

---

# pytfa Documentation

*Release 0.9.2*

**pytfa Team**

**Jun 27, 2022**



## CONTENTS:

|          |                                                      |            |
|----------|------------------------------------------------------|------------|
| <b>1</b> | <b>Quick start</b>                                   | <b>3</b>   |
| <b>2</b> | <b>pyTFA models</b>                                  | <b>5</b>   |
| 2.1      | Compartment data . . . . .                           | 5          |
| 2.2      | Metabolites . . . . .                                | 6          |
| 2.3      | Reactions . . . . .                                  | 6          |
| <b>3</b> | <b>Thermodynamic Databases</b>                       | <b>9</b>   |
| 3.1      | Converting a Matlab thermodynamic database . . . . . | 9          |
| 3.2      | Loading a thermodynamic database . . . . .           | 9          |
| 3.3      | Structure of a thermodynamic database . . . . .      | 9          |
| <b>4</b> | <b>Solver setup</b>                                  | <b>13</b>  |
| 4.1      | GPLK . . . . .                                       | 13         |
| 4.2      | CPLEX . . . . .                                      | 13         |
| 4.3      | Gurobi . . . . .                                     | 13         |
| <b>5</b> | <b>Integrating metabolomics</b>                      | <b>15</b>  |
| <b>6</b> | <b>API Reference</b>                                 | <b>17</b>  |
| 6.1      | pytfa . . . . .                                      | 17         |
| <b>7</b> | <b>Indices and tables</b>                            | <b>113</b> |
|          | <b>Python Module Index</b>                           | <b>115</b> |
|          | <b>Index</b>                                         | <b>117</b> |



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<sup>1</sup> !

---

<sup>1</sup> 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.



## 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<sup>1</sup>, a simple use case for TFA on a reduced *Escherichia coli*. We show that adding thermodynamics 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

---

<sup>1</sup> 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.



## 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 dictionary representing the membrane potential between this compartment (which is the source compartment) and the others.<br>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 dictionary)                                                                                                                                                                                                                           |

Here is an example:

```
cobra_model.compartments['c'] = {
    'c_max': 0.01,
    'c_min': 9.999999999999999e-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` 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   | <code>bool</code> Whether the thermodynamic values were computed or not.                                                                                                                         |
| deltaGR    | <code>float</code> Sum of the non-concentration terms for the reaction. Used as the right-hand side value of a constraint.<br>If the thermodynamic values were not computed, this is $10^{*7}$ . |
| deltaGRerr | <code>float</code> Error on deltaGR<br>If the thermodynamic values were not computed, this is $10^{*7}$ .                                                                                        |
| deltaGrxn  | <code>float</code> Sum of the deltaGF of all the metabolites in the reaction. <b>Not defined if computed is False !</b>                                                                          |
| isTrans    | <code>bool</code> 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,  
}
```



## 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 dictionary containing the metabolites' thermodynamic data. See *Metabolites* for more information.
- `cues` : *dict* A dictionary containing the cues' thermodynamic data. See *Cues* for more information.

### 3.3.1 Metabolites

This is a dictionary 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           | <code>string</code> SeedID of the metabolite                                                                                                                                      |
| charge_std   | <code>float</code> Charge of the metabolite (mV) in standard conditions                                                                                                           |
| delt-aGf_std | <code>float</code> Transformed Gibbs energy of formation of the metabolite, in standard conditions.                                                                               |
| delt-aGf_err | <code>float</code> Error on the transformed Gibbs energy of formation of the metabolite, in standard conditions                                                                   |
| mass_std     | <code>float</code> Mass of the metabolite (g.mol <sup>-1</sup> )                                                                                                                  |
| nH_std       | <code>int</code> Number of protons of the metabolite, in standard conditions                                                                                                      |
| error        | <code>string</code> Error on the metabolite's thermodynamic data. Thermodynamic values will be computed only if this equals to 'Nil'.                                             |
| formula      | <code>string</code> Formula of the metabolite.                                                                                                                                    |
| nH_std       | <code>int</code> Number of protons in the metabolite's formula                                                                                                                    |
| name         | <code>string</code> Name of the metabolite                                                                                                                                        |
| other_names  | <code>list (string)</code> Other names of the metabolite                                                                                                                          |
| pKa          | <code>list (float)</code> pKas of the metabolite                                                                                                                                  |
| struct_cues  | <code>dict (int)</code> cues of the metabolite<br>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 dictionary storing various thermodynamic data about cues. It is stored as a `dict` where each key is the cue ID, as referenced 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. <i>Optional</i>                            |
| 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
}
```



## 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):



### 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](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*



## 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:

$$5 \cdot 10^{-3} \text{ mol.L}^{-1} \leq [X] \leq 3 \cdot 10^{-2} \text{ mol.L}^{-1}$$

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.



## API REFERENCE

This page contains auto-generated API reference documentation<sup>1</sup>.

### 6.1 `pytfa`

Thermodynamic analysis for Flux-Based Analysis

#### 6.1.1 Subpackages

`pytfa.analysis`

##### Submodules

`pytfa.analysis.chebyshev`

Variability analysis

##### Module Contents

##### Classes

---

|                              |                                          |
|------------------------------|------------------------------------------|
| <code>ChebyshevRadius</code> | Variable representing a Chebyshev radius |
|------------------------------|------------------------------------------|

---

---

<sup>1</sup> Created with `sphinx-autoapi`

## Functions

---

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

---

## Attributes

---

|                   |  |
|-------------------|--|
| <code>BIGM</code> |  |
|-------------------|--|

---

`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

|                                         |                                               |     |                                                                         |
|-----------------------------------------|-----------------------------------------------|-----|-------------------------------------------------------------------------|
| <code>apply_reaction_variability</code> | <code>(tmodel, va, inplace=True)</code>       | in- | Applies the VA results as bounds for the reactions of a cobra_model     |
| <code>apply_generic_variability</code>  | <code>(tmodel, va, inplace=True)</code>       | in- | Reactions a dealt with cobra, but the other variables added use pytfa's |
| <code>apply_directionality</code>       | <code>(tmodel, solution, inplace=True)</code> | in- | 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

|                                      |                                                   |
|--------------------------------------|---------------------------------------------------|
| <code>GeneralizedHRSampler</code>    | The abstract base class for hit-and-run samplers. |
| <code>GeneralizedACHRSampler</code>  | The abstract base class for hit-and-run samplers. |
| <code>GeneralizedOptGPSampler</code> | The abstract base class for hit-and-run samplers. |

#### Functions

|                                                                                     |                                                            |
|-------------------------------------------------------------------------------------|------------------------------------------------------------|
| <code>sample(model, n, method='optgp', thinning=100, processes=1, seed=None)</code> | 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<sup>1</sup>.** 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<sup>2</sup>.

**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).

## pytfa.analysis.variability

### Variability analysis

<sup>1</sup> 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.

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

## Module Contents

### Functions

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

### Attributes

|                                |
|--------------------------------|
| <code>CPU_COUNT</code>         |
| <code>BEST_THREAD_RATIO</code> |

`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')`

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, given 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

|                                                |                                                                              |
|------------------------------------------------|------------------------------------------------------------------------------|
| <a href="#"><i>GeneralizedHRSampler</i></a>    | The abstract base class for hit-and-run samplers.                            |
| <a href="#"><i>GeneralizedACHRSampler</i></a>  | The abstract base class for hit-and-run samplers.                            |
| <a href="#"><i>GeneralizedOptGPSampler</i></a> | The abstract base class for hit-and-run samplers.                            |
| <a href="#"><i>DeltaG</i></a>                  | Class to represent a DeltaG                                                  |
| <a href="#"><i>ForbiddenProfile</i></a>        | Class to represent a forbidden net flux directionality profile               |
| <a href="#"><i>ForwardUseVariable</i></a>      | Class to represent a forward use variable, a type of binary variable used to |

## Functions

|                                                                                                           |                                                                             |
|-----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| <code>sample(model, n, method='optgp', thinning=100, processes=1, seed=None)</code>                       | Sample valid flux distributions from a thermo cobra_model.                  |
| <code>get_direction_use_variables(tmodel, solution)</code>                                                | Returns the active use variables in a solution. Use Variables are binary    |
| <code>get_bistream_logger(name)</code>                                                                    | Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages  |
| <code>find_bidirectional_reactions(va, tolerance=1e-08)</code>                                            | Returns the ids of reactions that can both carry net flux in the forward or |
| <code>find_directionality_profiles(tmodel, bidirectional, max_iter=10000.0, solver='optlang-glpk')</code> | Takes a ThermoModel and performs enumeration of the directionality profiles |
| <code>_bool2str(bool_list)</code>                                                                         | turns a list of booleans into a string                                      |
| <code>_variability_analysis_element(tmodel, var, sense)</code>                                            |                                                                             |
| <code>variability_analysis(tmodel, kind='reactions', proc_num=BEST_THREAD_RATIO)</code>                   | Performs variability analysis, given a variable type                        |
| <code>parallel_variability_analysis(tmodel, kind='reactions', proc_num=BEST_THREAD_RATIO)</code>          | WIP.                                                                        |
| <code>calculate_dissipation(tmodel, solution=None)</code>                                                 |                                                                             |
| <code>apply_reaction_variability(tmodel, va, inplace=True)</code>                                         | Applies the VA results as bounds for the reactions of a cobra_model         |
| <code>apply_generic_variability(tmodel, va, inplace=True)</code>                                          | Reactions a dealt with cobra, but the other variables added use pytfa's     |
| <code>apply_directionality(tmodel, solution, inplace=True)</code>                                         | 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**<sup>1</sup>. 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<sup>2</sup>.

**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.

<sup>1</sup> 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.

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

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:  $FU_{rxn\_1} + BU_{rxn\_2} + \dots + FU_{rxn\_n} \leq n-1$

**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

**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')`

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, given 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`

`pytfa.io`

**Submodules**

`pytfa.io.base`

Input/Output tools to import or export pytfa models

**Module Contents****Functions**

|                                                                    |                                                                                 |
|--------------------------------------------------------------------|---------------------------------------------------------------------------------|
| <code>import_matlab_model(path, variable_name=None)</code>         | Convert at matlab cobra_model to a pyTFA cobra_model, with Thermodynamic values |
| <code>recover_compartments(model, compartments_list)</code>        |                                                                                 |
| <code>write_matlab_model(tmodel, path, var-name='tmodel')</code>   | Writes the Thermo Model to a Matlab-compatible structure                        |
| <code>create_thermo_dict(tmodel)</code>                            | Dumps the thermodynamic information in a mat-compatible dictionary              |
| <code>varnames2matlab(name, tmodel)</code>                         | Transforms reaction variable pairs from ('ACALD', 'ACALD_reverse_xxxx') to      |
| <code>create_problem_dict(tmodel)</code>                           | Dumps the the MILP formulation for TFA in a mat-compatible dictionary           |
| <code>create_generalized_matrix(tmodel, array_type='dense')</code> | Returns the generalized stoichiometric matrix used for TFA                      |
| <code>load_thermoDB(path)</code>                                   | Load a thermodynamic database                                                   |
| <code>printLP(model)</code>                                        | Print the LP file corresponding to the cobra_model                              |
| <code>writeLP(model, path=None)</code>                             | 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.vardnames2matlab(name, tmodel)`

Transforms reaction variable pairs from ('ACALD', 'ACALD\_reverse\_xxxx') 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 stoichiometric 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.dict`

Make the model serializable

## Module Contents

## Functions

|                                                      |                                                                |
|------------------------------------------------------|----------------------------------------------------------------|
| get_all_subclasses(cls)                              |                                                                |
| make_subclasses_dict(cls)                            |                                                                |
| get_model_variable_subclasses()                      |                                                                |
| get_model_constraint_subclasses()                    |                                                                |
| metabolite_thermo_to_dict(metthermo)                 |                                                                |
| var_to_dict(variable)                                |                                                                |
| cons_to_dict(constraint)                             |                                                                |
| archive_variables(var_dict)                          |                                                                |
| archive_constraints(cons_dict)                       |                                                                |
| archive_compositions(compositions)                   | Turns a peptide compositions dict of the form:                 |
| _stoichiometry_to_dict(stoichiometric_dict)          | Turns a stoichiometric compositions dict of the form:          |
| get_solver_string(model)                             |                                                                |
| obj_to_dict(model)                                   |                                                                |
| model_to_dict(model)                                 |                                                                |
| <b>param model</b>                                   |                                                                |
| _add_thermo_reaction_info(rxn, rxn_dict)             |                                                                |
| _add_thermo_metabolite_info(met, met_dict)           |                                                                |
| model_from_dict(obj, solver=None, custom_hooks=None) | Custom_hooks looks like                                        |
| rebuild_obj_from_dict(new, objective_dict)           |                                                                |
| add_custom_classes(model, custom_hooks)              | Allows custom variable serialization                           |
| get_hook_dict(model, custom_hooks)                   |                                                                |
| init_thermo_model_from_dict(new, obj)                |                                                                |
| rebuild_compositions(new, compositions_dict)         | Performs the reverse operation of :func:archive_compositions   |
| _rebuild_stoichiometry(new, stoich)                  | Performs the reverse operation of :func:_stoichiometry_to_dict |

## Attributes

---

`BASE_NAME2HOOK`

---

`SOLVER_DICT`

---

`pytfa.get_all_subclasses(cls)``pytfa.make_subclasses_dict(cls)``pytfa.get_model_variable_subclasses()``pytfa.get_model_constraint_subclasses()``pytfa.BASE_NAME2HOOK``pytfa.SOLVER_DICT``pytfa.metabolite_thermo_to_dict(metthermo)``pytfa.var_to_dict(variable)``pytfa.cons_to_dict(constraint)``pytfa.archive_variables(var_dict)``pytfa.archive_constraints(cons_dict)``pytfa.archive_compositions(compositions)`

Turns a peptide compositions dict of the form: { 'b3991': defaultdict(int,

**<Metabolite ala\_\_L\_c at 0x7f7d25504f28>: -42,**  
    <Metabolite arg\_\_L\_c at 0x7f7d2550bcf8>: -11, <Metabolite asn\_\_L\_c at 0x7f7d2550beb8>:  
    -6, ... }},

... }

to:

**{ 'b3991': defaultdict(int,**  
    **{ 'ala\_\_L\_c': -42,**  
        'arg\_\_L\_c': -11, 'asn\_\_L\_c': -6,  
        ... }},

... } :param compositions: :return:

`pytfa._stoichiometry_to_dict(stoichiometric_dict)`

Turns a stoichiometric compositions dict of the form: 'b3991': defaultdict(int,

**<Metabolite ala\_\_L\_c at 0x7f7d25504f28>: -42,**  
    <Metabolite arg\_\_L\_c at 0x7f7d2550bcf8>: -11, <Metabolite asn\_\_L\_c at 0x7f7d2550beb8>:  
    -6, ... })

to:

**'b3991': defaultdict(int,**

