the reaction to proceed forwards Looks like: FU_rxn: 1000 FU_rxn + DGR_rxn < 1000

prefix = FU_

class pytfa.thermo.**BackwardDirectionCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a backward directionality coupling with thermodynamics on reaction variables Looks like : UR_rxn: R_rxn - M RU_rxn < 0

prefix = UR_

class pytfa.thermo.**ForwardDirectionCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent a forward directionality coupling with thermodynamics on reaction variables Looks like : UF_rxn: F_rxn - M FU_rxn < 0

prefix = UF_

class pytfa.thermo.**ReactionConstraint**(*reaction*, *expr*, ***kwargs*)
Bases: GenericConstraint

Class to represent a variable attached to a reaction

prefix = RC_

property reaction(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.thermo.**MetaboliteConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: GenericConstraint

Class to represent a variable attached to a enzyme

prefix = MC_

property metabolite(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.thermo.**DisplacementCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to represent the coupling to the thermodynamic displacement Looks like: Ln(Gamma) - (1/RT)*DGR_rxn = 0

prefix = DC_

class pytfa.thermo.**PotentialConstraint**(*metabolite*, *expr*, ***kwargs*)
Bases: *MetaboliteConstraint*

Class to represent a forbidden net flux directionality profile Looks like: DG_rxn_1 + DG_rxn_2 + … + DG_rxn_n == 0

prefix = PC_

class pytfa.thermo.**PotentialCoupling**(*reaction*, *expr*, ***kwargs*)
Bases: *ReactionConstraint*

Class to rep couple DG to potential variable

prefix = PPC_

class pytfa.thermo.**ThermoDisplacement**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent the thermodynamic displacement of a reaction Gamma = -DeltaG/RT

prefix = LnGamma_

class pytfa.thermo.**DeltaGstd**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaG^o (naught) - standard conditions

prefix = DGo_

class pytfa.thermo.**DeltaG**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*

Class to represent a DeltaG

prefix = DG_

class pytfa.thermo.**ForwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, BinaryVariable

Class to represent a forward use variable, a type of binary variable used to enforce forward directionality in reaction net fluxes

prefix = FU_

class pytfa.thermo.**BackwardUseVariable**(*reaction*, ***kwargs*)
Bases: *ReactionVariable*, BinaryVariable

Class to represent a backward use variable, a type of binary variable used to enforce backward directionality in reaction net fluxes

prefix = BU_

class pytfa.thermo.**LogConcentration**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a log concentration of a enzyme

prefix = LC_

class pytfa.thermo.**ReactionVariable**(*reaction*, ***kwargs*)
Bases: GenericVariable

Class to represent a variable attached to a reaction

prefix = RV_

property reaction(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.thermo.**MetaboliteVariable**(*metabolite*, ***kwargs*)
Bases: GenericVariable

Class to represent a variable attached to a enzyme

prefix = MV_

property metabolite(*self*)

property id(*self*)
for cobra.thermo.DictList compatibility :return:

property model(*self*)

class pytfa.thermo.**ThermoPotential**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a DeltaG

prefix = P_

class pytfa.thermo.**DeltaGFormstd**(*metabolite*, ***kwargs*)
Bases: *MetaboliteVariable*

Class to represent a DeltaGstf of formation

prefix = DGoF_

pytfa.thermo.**get_bistream_logger**(*name*)
Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages in the log file

Parameters **name** – a class __name__ attribute

Returns

pytfa.thermo.**BIGM**

pytfa.thermo.**BIGM_THERMO**

pytfa.thermo.**BIGM_DG**

pytfa.thermo.**BIGM_P**

pytfa.thermo.**EPSILON**

pytfa.thermo.**MAX_STOICH = 10**

class pytfa.thermo.**ThermoModel**(*thermo_data=None*, *model=Model()*, *name=None*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*)
Bases: pytfa.core.model.LCSBModel, cobra.Model

A class representing a cobra_model with thermodynamics information

_init_thermo(*self*)

normalize_reactions(*self*)
Find reactions with important stoichiometry and normalizes them :return:

_prepare_metabolite(*self*, *met*)

Parameters **met** –

Returns

_prepare_reaction(*self*, *reaction*)

prepare(*self*)
Prepares a COBRA toolbox cobra_model for TFBA analysis by doing the following:

1. checks if a reaction is a transport reaction
2. checks the ReactionDB for Gibbs energies of formation of metabolites
3. computes the Gibbs energies of reactions

_convert_metabolite(*self*, *met*, *add_potentials*, *verbose*)
Given a enzyme, proceeds to create the necessary variables and constraints for thermodynamics-based modeling

Parameters **met** –

Returns

_convert_reaction(*self*, *rxn*, *add_potentials*, *add_displacement*, *verbose*)

Parameters

- **rxn** –
- **add_potentials** –
- **add_displacement** –
- **verbose** –

Returns

convert(*self*, *add_potentials=False*, *add_displacement=False*, *verbose=True*)
Converts a cobra_model into a tFBA ready cobra_model by adding the thermodynamic constraints required

> **Warning:** This function requires you to have already called `prepare()`, otherwise it will raise an Exception !

print_info(*self*, *specific=False*)
Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memo*)

Parameters **memo** –

Returns

copy(*self*)
Needs to be reimplemented, as our objects have complicated hierarchy :return:

class pytfa.thermo.**MetaboliteThermo**(*metData*, *pH*, *ionicStr*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*, *debye_huckel_b=std.DEBYE_HUCKEL_B_0*, *thermo_unit='kJ/mol'*, *debug=False*)
A class representing the thermodynamic values of a enzyme

Parameters

- **metData** (*dict*) – A dictionary containing the values for the enzyme, from the thermodynamic database
- **pH** (*float*) – The pH of the enzyme's compartment
- **ionicStr** (*float*) – The ionic strength of the enzyme's compartment
- **temperature** –
- **min_ph** –
- **max_ph** –
- **debye_huckel_b** –
- **thermo_unit** (*string*) – The unit used in *metData*'s values
- **debug** (*bool*) – *Optional* If set to `True`, some debugging values will be printed. This is only useful for development or debugging purposes.