```
{'ala__L_c': -42,
  'arg__L_c': -11, 'asn__L_c': -6,
  ...})
```

```
pytfa.get_solver_string(model)
```

```
pytfa.obj_to_dict(model)
```

```
pytfa.model_to_dict(model)
```

#### Parameters

**model** –

#### Returns

```
pytfa._add_thermo_reaction_info(rxn, rxn_dict)
```

```
pytfa._add_thermo_metabolite_info(met, met_dict)
```

```
pytfa.model_from_dict(obj, solver=None, custom_hooks=None)
```

Custom\_hooks looks like

```
custom_hooks = {<EnzymeVariable Class at 0xfffff> : 'enzymes',
                 ... }
```

#### Parameters

- **obj** –
- **solver** –
- **custom\_hooks** –

#### Returns

```
pytfa.rebuild_obj_from_dict(new, objective_dict)
```

```
pytfa.add_custom_classes(model, custom_hooks)
```

Allows custom variable serialization

#### Parameters

- **model** –
- **base\_classes** –
- **base\_hooks** –
- **custom\_hooks** –

#### Returns

```
pytfa.get_hook_dict(model, custom_hooks)
```

```
pytfa.init_thermo_model_from_dict(new, obj)
```

```
pytfa.rebuild_compositions(new, compositions_dict)
```

Performs the reverse operation of :func:archive\_compositions

#### Parameters

- **new** –

- **compositions\_dict** –

**Returns**

`pytfa._rebuild_stoichiometry(new, stoich)`

Performs the reverse operation of `:func:_stoichiometry_to_dict`

**Parameters**

- **new** –
- **stoich** –

**Returns**

## pytfa.io.enrichment

Tools to import or export enrichment to and from pytfa models

## Module Contents

### Functions

|                                                               |                                                        |
|---------------------------------------------------------------|--------------------------------------------------------|
| <code>write_lexicon(tmodel, filepath)</code>                  | Writes a csv file in the format :                      |
| <code>annotate_from_lexicon(model, lexicon)</code>            | Converts a lexicon into annotation for the metabolites |
| <code>read_lexicon(filepath)</code>                           |                                                        |
| <code>write_compartment_data(tmodel, filepath)</code>         | <b>param filepath</b>                                  |
| <code>read_compartment_data(filepath)</code>                  |                                                        |
| <code>apply_compartment_data(tmodel, compartment_data)</code> |                                                        |

`pytfa.write_lexicon(tmodel, filepath)`

**Writes a csv file in the format :**

seed\_id

13BDglen\_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

|                        |                                                                           |
|------------------------|---------------------------------------------------------------------------|
| <code>MyEncoder</code> | We define an encoder that takes care of the serialization of numpy types, |
|------------------------|---------------------------------------------------------------------------|

### Functions

|                                               |                                       |
|-----------------------------------------------|---------------------------------------|
| <code>check_json_extension(filepath)</code>   |                                       |
| <code>save_json_model(model, filepath)</code> |                                       |
| <code>load_json_model(filepath)</code>        |                                       |
| <code>json_dumps_model(model)</code>          | Returns a JSON dump as a string       |
| <code>json_loads_model(s)</code>              | 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)`

---

|                                                                        |                                                                       |
|------------------------------------------------------------------------|-----------------------------------------------------------------------|
| <code>plot_thermo_displacement_histogram(tmodel, solution=None)</code> | Plot a histogram of the thermodynamic displacement. if no solution is |
|------------------------------------------------------------------------|-----------------------------------------------------------------------|

---

|                                               |                                                          |
|-----------------------------------------------|----------------------------------------------------------|
| <code>plot_histogram(values, **kwargs)</code> | 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 visualize results

## Module Contents

### Functions

|                                                                                |       |                                                                          |
|--------------------------------------------------------------------------------|-------|--------------------------------------------------------------------------|
| <code>export_variable_for_escher(tmodel, variable_type, data, filename)</code> | vari- | Exports all the variables of a given type into a csv file, indexed by    |
| <code>get_reaction_data(tmodel, data)</code>                                   |       | Exports values indexed by reaction ids. Reconciles Forward and Backwards |
| <code>export_reactions_for_escher(tmodel, data, filename)</code>               | file- | 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 readable 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

|                                                                       |                                                                                 |
|-----------------------------------------------------------------------|---------------------------------------------------------------------------------|
| <code>import_matlab_model(path, variable_name=None)</code>            | Convert at matlab cobra_model to a pyTFA cobra_model, with Thermodynamic values |
| <code>recover_compartments(model, compartments_list)</code>           |                                                                                 |
| <code>write_matlab_model(tmodel, path, variable_name='tmodel')</code> | Writes the Thermo Model to a Matlab-compatible structure                        |
| <code>create_thermo_dict(tmodel)</code>                               | Dumps the thermodynamic information in a mat-compatible dictionary              |
| <code>varnames2matlab(name, tmodel)</code>                            | Transforms reaction variable pairs from ('ACALD', 'ACALD_reverse_xxxx') to      |
| <code>create_problem_dict(tmodel)</code>                              | Dumps the the MILP formulation for TFA in a mat-compatible dictionary           |
| <code>create_generalized_matrix(tmodel, array_type='dense')</code>    | Returns the generalized stoichiometric matrix used for TFA                      |
| <code>load_thermoDB(path)</code>                                      | Load a thermodynamic database                                                   |
| <code>printLP(model)</code>                                           | Print the LP file corresponding to the cobra_model                              |
| <code>writeLP(model, path=None)</code>                                | Write the LP file of the specified cobra_model to the file indicated by path.   |
| <code>write_lexicon(tmodel, filepath)</code>                          | Writes a csv file in the format :                                               |
| <code>annotate_from_lexicon(model, lexicon)</code>                    | Converts a lexicon into annotation for the metabolites                          |
| <code>read_lexicon(filepath)</code>                                   |                                                                                 |
| <code>write_compartment_data(tmodel, filepath)</code>                 | <b>param filepath</b>                                                           |
| <code>read_compartment_data(filepath)</code>                          |                                                                                 |
| <code>apply_compartment_data(tmodel, compartment_data)</code>         |                                                                                 |

## 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\_xxxx') 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 stoichiometric 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

13BDglen\_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.                               |
| 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                         |

```
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 the 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, concatenates 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.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:  $SU_{rxn}: FU_{rxn} + BU_{rxn} \leq 1$

**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:  $FU_{rxn_1} + BU_{rxn_2} + \dots + FU_{rxn_n} \leq n-1$

**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)

## pytfa.optim.debugging

Debugging of models

## Module Contents

### Functions

---

|                                                            |                                                                    |
|------------------------------------------------------------|--------------------------------------------------------------------|
| <code>debug_iis(model)</code>                              | Performs reduction to an Irreducible Inconsistent Sub-system (IIS) |
| <code>find_extreme_coeffs(model, n=5)</code>               |                                                                    |
| <code>find_maxed_vars(model, ub=1000, epsilon=0.01)</code> |                                                                    |

---

`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

|                                |                                                      |
|--------------------------------|------------------------------------------------------|
| <code>RequirePrefixMeta</code> | Metaclass that enforces child classes define prefix. |
|--------------------------------|------------------------------------------------------|

### Attributes

|                                   |
|-----------------------------------|
| <code>ABCRequirePrefixMeta</code> |
|-----------------------------------|

**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

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

---

### Attributes

|                              |
|------------------------------|
| <code>ConstraintTuple</code> |
| <code>OPTLANG_BINARY</code>  |

---

`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  $z = b*f(y) \Leftrightarrow z - b*f(y) = 0 \Leftrightarrow \{ L*b \leq z \leq U*b \mid f(y) - U*(1-b) \leq z \leq f(y) - L*(1-b) \}$

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  $z = b*x \Leftrightarrow z - b*x = 0 \Leftrightarrow \{ x + M*b - M \leq z \leq M*b \mid z \leq x \}$

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)`


---



---

|                                           |                                      |                       |
|-------------------------------------------|--------------------------------------|-----------------------|
| <code>relax_dgo(tmodel,</code>            | <code>reactions_to_ignore=(),</code> |                       |
| <code>solver=None, in_place=False)</code> |                                      | <b>param t_tmodel</b> |

---



---

|                               |                                        |                                    |
|-------------------------------|----------------------------------------|------------------------------------|
| <code>relax_lc(tmodel,</code> | <code>metabolites_to_ignore=(),</code> |                                    |
| <code>solver=None)</code>     |                                        | <b>param metabolites_to_ignore</b> |

---

**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\_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

|                                                                                              |                                                                                   |
|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <code>get_all_subclasses(cls)</code>                                                         | Given a variable or constraint class, get all the subclasses                      |
| <code>chunk_sum(variables)</code>                                                            | This functions handles the sum of many sympy variables by chunks, which           |
| <code>symbol_sum(variables)</code>                                                           | ```python                                                                         |
| <code>get_solution_value_for_variables(solution, these_vars, index_by_reaction=False)</code> |                                                                                   |
| <code>compare_solutions(models)</code>                                                       | returns the solution dictionary for each cobra_model                              |
| <code>evaluate_constraint_at_solution(constraint, solution)</code>                           | <b>param expression</b>                                                           |
| <code>get_active_use_variables(tmodel, solution)</code>                                      | Returns the active use variables in a solution. Use Variables are binary          |
| <code>get_direction_use_variables(tmodel, solution)</code>                                   | Returns the active use variables in a solution. Use Variables are binary          |
| <code>get_primal(tmodel, vartype, index_by_reactions=False)</code>                           | Returns the primal value of the cobra_model for variables of a given type         |
| <code>strip_from_integer_variables(tmodel)</code>                                            | Removes all integer and binary variables of a cobra_model, to make it sample-able |
| <code>copy_solver_configuration(source, target)</code>                                       | 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 subclasses 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 dictionary 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                                 |
| 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                                                  |
| DeltaGstd                  | Class to represent a DeltaG <sup>o</sup> (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         |
| 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 the 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, concatenates 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  $X_{\text{hat}} = X/a$ . This returns  $X = a \cdot X_{\text{hat}}$  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*)

Subtracting 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.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 an 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.**DeltaGstd**(*reaction*, *\*\*kwargs*)

Bases: [ReactionVariable](#)

Class to represent a DeltaG<sup>o</sup> (naught) - standard conditions

**prefix** = **DGo\_**

**class** pytfa.**ThermoDisplacement**(*reaction*, *\*\*kwargs*)

Bases: [ReactionVariable](#)

Class to represent the thermodynamic displacement of a reaction  $\Gamma = -\Delta G/RT$

**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.**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

|                                  |                                                                              |
|----------------------------------|------------------------------------------------------------------------------|
| <i>GenericConstraint</i>         | Class to represent a generic constraint. The purpose is that the interface   |
| <i>ModelConstraint</i>           | Class to represent a variable attached to the model                          |
| <i>GeneConstraint</i>            | Class to represent a variable attached to a enzyme                           |
| <i>ReactionConstraint</i>        | Class to represent a variable attached to a reaction                         |
| <i>MetaboliteConstraint</i>      | Class to represent a variable attached to a enzyme                           |
| <i>NegativeDeltaG</i>            | Class to represent thermodynamics constraints.                               |
| <i>ForwardDeltaGCoupling</i>     | Class to represent thermodynamics coupling: DeltaG of reactions has to be    |
| <i>BackwardDeltaGCoupling</i>    | Class to represent thermodynamics coupling: DeltaG of reactions has to be    |
| <i>ForwardDirectionCoupling</i>  | Class to represent a forward directionality coupling with thermodynamics on  |
| <i>BackwardDirectionCoupling</i> | Class to represent a backward directionality coupling with thermodynamics on |
| <i>SimultaneousUse</i>           | Class to represent a simultaneous use constraint on reaction variables       |
| <i>DisplacementCoupling</i>      | Class to represent the coupling to the thermodynamic displacement            |
| <i>ForbiddenProfile</i>          | Class to represent a forbidden net flux directionality profile               |
| <i>LinearizationConstraint</i>   | Class to represent a variable attached to a reaction                         |
| <i>GenericVariable</i>           | Class to represent a generic variable. The purpose is that the interface     |
| <i>ModelVariable</i>             | Class to represent a variable attached to the model                          |
| <i>GeneVariable</i>              | Class to represent a gene variable                                           |
| <i>BinaryVariable</i>            | Class to represent a generic binary variable                                 |
| <i>ReactionVariable</i>          | Class to represent a variable attached to a reaction                         |
| <i>MetaboliteVariable</i>        | Class to represent a variable attached to a enzyme                           |
| <i>ForwardUseVariable</i>        | Class to represent a forward use variable, a type of binary variable used to |

continues on next page

Table 1 – continued from previous page

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

## Functions

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

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 the 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, concatenates 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  $X_{\text{hat}} = X/a$ . This returns  $X = a.X_{\text{hat}}$  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*)

Subtracting 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.**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.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  $\Gamma = -\Delta G/RT$ 
    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.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  $\Delta G^{\circ}$  (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\_

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_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 the 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, concatenates 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 the 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, concatenates 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  $X_{\text{hat}} = X/a$ . This returns  $X = a.X_{\text{hat}}$  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)**

Subtracting 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 subclasses 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 dictionary 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

|                    |                                                      |
|--------------------|------------------------------------------------------|
| <i>FluxKO</i>      | Class to represent a variable attached to a reaction |
| <i>UseOrKOInt</i>  | Class to represent a variable attached to a reaction |
| <i>UseOrKOFlux</i> | Class to represent a variable attached to a reaction |
| <i>LumpGEM</i>     | A class encapsulating the LumpGEM algorithm          |

## Functions

---

*sum\_reactions*(rxn\_dict, id\_='summed\_reaction', Keys are reactions  
epsilon=1e-09)

---

## 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 :  
 $\text{rxn.forward\_variable} + \text{rxn.reverse\_variable} + \text{activation\_var} * C_{\text{uptake}} < C_{\text{uptake}}$

**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*)

#### 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)`

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*)

Extracts the reduced gem.

**Returns**

None

**run**(*self*)

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)`

---

`trim_epsilon_mets(met_dict, epsilon)`

---

`set_medium(model, medium_dict, inplace)`

---

`pytfa.redgem.utils.remove_blocked_reactions(model)`

`pytfa.redgem.utils.trim_epsilon_mets(met_dict, epsilon)`

`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

---

|                                                                     |                                                        |
|---------------------------------------------------------------------|--------------------------------------------------------|
| <code>build_thermo_from_equilibrator(model, T=TEMPERATURE_0)</code> | Build <i>thermo_data</i> structure from a cobra Model. |
|---------------------------------------------------------------------|--------------------------------------------------------|

---

|                                        |                                            |
|----------------------------------------|--------------------------------------------|
| <code>compute_dGf(compound, cc)</code> | Get Gf from equilibrator <i>compound</i> . |
|----------------------------------------|--------------------------------------------|

---

|                                              |                                                                          |
|----------------------------------------------|--------------------------------------------------------------------------|
| <code>compound_to_entry(compound, cc)</code> | Build thermo structure entry from a <i>equilibrator_cache.Compound</i> . |
|----------------------------------------------|--------------------------------------------------------------------------|

---

## 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 dictionary, it is better to access the attributes aforementioned of this class as if it was a dictionary : `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**

|                                                         |                                |                                                                     |
|---------------------------------------------------------|--------------------------------|---------------------------------------------------------------------|
| <code>calcDGtpt_rhs(reaction, thermo_units)</code>      | <code>compartmentsData,</code> | Calculates the RHS of the deltaG constraint, i.e. the sum of the    |
| <code>calcDGR_cues(reaction, reaction_cues_data)</code> |                                | Calculates the deltaG reaction and error of the reaction using the  |
| <code>calcDGF_cues(cues, reaction_cues_data)</code>     |                                | Calculates the deltaG formation and error of the compound using its |
| <code>get_debye_huckel_b(T)</code>                      |                                | 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 = cpd000008 + 4 cpd000067 + cpd000009 <=> cpd000002 + 3 cpd000067 + cpd000001
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 dictionary) and their indices (values of the dictionary), 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 dictionary 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*, *null\_error\_override*=2)

**Parameters**

- **reaction** –
- **null\_error\_override** – overrides DeltaG when it is 0 to allow flexibility. 2kcal/mol is standard in estimation frameworks like GCM.

**Returns**

**prepare**(*self*, *null\_error\_override*=2)

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

**Parameters**

**null\_error\_override** – overrides DeltaG when it is 0 to allow flexibility. 2kcal/mol is standard in estimation frameworks like GCM.