Note: The values are automatically computed on class creation. Usually you don't have to call any methods defined by this class, but only to access the attributes you need.

The available attributes are :

Since the reactions expose similar values through a dictionnary, it is better to access the attributes aforementionned of this class as if it was a dictionnary : `enzyme.thermo['pH']`.

__getitem__(*self*, *key*)

__repr__(*self*)
Return repr(self).

keys(*self*)

values(*self*)

items(*self*)

__cmp__(*self*, *dict_*)

__contains__(*self*, *item*)

__iter__(*self*)

__unicode__(*self*)

calcDGis(*self*)
Calculate the transformed Gibbs energy of formation of specie with given pH and ionic strength using formula given by Goldberg and Tewari, 1991

Equation 4.5-6 in Alberty's book

Returns DG_is for the enzyme

Return type float

calcDGsp(*self*)
Calculate the transformed Gibbs energy of formation of specie with given pH and ionic strength using formula given by Goldberg and Tewari, 1991

Equation 4.4-10 in Alberty's book

Returns DG_sp for the enzyme

Return type float

calc_potential(*self*)
Calculate the binding polynomial of a specie, with pK values

Returns The potential of the enzyme

Return type float

get_pka(*self*)
Get the pKas of the enzyme

Returns The pKas of the enzyme

Return type list(float)

_calc_pka(*self*, *pka*, *sigmanusq*)

calcDGspA(*self*)

Calculates deltaGf, charge and nH of the specie when it is at least protonated state based on MFAToolkit compound data for the pKa values within the range considered (MIN_pH to MAX_pH).

These values are used as the starting point for Alberty's calculations.

Returns deltaGspA, sp_charge and sp_nH

Return type tuple(float, float, int)

pytfa.utils

Submodules

pytfa.utils.logger

Logging utilities

Module Contents

Functions

get_bistream_logger(name)	Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages
get_timestr()	

Attributes

LOGFOLDERNAME

pytfa.**LOGFOLDERNAME = logs**

pytfa.**get_bistream_logger**(*name*)

Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages in the log file

Parameters **name** – a class __name__ attribute

Returns

pytfa.**get_timestr**()

pytfa.utils.numerics

BIG M and epsilon constants definitions

Module Contents

pytfa.utils.numerics.**BIGM = 1000**

pytfa.utils.numerics.**BIGM_THERMO = 1000.0**

pytfa.utils.numerics.**BIGM_DG = 1000.0**

pytfa.utils.numerics.**BIGM_P = 1000.0**

pytfa.utils.numerics.**EPSILON = 1e-06**

pytfa.utils.str

Some tools used by pyTFA

Module Contents

Functions

camel2underscores(name)	
varnames2ids(tmodel, variables)	

pytfa.**camel2underscores**(*name*)

pytfa.**varnames2ids**(*tmodel*, *variables*)

6.1.2 Package Contents

Classes

ThermoModel	A class representing a cobra_model with thermodynamics information

class pytfa.**ThermoModel**(*thermo_data=None*, *model=Model()*, *name=None*, *temperature=std.TEMPERATURE_0*, *min_ph=std.MIN_PH*, *max_ph=std.MAX_PH*)

Bases: pytfa.core.model.LCSBModel, cobra.Model

A class representing a cobra_model with thermodynamics information

_init_thermo(*self*)

normalize_reactions(*self*)

Find reactions with important stoichiometry and normalizes them :return:

_prepare_metabolite(*self*, *met*)

Parameters met –

Returns

_prepare_reaction(*self*, *reaction*)

prepare(*self*)
Prepares a COBRA toolbox cobra_model for TFBA analysis by doing the following:

1. checks if a reaction is a transport reaction
2. checks the ReactionDB for Gibbs energies of formation of metabolites
3. computes the Gibbs energies of reactions

_convert_metabolite(*self*, *met*, *add_potentials*, *verbose*)
Given a enzyme, proceeds to create the necessary variables and constraints for thermodynamics-based modeling

Parameters met –

Returns

_convert_reaction(*self*, *rxn*, *add_potentials*, *add_displacement*, *verbose*)

Parameters

- rxn –
- add_potentials –
- add_displacement –
- verbose –

Returns

convert(*self*, *add_potentials=False*, *add_displacement=False*, *verbose=True*)
Converts a cobra_model into a tFBA ready cobra_model by adding the thermodynamic constraints required

> **Warning:** This function requires you to have already called `prepare()`, otherwise it will raise an Exception !

print_info(*self*, *specific=False*)
Print information and counts for the cobra_model :return:

__deepcopy__(*self*, *memo*)

Parameters memo –

Returns

copy(*self*)
Needs to be reimplemented, as our objects have complicated hierarchy :return:

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

r

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W