**\_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

|                                                                  |                                                                        |
|------------------------------------------------------------------|------------------------------------------------------------------------|
| <code>check_reaction_balance(reaction, proton=None)</code>       | Check the balance of a reaction, and eventually add protons to balance |
| <code>find_transported_mets(reaction)</code>                     | Get a list of the transported metabolites of the reaction.             |
| <code>check_transport_reaction(reaction)</code>                  | Check if a reaction is a transport reaction                            |
| <code>is_same_stoichiometry(this_reaction, that_reaction)</code> |                                                                        |
| <code>is_exchange(rxn)</code>                                    |                                                                        |
| <code>get_reaction_compartment(reaction)</code>                  | Get the compartment of a reaction to then prepare it for conversion.   |

### Attributes

|                            |
|----------------------------|
| <code>Formula_regex</code> |
|----------------------------|

---

## 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 dictionary of the transported metabolites. The index corresponds to the `seed_id` of the transported enzyme

The value is a dictionary with the following values:

- **coeff (`float`):**  
The stoichiometric 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

|                                  |                                                                              |
|----------------------------------|------------------------------------------------------------------------------|
| <i>MetaboliteThermo</i>          | A class representing the thermodynamic values of a enzyme                    |
| <i>SimultaneousUse</i>           | Class to represent a simultaneous use constraint on reaction variables       |
| <i>NegativeDeltaG</i>            | Class to represent thermodynamics constraints.                               |
| <i>BackwardDeltaGCoupling</i>    | Class to represent thermodynamics coupling: DeltaG of reactions has to be    |
| <i>ForwardDeltaGCoupling</i>     | Class to represent thermodynamics coupling: DeltaG of reactions has to be    |
| <i>BackwardDirectionCoupling</i> | Class to represent a backward directionality coupling with thermodynamics on |
| <i>ForwardDirectionCoupling</i>  | Class to represent a forward directionality coupling with thermodynamics on  |
| <i>ReactionConstraint</i>        | Class to represent a variable attached to a reaction                         |
| <i>MetaboliteConstraint</i>      | Class to represent a variable attached to a enzyme                           |
| <i>DisplacementCoupling</i>      | Class to represent the coupling to the thermodynamic displacement            |
| <i>ThermoDisplacement</i>        | Class to represent the thermodynamic displacement of a reaction              |
| <i>DeltaGstd</i>                 | Class to represent a $\Delta G^o$ (naught) - standard conditions             |
| <i>DeltaG</i>                    | Class to represent a DeltaG                                                  |
| <i>ForwardUseVariable</i>        | Class to represent a forward use variable, a type of binary variable used to |
| <i>BackwardUseVariable</i>       | Class to represent a backward use variable, a type of binary variable used   |
| <i>LogConcentration</i>          | Class to represent a log concentration of a enzyme                           |
| <i>ReactionVariable</i>          | Class to represent a variable attached to a reaction                         |
| <i>MetaboliteVariable</i>        | Class to represent a variable attached to a enzyme                           |
| <i>ThermoModel</i>               | A class representing a cobra_model with thermodynamics information           |
| <i>MetaboliteThermo</i>          | A class representing the thermodynamic values of a enzyme                    |

## Functions

|                                                                       |                                                                            |
|-----------------------------------------------------------------------|----------------------------------------------------------------------------|
| <code>calcDGtpt_rhs</code> (reaction, compartmentsData, thermo_units) | Calculates the RHS of the deltaG constraint, i.e. the sum of the           |
| <code>calcDGR_cues</code> (reaction, reaction_cues_data)              | Calculates the deltaG reaction and error of the reaction using the         |
| <code>get_debye_huckel_b</code> (T)                                   | The Debye-Huckel A and B do depend on the temperature                      |
| <code>check_reaction_balance</code> (reaction, proton=None)           | Check the balance of a reaction, and eventually add protons to balance     |
| <code>check_transport_reaction</code> (reaction)                      | Check if a reaction is a transport reaction                                |
| <code>find_transported_mets</code> (reaction)                         | Get a list of the transported metabolites of the reaction.                 |
| <code>get_reaction_compartment</code> (reaction)                      | Get the compartment of a reaction to then prepare it for conversion.       |
| <code>get_bistream_logger</code> (name)                               | Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages |

## Attributes

|                          |
|--------------------------|
| <code>BIGM</code>        |
| <code>BIGM_THERMO</code> |
| <code>BIGM_DG</code>     |
| <code>BIGM_P</code>      |
| <code>EPSILON</code>     |
| <code>MAX_STOICH</code>  |

```
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 dictionary, it is better to access the attributes aforementioned of this class as if it was a dictionary : `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 = cpd000008 + 4 cpd000067 + cpd000009 <=> cpd000002 + 3 cpd000067 + cpd000001
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 dictionary) and their indices (values of the dictionary), 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 dictionary of the transported metabolites. The index corresponds to the `seed_id` of the transported enzyme

The value is a dictionary with the following values:

- **coeff (`float`):**  
The stoichiometric 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.thermo.get_reaction_compartment(reaction)`

Get the compartment of a reaction to then prepare it for conversion.

**class** `pytfa.thermo.SimultaneousUse(reaction, expr, **kwargs)`

Bases: `ReactionConstraint`

Class to represent a simultaneous use constraint on reaction variables Looks like:  $SU_{rxn}: FU_{rxn} + BU_{rxn} \leq 1$

**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:  $\ln(\text{Gamma}) - (1/RT) * \text{DGR}_{\text{rxn}} = 0$

**prefix** = DC\_

**class** pytfa.thermo.**ThermoDisplacement**(*reaction*, *\*\*kwargs*)

Bases: [ReactionVariable](#)

Class to represent the thermodynamic displacement of a reaction  $\text{Gamma} = -\Delta G/RT$

**prefix** = LnGamma\_

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

Bases: [ReactionVariable](#)

Class to represent a  $\Delta G^o$  (naught) - standard conditions

**prefix** = DGo\_

**class** pytfa.thermo.**DeltaG**(*reaction*, *\*\*kwargs*)

Bases: [ReactionVariable](#)

Class to represent a  $\Delta G$

**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)`

`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*, *null\_error\_override*=2)

**Parameters**

- **reaction** –
- **null\_error\_override** – overrides DeltaG when it is 0 to allow flexibility. 2kcal/mol is standard in estimation frameworks like GCM.

**Returns**

**prepare**(*self*, *null\_error\_override*=2)

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

**Parameters**

**null\_error\_override** – overrides DeltaG when it is 0 to allow flexibility. 2kcal/mol is standard in estimation frameworks like GCM.

**\_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 dictionary, it is better to access the attributes aforementioned of this class as if it was a dictionary : `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

---

|                                        |                                                                            |
|----------------------------------------|----------------------------------------------------------------------------|
| <code>get_bistream_logger(name)</code> | Sets up a logger that outputs INFO+ messages on stdout and DEBUG+ messages |
| <code>get_timestr()</code>             |                                                                            |

---

#### Attributes

---

|                            |
|----------------------------|
| <code>LOGFOLDERNAME</code> |
|----------------------------|

---

`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

---

|                    |                                                                    |
|--------------------|--------------------------------------------------------------------|
| <i>ThermoModel</i> | 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, null\_error\_override=2)

**Parameters**

• **reaction** –

• **null\_error\_override** – overrides DeltaG when it is 0 to allow flexibility. 2kcal/mol is standard in estimation frameworks like GCM.

**Returns**

**prepare**(*self*, *null\_error\_override*=2)

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

**Parameters**

**null\_error\_override** – overrides DeltaG when it is 0 to allow flexibility. 2kcal/mol is standard in estimation frameworks like GCM.

**\_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:

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### p

- `pytfa` (*Unix, Windows*), 111
- `pytfa.analysis`, 17
  - `pytfa.analysis.chebyshev`, 17
  - `pytfa.analysis.manipulation`, 19
  - `pytfa.analysis.sampling`, 20
  - `pytfa.analysis.variability`, 23
- `pytfa.core`, 32
- `pytfa.io`, 32
  - `pytfa.io.base`, 32
  - `pytfa.io.dict`, 34
  - `pytfa.io.enrichment`, 38
  - `pytfa.io.json`, 39
  - `pytfa.io.plotting`, 40
  - `pytfa.io.viz`, 41
- `pytfa.optim`, 46
  - `pytfa.optim.config`, 46
  - `pytfa.optim.constraints`, 47
  - `pytfa.optim.debugging`, 50
  - `pytfa.optim.meta`, 51
  - `pytfa.optim.reformulation`, 51
  - `pytfa.optim.relaxation`, 53
  - `pytfa.optim.utils`, 54
  - `pytfa.optim.variables`, 57
- `pytfa.redgem`, 81
  - `pytfa.redgem.debugging`, 81
  - `pytfa.redgem.lumpgem`, 81
  - `pytfa.redgem.network_expansion`, 84
  - `pytfa.redgem.redgem`, 87
  - `pytfa.redgem.utils`, 88
- `pytfa.thermo`, 88
  - `pytfa.thermo.equilibrator`, 88
  - `pytfa.thermo.metabolite`, 89
  - `pytfa.thermo.reaction`, 92
  - `pytfa.thermo.std`, 93
  - `pytfa.thermo.tmodel`, 94
  - `pytfa.thermo.utils`, 96
- `pytfa.utils`, 110
  - `pytfa.utils.logger`, 110
  - `pytfa.utils.numerics`, 110
  - `pytfa.utils.str`, 111

### r

- `redgem` (*Unix, Windows*), 87



## Symbols

- `__add__()` (*pytfa.GenericVariable* method), 60
- `__add__()` (*pytfa.optim.GenericVariable* method), 70, 78
- `__attrname__` (*pytfa.GenericConstraint* property), 47
- `__attrname__` (*pytfa.GenericVariable* property), 59
- `__attrname__` (*pytfa.optim.GenericConstraint* property), 66, 76
- `__attrname__` (*pytfa.optim.GenericVariable* property), 69, 77
- `__cmp__()` (*pytfa.MetaboliteThermo* method), 90
- `__cmp__()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- `__contains__()` (*pytfa.MetaboliteThermo* method), 91
- `__contains__()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- `__deepcopy__()` (*pytfa.ThermoModel* method), 96, 112
- `__deepcopy__()` (*pytfa.thermo.ThermoModel* method), 107
- `__getitem__()` (*pytfa.MetaboliteThermo* method), 90
- `__getitem__()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- `__iter__()` (*pytfa.MetaboliteThermo* method), 91
- `__iter__()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- `__mul__()` (*pytfa.GenericVariable* method), 60
- `__mul__()` (*pytfa.optim.GenericVariable* method), 71, 78
- `__radd__()` (*pytfa.GenericVariable* method), 60
- `__radd__()` (*pytfa.optim.GenericVariable* method), 70, 78
- `__repr__()` (*pytfa.GenericConstraint* method), 48
- `__repr__()` (*pytfa.GenericVariable* method), 60
- `__repr__()` (*pytfa.MetaboliteThermo* method), 90
- `__repr__()` (*pytfa.optim.GenericConstraint* method), 67, 77
- `__repr__()` (*pytfa.optim.GenericVariable* method), 71, 79
- `__repr__()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- `__rmul__()` (*pytfa.GenericVariable* method), 60
- `__rmul__()` (*pytfa.optim.GenericVariable* method), 71, 78
- `__rsub__()` (*pytfa.GenericVariable* method), 60
- `__rsub__()` (*pytfa.optim.GenericVariable* method), 71, 78
- `__rtruediv__()` (*pytfa.GenericVariable* method), 60
- `__rtruediv__()` (*pytfa.optim.GenericVariable* method), 71, 79
- `__sub__()` (*pytfa.GenericVariable* method), 60
- `__sub__()` (*pytfa.optim.GenericVariable* method), 71, 78
- `__truediv__()` (*pytfa.GenericVariable* method), 60
- `__truediv__()` (*pytfa.optim.GenericVariable* method), 71, 78
- `__unicode__()` (*pytfa.MetaboliteThermo* method), 91
- `__unicode__()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- `_add_thermo_metabolite_info()` (in module *pytfa*), 37
- `_add_thermo_reaction_info()` (in module *pytfa*), 37
- `_bool2str()` (in module *pytfa*), 24
- `_bool2str()` (in module *pytfa.analysis*), 31
- `_build_lump()` (*pytfa.redgem.lumpgem.LumpGEM* method), 83
- `_calc_pka()` (*pytfa.MetaboliteThermo* method), 91
- `_calc_pka()` (*pytfa.thermo.MetaboliteThermo* method), 101, 109
- `_convert_metabolite()` (*pytfa.ThermoModel* method), 95, 112
- `_convert_metabolite()` (*pytfa.thermo.ThermoModel* method), 107
- `_convert_reaction()` (*pytfa.ThermoModel* method), 95, 112
- `_convert_reaction()` (*pytfa.thermo.ThermoModel* method), 107
- `_extract_inorganics()` (*redgem.RedGEM* method), 88
- `_generate_objective()` (*pytfa.redgem.lumpgem.LumpGEM* method), 83
- `_generate_usage_constraints()` (*pytfa.redgem.lumpgem.LumpGEM* method), 83
- `_init_thermo()` (*pytfa.ThermoModel* method), 94, 111
- `_init_thermo()` (*pytfa.thermo.ThermoModel* method), 106

`_lump_min_plus_p()` (*pytfa.redgem.lumpgem.LumpGEM method*), 83

`_lump_one_per_bbb()` (*pytfa.redgem.lumpgem.LumpGEM method*), 83

`_prepare_metabolite()` (*pytfa.ThermoModel method*), 94, 111

`_prepare_metabolite()` (*pytfa.thermo.ThermoModel method*), 106

`_prepare_reaction()` (*pytfa.ThermoModel method*), 95, 111

`_prepare_reaction()` (*pytfa.thermo.ThermoModel method*), 106

`_prepare_sinks()` (*pytfa.redgem.lumpgem.LumpGEM method*), 83

`_rebuild_stoichiometry()` (*in module pytfa*), 38

`_stoichiometry_to_dict()` (*in module pytfa*), 36

`_variability_analysis_element()` (*in module pytfa*), 25

`_variability_analysis_element()` (*in module pytfa.analysis*), 31

## A

`A_COUPLE` (*in module pytfa.thermo.std*), 93

`A_LITTLE` (*in module pytfa.thermo.std*), 93

`A_LOT` (*in module pytfa.thermo.std*), 93

`ABCRequirePrefixMeta` (*in module pytfa*), 51

`ABCRequirePrefixMeta` (*in module pytfa.optim*), 66, 69

`add_BBB_sinks()` (*in module redgem*), 81

`add_custom_classes()` (*in module pytfa*), 37

`add_lump()` (*in module redgem*), 88

`annotate_from_lexicon()` (*in module pytfa*), 38

`annotate_from_lexicon()` (*in module pytfa.io*), 46

`apply_compartment_data()` (*in module pytfa*), 39

`apply_compartment_data()` (*in module pytfa.io*), 46

`apply_directionality()` (*in module pytfa.analysis*), 32

`apply_directionality()` (*in module pytfa.analysis.manipulation*), 19

`apply_generic_variability()` (*in module pytfa.analysis*), 31

`apply_generic_variability()` (*in module pytfa.analysis.manipulation*), 19

`apply_reaction_variability()` (*in module pytfa.analysis*), 31

`apply_reaction_variability()` (*in module pytfa.analysis.manipulation*), 19

`archive_compositions()` (*in module pytfa*), 36

`archive_constraints()` (*in module pytfa*), 36

`archive_variables()` (*in module pytfa*), 36

## B

`BackwardDeltaGCoupling` (*class in pytfa*), 49

`BackwardDeltaGCoupling` (*class in pytfa.optim*), 68

`BackwardDeltaGCoupling` (*class in pytfa.thermo*), 104

`BackwardDirectionCoupling` (*class in pytfa*), 49

`BackwardDirectionCoupling` (*class in pytfa.optim*), 68

`BackwardDirectionCoupling` (*class in pytfa.thermo*), 104

`BackwardUseVariable` (*class in pytfa*), 61

`BackwardUseVariable` (*class in pytfa.optim*), 72, 77

`BackwardUseVariable` (*class in pytfa.thermo*), 105

`BASE_NAME2HOOK` (*in module pytfa*), 36

`BEST_THREAD_RATIO` (*in module pytfa*), 24

`BEST_THREAD_RATIO` (*in module pytfa.analysis*), 30

`BIGM` (*in module pytfa*), 18, 54, 94

`BIGM` (*in module pytfa.optim*), 75

`BIGM` (*in module pytfa.thermo*), 106

`BIGM` (*in module pytfa.utils.numerics*), 110

`BIGM_DG` (*in module pytfa*), 54, 94

`BIGM_DG` (*in module pytfa.io*), 43

`BIGM_DG` (*in module pytfa.optim*), 75

`BIGM_DG` (*in module pytfa.thermo*), 106

`BIGM_DG` (*in module pytfa.utils.numerics*), 110

`BIGM_P` (*in module pytfa*), 54, 94

`BIGM_P` (*in module pytfa.optim*), 75

`BIGM_P` (*in module pytfa.thermo*), 106

`BIGM_P` (*in module pytfa.utils.numerics*), 110

`BIGM_THERMO` (*in module pytfa*), 54, 94

`BIGM_THERMO` (*in module pytfa.optim*), 75

`BIGM_THERMO` (*in module pytfa.thermo*), 106

`BIGM_THERMO` (*in module pytfa.utils.numerics*), 110

`BinaryVariable` (*class in pytfa*), 61

`BinaryVariable` (*class in pytfa.optim*), 71

`breadth_search_extracellular_system_paths()` (*redgem.NetworkExpansion method*), 86

`breadth_search_subsystems_paths_length_d()` (*redgem.NetworkExpansion method*), 84

`build_thermo_from_equilibrator()` (*in module pytfa*), 89

## C

`calc_potential()` (*pytfa.MetaboliteThermo method*), 91

`calc_potential()` (*pytfa.thermo.MetaboliteThermo method*), 100, 109

`calcDGF_cues()` (*in module pytfa*), 93

`calcDGis()` (*pytfa.MetaboliteThermo method*), 91

`calcDGis()` (*pytfa.thermo.MetaboliteThermo method*), 100, 108

`calcDGR_cues()` (*in module pytfa*), 92

`calcDGR_cues()` (*in module pytfa.thermo*), 101

`calcDGsp()` (*pytfa.MetaboliteThermo method*), 91

`calcDGsp()` (*pytfa.thermo.MetaboliteThermo method*), 100, 109

`calcDGspA()` (*pytfa.MetaboliteThermo method*), 91

calcDGspA() (*pytfa.thermo.MetaboliteThermo method*), 101, 109  
 calcDGtpt\_rhs() (*in module pytfa*), 92  
 calcDGtpt\_rhs() (*in module pytfa.thermo*), 101  
 calculate\_dissipation() (*in module pytfa*), 25  
 calculate\_dissipation() (*in module pytfa.analysis*), 31  
 camel2underscores() (*in module pytfa*), 111  
 camel2underscores() (*in module pytfa.optim*), 66, 69  
 ccache (*in module pytfa*), 89  
 change\_expr() (*pytfa.GenericConstraint method*), 48  
 change\_expr() (*pytfa.optim.GenericConstraint method*), 67, 76  
 chebyshev\_center() (*in module pytfa*), 18  
 chebyshev\_transform() (*in module pytfa*), 18  
 ChebyshevRadius (*class in pytfa*), 18  
 check\_BBB\_production() (*in module redgem*), 81  
 check\_json\_extension() (*in module pytfa*), 40  
 check\_reaction\_balance() (*in module pytfa*), 96  
 check\_reaction\_balance() (*in module pytfa.thermo*), 102  
 check\_transport\_reaction() (*in module pytfa*), 97  
 check\_transport\_reaction() (*in module pytfa.thermo*), 102  
 chunk\_sum() (*in module pytfa*), 55  
 chunk\_sum() (*in module pytfa.optim*), 74, 79  
 compare\_solutions() (*in module pytfa*), 56  
 compare\_solutions() (*in module pytfa.optim*), 79  
 compound\_to\_entry() (*in module pytfa*), 89  
 compute\_dGf() (*in module pytfa*), 89  
 compute\_lumps() (*pytfa.redgem.lumpgem.LumpGEM method*), 83  
 cons\_to\_dict() (*in module pytfa*), 36  
 constraint (*pytfa.GenericConstraint property*), 48  
 constraint (*pytfa.optim.GenericConstraint property*), 67, 76  
 ConstraintTuple (*in module pytfa*), 52  
 convert() (*pytfa.thermo.ThermoModel method*), 107  
 convert() (*pytfa.ThermoModel method*), 95, 112  
 copy() (*pytfa.thermo.ThermoModel method*), 108  
 copy() (*pytfa.ThermoModel method*), 96, 112  
 copy\_solver\_configuration() (*in module pytfa*), 57  
 copy\_solver\_configuration() (*in module pytfa.optim*), 80  
 CPD\_PROTON (*in module pytfa*), 90  
 CPLEX (*in module pytfa.redgem.lumpgem*), 82  
 CPU\_COUNT (*in module pytfa*), 24  
 CPU\_COUNT (*in module pytfa.analysis*), 30  
 create\_generalized\_matrix() (*in module pytfa*), 33  
 create\_generalized\_matrix() (*in module pytfa.io*), 44  
 create\_new\_stoichiometric\_matrix() (*redgem.NetworkExpansion method*), 84  
 create\_problem\_dict() (*in module pytfa*), 33

create\_problem\_dict() (*in module pytfa.io*), 44  
 create\_thermo\_dict() (*in module pytfa*), 33  
 create\_thermo\_dict() (*in module pytfa.io*), 44

## D

debug\_iis() (*in module pytfa*), 50  
 DEBYE\_HUCKEL\_A (*in module pytfa.thermo.std*), 93  
 DEBYE\_HUCKEL\_B\_0 (*in module pytfa.thermo.std*), 93  
 default() (*pytfa.io.MyEncoder method*), 45  
 default() (*pytfa.MyEncoder method*), 39  
 DEFAULT\_EPS (*in module pytfa.redgem.lumpgem*), 82  
 DEFAULT\_VAL (*in module pytfa*), 90  
 DeltaG (*class in pytfa*), 62  
 DeltaG (*class in pytfa.analysis*), 30  
 DeltaG (*class in pytfa.optim*), 73  
 DeltaG (*class in pytfa.thermo*), 105  
 DeltaGErr (*class in pytfa*), 62  
 DeltaGErr (*class in pytfa.optim*), 72  
 DeltaGstd (*class in pytfa*), 62  
 DeltaGstd (*class in pytfa.optim*), 73, 75  
 DeltaGstd (*class in pytfa.thermo*), 105  
 dg\_relax\_config() (*in module pytfa*), 46  
 dg\_relax\_config() (*in module pytfa.optim*), 74  
 disambiguate (*in module pytfa.redgem.lumpgem*), 82  
 DisplacementCoupling (*class in pytfa*), 50  
 DisplacementCoupling (*class in pytfa.optim*), 69  
 DisplacementCoupling (*class in pytfa.thermo*), 104

## E

EPSILON (*in module pytfa*), 54, 94  
 EPSILON (*in module pytfa.optim*), 75  
 EPSILON (*in module pytfa.thermo*), 106  
 EPSILON (*in module pytfa.utils.numerics*), 110  
 evaluate\_constraint\_at\_solution() (*in module pytfa*), 56  
 evaluate\_constraint\_at\_solution() (*in module pytfa.optim*), 80  
 export\_reactions\_for\_escher() (*in module pytfa*), 41  
 export\_variable\_for\_escher() (*in module pytfa*), 41  
 expr (*pytfa.GenericConstraint property*), 48  
 expr (*pytfa.optim.GenericConstraint property*), 67, 76  
 extract\_sub\_network() (*redgem.NetworkExpansion method*), 87  
 extract\_subsystem\_metabolites() (*redgem.NetworkExpansion method*), 84  
 extract\_subsystem\_reactions() (*redgem.NetworkExpansion method*), 84

## F

fill\_default\_params() (*redgem.RedGEM method*), 87

[find\\_bidirectional\\_reactions\(\)](#) (in module *pytfa*), 24  
[find\\_bidirectional\\_reactions\(\)](#) (in module *pytfa.analysis*), 30  
[find\\_directionality\\_profiles\(\)](#) (in module *pytfa*), 24  
[find\\_directionality\\_profiles\(\)](#) (in module *pytfa.analysis*), 31  
[find\\_extreme\\_coeffs\(\)](#) (in module *pytfa*), 51  
[find\\_maxed\\_vars\(\)](#) (in module *pytfa*), 51  
[find\\_min\\_distance\\_between\\_subsystems\(\)](#) (*redgem.NetworkExpansion* method), 86  
[find\\_transporteds\\_mets\(\)](#) (in module *pytfa*), 97  
[find\\_transporteds\\_mets\(\)](#) (in module *pytfa.thermo*), 103  
[FluxKO](#) (class in *pytfa.redgem.lumpgem*), 82  
[ForbiddenProfile](#) (class in *pytfa*), 50  
[ForbiddenProfile](#) (class in *pytfa.analysis*), 30  
[ForbiddenProfile](#) (class in *pytfa.optim*), 69  
[Formula\\_regex](#) (in module *pytfa*), 96  
[ForwardBackwardUseVariable](#) (class in *pytfa*), 61  
[ForwardBackwardUseVariable](#) (class in *pytfa.optim*), 72  
[ForwardDeltaGCoupling](#) (class in *pytfa*), 49  
[ForwardDeltaGCoupling](#) (class in *pytfa.optim*), 68  
[ForwardDeltaGCoupling](#) (class in *pytfa.thermo*), 104  
[ForwardDirectionCoupling](#) (class in *pytfa*), 49  
[ForwardDirectionCoupling](#) (class in *pytfa.optim*), 68  
[ForwardDirectionCoupling](#) (class in *pytfa.thermo*), 104  
[ForwardUseVariable](#) (class in *pytfa*), 61  
[ForwardUseVariable](#) (class in *pytfa.analysis*), 30  
[ForwardUseVariable](#) (class in *pytfa.optim*), 72, 77  
[ForwardUseVariable](#) (class in *pytfa.thermo*), 105  
[from\\_constraints\(\)](#) (*pytfa.LinearizationConstraint* static method), 50  
[from\\_constraints\(\)](#) (*pytfa.optim.LinearizationConstraint* static method), 69

## G

[gene](#) (*pytfa.GeneConstraint* property), 48  
[gene](#) (*pytfa.GeneVariable* property), 61  
[gene](#) (*pytfa.optim.GeneConstraint* property), 67  
[gene](#) (*pytfa.optim.GeneVariable* property), 71  
[GeneConstraint](#) (class in *pytfa*), 48  
[GeneConstraint](#) (class in *pytfa.optim*), 67  
[GeneralizedACHRSampler](#) (class in *pytfa*), 21  
[GeneralizedACHRSampler](#) (class in *pytfa.analysis*), 27  
[GeneralizedHRSampler](#) (class in *pytfa*), 20  
[GeneralizedHRSampler](#) (class in *pytfa.analysis*), 26  
[GeneralizedOptGPSampler](#) (class in *pytfa*), 22  
[GeneralizedOptGPSampler](#) (class in *pytfa.analysis*), 28  
[generate\\_fva\\_warmup\(\)](#) (*pytfa.analysis.GeneralizedHRSampler*

method), 27  
[generate\\_fva\\_warmup\(\)](#) (*pytfa.GeneralizedHRSampler* method), 21  
[GenericConstraint](#) (class in *pytfa*), 47  
[GenericConstraint](#) (class in *pytfa.optim*), 66, 76  
[GenericVariable](#) (class in *pytfa*), 58  
[GenericVariable](#) (class in *pytfa.optim*), 69, 77  
[GeneVariable](#) (class in *pytfa*), 60  
[GeneVariable](#) (class in *pytfa.optim*), 71  
[get\\_active\\_use\\_variables\(\)](#) (in module *pytfa*), 56  
[get\\_active\\_use\\_variables\(\)](#) (in module *pytfa.optim*), 80  
[get\\_all\\_subclasses\(\)](#) (in module *pytfa*), 36, 55  
[get\\_all\\_subclasses\(\)](#) (in module *pytfa.optim*), 79  
[get\\_binary\\_type\(\)](#) (in module *pytfa*), 60  
[get\\_binary\\_type\(\)](#) (in module *pytfa.optim*), 71  
[get\\_bistream\\_logger\(\)](#) (in module *pytfa*), 110  
[get\\_bistream\\_logger\(\)](#) (in module *pytfa.analysis*), 30  
[get\\_bistream\\_logger\(\)](#) (in module *pytfa.thermo*), 106  
[get\\_cofactor\\_adjusted\\_stoich\(\)](#) (*pytfa.redgem.lumpgem.LumpGEM* method), 83  
[get\\_cons\\_var\\_classes\(\)](#) (in module *pytfa*), 19  
[get\\_debye\\_huckel\\_b\(\)](#) (in module *pytfa*), 93  
[get\\_debye\\_huckel\\_b\(\)](#) (in module *pytfa.thermo*), 102  
[get\\_direction\\_use\\_variables\(\)](#) (in module *pytfa*), 56  
[get\\_direction\\_use\\_variables\(\)](#) (in module *pytfa.analysis*), 30  
[get\\_direction\\_use\\_variables\(\)](#) (in module *pytfa.optim*), 80  
[get\\_hook\\_dict\(\)](#) (in module *pytfa*), 37  
[get\\_interface\(\)](#) (*pytfa.GenericConstraint* method), 47  
[get\\_interface\(\)](#) (*pytfa.GenericVariable* method), 59  
[get\\_interface\(\)](#) (*pytfa.optim.GenericConstraint* method), 66, 76  
[get\\_interface\(\)](#) (*pytfa.optim.GenericVariable* method), 69, 77  
[get\\_model\\_constraint\\_subclasses\(\)](#) (in module *pytfa*), 36  
[get\\_model\\_variable\\_subclasses\(\)](#) (in module *pytfa*), 36  
[get\\_operand\(\)](#) (*pytfa.GenericVariable* method), 59  
[get\\_operand\(\)](#) (*pytfa.optim.GenericVariable* method), 70, 78  
[get\\_pka\(\)](#) (*pytfa.MetaboliteThermo* method), 91  
[get\\_pka\(\)](#) (*pytfa.thermo.MetaboliteThermo* method), 101, 109  
[get\\_primal\(\)](#) (in module *pytfa*), 57  
[get\\_primal\(\)](#) (in module *pytfa.optim*), 80  
[get\\_reaction\\_compartment\(\)](#) (in module *pytfa*), 98  
[get\\_reaction\\_compartment\(\)](#) (in module *pytfa.thermo*), 103  
[get\\_reaction\\_data\(\)](#) (in module *pytfa*), 41

- `get_solution_value_for_variables()` (in module *pytfa*), 56  
`get_solution_value_for_variables()` (in module *pytfa.optim*), 74, 79  
`get_solver_string()` (in module *pytfa*), 37  
`get_timestr()` (in module *pytfa*), 110  
`get_variables()` (in module *pytfa*), 19  
`glovers_linearization()` (in module *pytfa*), 52  
GLPK (in module *pytfa.redgem.lumpgem*), 82  
GUROBI (in module *pytfa.redgem.lumpgem*), 82
- ## I
- `id` (*pytfa.GeneConstraint* property), 48  
`id` (*pytfa.GenericConstraint* property), 48  
`id` (*pytfa.GenericVariable* property), 59  
`id` (*pytfa.GeneVariable* property), 61  
`id` (*pytfa.MetaboliteConstraint* property), 49  
`id` (*pytfa.MetaboliteVariable* property), 61  
`id` (*pytfa.optim.GeneConstraint* property), 67  
`id` (*pytfa.optim.GenericConstraint* property), 67, 76  
`id` (*pytfa.optim.GenericVariable* property), 70, 77  
`id` (*pytfa.optim.GeneVariable* property), 71  
`id` (*pytfa.optim.MetaboliteConstraint* property), 68  
`id` (*pytfa.optim.MetaboliteVariable* property), 72  
`id` (*pytfa.optim.ReactionConstraint* property), 67  
`id` (*pytfa.optim.ReactionVariable* property), 72  
`id` (*pytfa.ReactionConstraint* property), 48  
`id` (*pytfa.ReactionVariable* property), 61  
`id` (*pytfa.thermo.MetaboliteConstraint* property), 104  
`id` (*pytfa.thermo.MetaboliteVariable* property), 106  
`id` (*pytfa.thermo.ReactionConstraint* property), 104  
`id` (*pytfa.thermo.ReactionVariable* property), 105  
`import_matlab_model()` (in module *pytfa*), 33  
`import_matlab_model()` (in module *pytfa.io*), 43  
InfeasibleExcept, 82  
`init_params()` (*pytfa.redgem.lumpgem.LumpGEM* method), 83  
`init_thermo_model_from_dict()` (in module *pytfa*), 37  
INTEGER\_VARIABLE\_TYPES (in module *pytfa*), 55  
INTEGER\_VARIABLE\_TYPES (in module *pytfa.optim*), 79  
`is_exchange()` (in module *pytfa*), 98  
`is_inequality()` (in module *pytfa*), 18  
`is_node_allowed()` (*redgem.NetworkExpansion* method), 85  
`is_node_allowed_extracellular()` (*redgem.NetworkExpansion* method), 86  
`is_same_stoichiometry()` (in module *pytfa*), 98  
`items()` (*pytfa.MetaboliteThermo* method), 90  
`items()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- ## J
- `json_dumps_model()` (in module *pytfa*), 40
- ## K
- `keys()` (*pytfa.MetaboliteThermo* method), 90  
`keys()` (*pytfa.thermo.MetaboliteThermo* method), 100, 108
- ## L
- LinearizationConstraint (class in *pytfa*), 50  
LinearizationConstraint (class in *pytfa.optim*), 69  
LinearizationVariable (class in *pytfa*), 63  
LinearizationVariable (class in *pytfa.optim*), 73  
`linearize_product()` (in module *pytfa*), 53  
`load_json_model()` (in module *pytfa*), 40  
`load_thermoDB()` (in module *pytfa*), 34  
`load_thermoDB()` (in module *pytfa.io*), 44  
LogConcentration (class in *pytfa*), 62  
LogConcentration (class in *pytfa.optim*), 72, 75  
LogConcentration (class in *pytfa.thermo*), 105  
LOGFOLDERNAME (in module *pytfa*), 110  
logger (in module *pytfa*), 89  
Lump (in module *pytfa.redgem.lumpgem*), 82  
LumpGEM (class in *pytfa.redgem.lumpgem*), 83
- ## M
- `make_name()` (*pytfa.GenericConstraint* method), 48  
`make_name()` (*pytfa.GenericVariable* method), 59  
`make_name()` (*pytfa.optim.GenericConstraint* method), 66, 76  
`make_name()` (*pytfa.optim.GenericVariable* method), 70, 77  
`make_result()` (*pytfa.GenericVariable* method), 60  
`make_result()` (*pytfa.optim.GenericVariable* method), 71, 79  
`make_sink()` (in module *redgem*), 81  
`make_subclasses_dict()` (in module *pytfa*), 36  
MANY (in module *pytfa.thermo.std*), 93  
MAX\_PH (in module *pytfa.thermo.std*), 93  
MAX\_STOICH (in module *pytfa*), 94  
MAX\_STOICH (in module *pytfa.thermo*), 106  
metabolite (*pytfa.MetaboliteConstraint* property), 49  
metabolite (*pytfa.MetaboliteVariable* property), 61  
metabolite (*pytfa.optim.MetaboliteConstraint* property), 68  
metabolite (*pytfa.optim.MetaboliteVariable* property), 72  
metabolite (*pytfa.thermo.MetaboliteConstraint* property), 104  
metabolite (*pytfa.thermo.MetaboliteVariable* property), 106  
`metabolite_thermo_to_dict()` (in module *pytfa*), 36  
MetaboliteConstraint (class in *pytfa*), 49  
MetaboliteConstraint (class in *pytfa.optim*), 67  
MetaboliteConstraint (class in *pytfa.thermo*), 104

MetaboliteThermo (class in *pytfa*), 90  
MetaboliteThermo (class in *pytfa.thermo*), 99, 108  
MetaboliteVariable (class in *pytfa*), 61  
MetaboliteVariable (class in *pytfa.optim*), 72  
MetaboliteVariable (class in *pytfa.thermo*), 106  
min\_BBB\_uptake() (in module *redgem*), 81  
MIN\_PH (in module *pytfa.thermo.std*), 93  
model (*pytfa.GeneConstraint* property), 48  
model (*pytfa.GenericConstraint* property), 48  
model (*pytfa.GenericVariable* property), 59  
model (*pytfa.GeneVariable* property), 61  
model (*pytfa.MetaboliteConstraint* property), 49  
model (*pytfa.MetaboliteVariable* property), 61  
model (*pytfa.optim.GeneConstraint* property), 67  
model (*pytfa.optim.GenericConstraint* property), 67, 77  
model (*pytfa.optim.GenericVariable* property), 70, 78  
model (*pytfa.optim.GeneVariable* property), 71  
model (*pytfa.optim.MetaboliteConstraint* property), 68  
model (*pytfa.optim.MetaboliteVariable* property), 72  
model (*pytfa.optim.ReactionConstraint* property), 67  
model (*pytfa.optim.ReactionVariable* property), 72  
model (*pytfa.ReactionConstraint* property), 49  
model (*pytfa.ReactionVariable* property), 61  
model (*pytfa.thermo.MetaboliteConstraint* property), 104  
model (*pytfa.thermo.MetaboliteVariable* property), 106  
model (*pytfa.thermo.ReactionConstraint* property), 104  
model (*pytfa.thermo.ReactionVariable* property), 106  
model\_from\_dict() (in module *pytfa*), 37  
model\_to\_dict() (in module *pytfa*), 37  
ModelConstraint (class in *pytfa*), 48  
ModelConstraint (class in *pytfa.optim*), 67  
ModelVariable (class in *pytfa*), 60  
ModelVariable (class in *pytfa.optim*), 71  
module  
    *pytfa*, 17, 20, 23, 32, 34, 38–41, 46, 47, 50, 51, 53, 54, 57, 88, 89, 92, 94, 96, 110, 111  
    *pytfa.analysis*, 17  
    *pytfa.analysis.chebyshev*, 17  
    *pytfa.analysis.manipulation*, 19  
    *pytfa.analysis.sampling*, 20  
    *pytfa.analysis.viability*, 23  
    *pytfa.core*, 32  
    *pytfa.io*, 32  
    *pytfa.io.base*, 32  
    *pytfa.io.dict*, 34  
    *pytfa.io.enrichment*, 38  
    *pytfa.io.json*, 39  
    *pytfa.io.plotting*, 40  
    *pytfa.io.viz*, 41  
    *pytfa.optim*, 46  
    *pytfa.optim.config*, 46  
    *pytfa.optim.constraints*, 47  
    *pytfa.optim.debugging*, 50  
    *pytfa.optim.meta*, 51

*pytfa.optim.reformulation*, 51  
*pytfa.optim.relaxation*, 53  
*pytfa.optim.utils*, 54  
*pytfa.optim.variables*, 57  
*pytfa.redgem*, 81  
*pytfa.redgem.debugging*, 81  
*pytfa.redgem.lumpgem*, 81  
*pytfa.redgem.network\_expansion*, 84  
*pytfa.redgem.redgem*, 87  
*pytfa.redgem.utils*, 88  
*pytfa.thermo*, 88  
*pytfa.thermo.equilibrator*, 88  
*pytfa.thermo.metabolite*, 89  
*pytfa.thermo.reaction*, 92  
*pytfa.thermo.std*, 93  
*pytfa.thermo.tmodel*, 94  
*pytfa.thermo.utils*, 96  
*pytfa.utils*, 110  
*pytfa.utils.logger*, 110  
*pytfa.utils.numerics*, 110  
*pytfa.utils.str*, 111  
*redgem*, 81, 84, 87

MyEncoder (class in *pytfa*), 39

MyEncoder (class in *pytfa.io*), 45

## N

name (*pytfa.GenericConstraint* property), 48  
name (*pytfa.GenericVariable* property), 59  
name (*pytfa.optim.GenericConstraint* property), 67, 76  
name (*pytfa.optim.GenericVariable* property), 70, 77  
NegativeDeltaG (class in *pytfa*), 49  
NegativeDeltaG (class in *pytfa.optim*), 68, 73  
NegativeDeltaG (class in *pytfa.thermo*), 103  
NegSlackLC (class in *pytfa*), 63  
NegSlackLC (class in *pytfa.optim*), 73, 75  
NegSlackVariable (class in *pytfa*), 62  
NegSlackVariable (class in *pytfa.optim*), 73, 75  
NetworkExpansion (class in *redgem*), 84  
normalize\_reactions() (*pytfa.thermo.ThermoModel* method), 106  
normalize\_reactions() (*pytfa.ThermoModel* method), 94, 111

## O

obj\_to\_dict() (in module *pytfa*), 37  
op\_replace\_dict (in module *pytfa*), 58  
op\_replace\_dict (in module *pytfa.optim*), 69  
OPTLANG\_BINARY (in module *pytfa*), 52

## P

parallel\_variability\_analysis() (in module *pytfa*), 25  
parallel\_variability\_analysis() (in module *pytfa.analysis*), 31

petersen\_linearization() (in module pytfa), 52  
 plot\_fva\_tva\_comparison() (in module pytfa), 40  
 plot\_histogram() (in module pytfa), 41  
 plot\_thermo\_displacement\_histogram() (in module pytfa), 40  
 PosSlackLC (class in pytfa), 62  
 PosSlackLC (class in pytfa.optim), 73, 75  
 PosSlackVariable (class in pytfa), 62  
 PosSlackVariable (class in pytfa.optim), 73, 74  
 prefix (pytfa.analysis.DeltaG attribute), 30  
 prefix (pytfa.analysis.ForbiddenProfile attribute), 30  
 prefix (pytfa.analysis.ForwardUseVariable attribute), 30  
 prefix (pytfa.BackwardDeltaGCoupling attribute), 49  
 prefix (pytfa.BackwardDirectionCoupling attribute), 50  
 prefix (pytfa.BackwardUseVariable attribute), 61  
 prefix (pytfa.BinaryVariable attribute), 61  
 prefix (pytfa.ChebyshevRadius attribute), 18  
 prefix (pytfa.DeltaG attribute), 62  
 prefix (pytfa.DeltaGErr attribute), 62  
 prefix (pytfa.DeltaGstd attribute), 62  
 prefix (pytfa.DisplacementCoupling attribute), 50  
 prefix (pytfa.ForbiddenProfile attribute), 50  
 prefix (pytfa.ForwardBackwardUseVariable attribute), 62  
 prefix (pytfa.ForwardDeltaGCoupling attribute), 49  
 prefix (pytfa.ForwardDirectionCoupling attribute), 49  
 prefix (pytfa.ForwardUseVariable attribute), 61  
 prefix (pytfa.GeneConstraint attribute), 48  
 prefix (pytfa.GenericConstraint attribute), 47  
 prefix (pytfa.GenericVariable attribute), 59  
 prefix (pytfa.GeneVariable attribute), 60  
 prefix (pytfa.LinearizationConstraint attribute), 50  
 prefix (pytfa.LinearizationVariable attribute), 63  
 prefix (pytfa.LogConcentration attribute), 62  
 prefix (pytfa.MetaboliteConstraint attribute), 49  
 prefix (pytfa.MetaboliteVariable attribute), 61  
 prefix (pytfa.ModelConstraint attribute), 48  
 prefix (pytfa.ModelVariable attribute), 60  
 prefix (pytfa.NegativeDeltaG attribute), 49  
 prefix (pytfa.NegSlackLC attribute), 63  
 prefix (pytfa.NegSlackVariable attribute), 62  
 prefix (pytfa.optim.BackwardDeltaGCoupling attribute), 68  
 prefix (pytfa.optim.BackwardDirectionCoupling attribute), 68  
 prefix (pytfa.optim.BackwardUseVariable attribute), 72, 77  
 prefix (pytfa.optim.BinaryVariable attribute), 71  
 prefix (pytfa.optim.DeltaG attribute), 73  
 prefix (pytfa.optim.DeltaGErr attribute), 73  
 prefix (pytfa.optim.DeltaGstd attribute), 73, 75  
 prefix (pytfa.optim.DisplacementCoupling attribute), 69  
 prefix (pytfa.optim.ForbiddenProfile attribute), 69  
 prefix (pytfa.optim.ForwardBackwardUseVariable attribute), 72  
 prefix (pytfa.optim.ForwardDeltaGCoupling attribute), 68  
 prefix (pytfa.optim.ForwardDirectionCoupling attribute), 68  
 prefix (pytfa.optim.ForwardUseVariable attribute), 72, 77  
 prefix (pytfa.optim.GeneConstraint attribute), 67  
 prefix (pytfa.optim.GenericConstraint attribute), 66, 76  
 prefix (pytfa.optim.GenericVariable attribute), 69, 77  
 prefix (pytfa.optim.GeneVariable attribute), 71  
 prefix (pytfa.optim.LinearizationConstraint attribute), 69  
 prefix (pytfa.optim.LinearizationVariable attribute), 73  
 prefix (pytfa.optim.LogConcentration attribute), 72, 75  
 prefix (pytfa.optim.MetaboliteConstraint attribute), 68  
 prefix (pytfa.optim.MetaboliteVariable attribute), 72  
 prefix (pytfa.optim.ModelConstraint attribute), 67  
 prefix (pytfa.optim.ModelVariable attribute), 71  
 prefix (pytfa.optim.NegativeDeltaG attribute), 68, 74  
 prefix (pytfa.optim.NegSlackLC attribute), 73, 75  
 prefix (pytfa.optim.NegSlackVariable attribute), 73, 75  
 prefix (pytfa.optim.PosSlackLC attribute), 73, 75  
 prefix (pytfa.optim.PosSlackVariable attribute), 73, 75  
 prefix (pytfa.optim.ReactionConstraint attribute), 67  
 prefix (pytfa.optim.ReactionVariable attribute), 72  
 prefix (pytfa.optim.SimultaneousUse attribute), 69  
 prefix (pytfa.optim.ThermoDisplacement attribute), 73  
 prefix (pytfa.PosSlackLC attribute), 62  
 prefix (pytfa.PosSlackVariable attribute), 62  
 prefix (pytfa.ReactionConstraint attribute), 48  
 prefix (pytfa.ReactionVariable attribute), 61  
 prefix (pytfa.redgem.lumpgem.FluxKO attribute), 82  
 prefix (pytfa.redgem.lumpgem.UseOrKOFlux attribute), 83  
 prefix (pytfa.redgem.lumpgem.UseOrKOInt attribute), 82  
 prefix (pytfa.SimultaneousUse attribute), 50  
 prefix (pytfa.thermo.BackwardDeltaGCoupling attribute), 104  
 prefix (pytfa.thermo.BackwardDirectionCoupling attribute), 104  
 prefix (pytfa.thermo.BackwardUseVariable attribute), 105  
 prefix (pytfa.thermo.DeltaG attribute), 105  
 prefix (pytfa.thermo.DeltaGstd attribute), 105  
 prefix (pytfa.thermo.DisplacementCoupling attribute), 105  
 prefix (pytfa.thermo.ForwardDeltaGCoupling attribute), 104  
 prefix (pytfa.thermo.ForwardDirectionCoupling attribute), 104

prefix (*pytfa.thermo.ForwardUseVariable* attribute), 105  
 prefix (*pytfa.thermo.LogConcentration* attribute), 105  
 prefix (*pytfa.thermo.MetaboliteConstraint* attribute), 104  
 prefix (*pytfa.thermo.MetaboliteVariable* attribute), 106  
 prefix (*pytfa.thermo.NegativeDeltaG* attribute), 103  
 prefix (*pytfa.thermo.ReactionConstraint* attribute), 104  
 prefix (*pytfa.thermo.ReactionVariable* attribute), 105  
 prefix (*pytfa.thermo.SimultaneousUse* attribute), 103  
 prefix (*pytfa.thermo.ThermoDisplacement* attribute), 105  
 prefix (*pytfa.ThermoDisplacement* attribute), 62  
 prepare() (*pytfa.thermo.ThermoModel* method), 107  
 prepare() (*pytfa.ThermoModel* method), 95, 111  
 print\_info() (*pytfa.thermo.ThermoModel* method), 107  
 print\_info() (*pytfa.ThermoModel* method), 95, 112  
 printLP() (in module *pytfa*), 34  
 printLP() (in module *pytfa.io*), 44  
 pytfa  
     module, 17, 20, 23, 32, 34, 38–41, 46, 47, 50, 51, 53, 54, 57, 88, 89, 92, 94, 96, 110, 111  
 pytfa.analysis  
     module, 17  
 pytfa.analysis.chebyshev  
     module, 17  
 pytfa.analysis.manipulation  
     module, 19  
 pytfa.analysis.sampling  
     module, 20  
 pytfa.analysis.variability  
     module, 23  
 pytfa.core  
     module, 32  
 pytfa.io  
     module, 32  
 pytfa.io.base  
     module, 32  
 pytfa.io.dict  
     module, 34  
 pytfa.io.enrichment  
     module, 38  
 pytfa.io.json  
     module, 39  
 pytfa.io.plotting  
     module, 40  
 pytfa.io.viz  
     module, 41  
 pytfa.optim  
     module, 46  
 pytfa.optim.config  
     module, 46  
 pytfa.optim.constraints  
     module, 47  
 pytfa.optim.debugging  
     module, 50  
 pytfa.optim.meta  
     module, 51  
 pytfa.optim.reformulation  
     module, 51  
 pytfa.optim.relaxation  
     module, 53  
 pytfa.optim.utils  
     module, 54  
 pytfa.optim.variables  
     module, 57  
 pytfa.redgem  
     module, 81  
 pytfa.redgem.debugging  
     module, 81  
 pytfa.redgem.lumpgem  
     module, 81  
 pytfa.redgem.network\_expansion  
     module, 84  
 pytfa.redgem.redgem  
     module, 87  
 pytfa.redgem.utils  
     module, 88  
 pytfa.thermo  
     module, 88  
 pytfa.thermo.equilibrator  
     module, 88  
 pytfa.thermo.metabolite  
     module, 89  
 pytfa.thermo.reaction  
     module, 92  
 pytfa.thermo.std  
     module, 93  
 pytfa.thermo.tmodel  
     module, 94  
 pytfa.thermo.utils  
     module, 96  
 pytfa.utils  
     module, 110  
 pytfa.utils.logger  
     module, 110  
 pytfa.utils.numerics  
     module, 110  
 pytfa.utils.str  
     module, 111

## R

reaction (*pytfa.optim.ReactionConstraint* property), 67  
 reaction (*pytfa.optim.ReactionVariable* property), 72  
 reaction (*pytfa.ReactionConstraint* property), 48  
 reaction (*pytfa.ReactionVariable* property), 61

- reaction (*pytfa.thermo.ReactionConstraint* property), 104  
 reaction (*pytfa.thermo.ReactionVariable* property), 105  
 ReactionConstraint (class in *pytfa*), 48  
 ReactionConstraint (class in *pytfa.optim*), 67  
 ReactionConstraint (class in *pytfa.thermo*), 104  
 ReactionVariable (class in *pytfa*), 61  
 ReactionVariable (class in *pytfa.optim*), 72  
 ReactionVariable (class in *pytfa.thermo*), 105  
 read\_compartment\_data() (in module *pytfa*), 39  
 read\_compartment\_data() (in module *pytfa.io*), 46  
 read\_lexicon() (in module *pytfa*), 39  
 read\_lexicon() (in module *pytfa.io*), 46  
 read\_parameters() (*redgem.RedGEM* method), 87  
 rebuild\_compositions() (in module *pytfa*), 37  
 rebuild\_obj\_from\_dict() (in module *pytfa*), 37  
 recover\_compartments() (in module *pytfa*), 33  
 recover\_compartments() (in module *pytfa.io*), 43  
 redgem  
   module, 81, 84, 87  
 RedGEM (class in *redgem*), 87  
 relax\_dgo() (in module *pytfa*), 54  
 relax\_dgo() (in module *pytfa.optim*), 75  
 relax\_dgo\_gurobi() (in module *pytfa*), 54  
 relax\_dgo\_gurobi() (in module *pytfa.optim*), 75  
 relax\_lc() (in module *pytfa*), 54  
 relax\_lc() (in module *pytfa.optim*), 76  
 remove\_blocked\_reactions() (in module *pytfa.redgem.utils*), 88  
 RequirePrefixMeta (class in *pytfa*), 51  
 retrieve\_all\_paths() (*redgem.NetworkExpansion* method), 85  
 retrieve\_intermediate\_extracellular\_metabolites\_and\_reactions() (*redgem.NetworkExpansion* method), 86  
 retrieve\_intermediate\_metabolites\_and\_reactions() (*redgem.NetworkExpansion* method), 85  
 run() (*redgem.NetworkExpansion* method), 87  
 run() (*redgem.RedGEM* method), 87  
 run\_between\_all\_subsystems() (*redgem.NetworkExpansion* method), 87  
 run\_extracellular\_system() (*redgem.NetworkExpansion* method), 87
- ## S
- sample() (in module *pytfa*), 22  
 sample() (in module *pytfa.analysis*), 29  
 save\_json\_model() (in module *pytfa*), 40  
 scaling\_factor (*pytfa.GenericVariable* property), 59  
 scaling\_factor (*pytfa.optim.GenericVariable* property), 70, 78  
 set\_medium() (in module *pytfa.redgem.utils*), 88  
 set\_solver() (*redgem.RedGEM* method), 87  
 SimultaneousUse (class in *pytfa*), 50  
 SimultaneousUse (class in *pytfa.optim*), 69  
 SimultaneousUse (class in *pytfa.thermo*), 103  
 SOLVER\_DICT (in module *pytfa*), 36  
 strip\_from\_integer\_variables() (in module *pytfa*), 57  
 strip\_from\_integer\_variables() (in module *pytfa.optim*), 80  
 subs\_bilinear() (in module *pytfa*), 52  
 sum\_reactions() (in module *pytfa.redgem.lumpgem*), 84  
 symbol\_sum() (in module *pytfa*), 56  
 symbol\_sum() (in module *pytfa.optim*), 74, 79  
 SYMPY\_ADD\_CHUNKSIZE (in module *pytfa*), 55  
 SYMPY\_ADD\_CHUNKSIZE (in module *pytfa.optim*), 79
- ## T
- TEMPERATURE\_0 (in module *pytfa.thermo.std*), 93  
 test\_consistency() (*pytfa.GenericVariable* method), 59  
 test\_consistency() (*pytfa.optim.GenericVariable* method), 70, 78  
 ThermoDisplacement (class in *pytfa*), 62  
 ThermoDisplacement (class in *pytfa.optim*), 73  
 ThermoDisplacement (class in *pytfa.thermo*), 105  
 ThermoModel (class in *pytfa*), 94, 111  
 ThermoModel (class in *pytfa.thermo*), 106  
 TimeoutExcept, 82  
 trim\_epsilon\_mets() (in module *pytfa.redgem.utils*), 88  
 type (*pytfa.GenericVariable* property), 59  
 type (*pytfa.optim.GenericVariable* property), 70, 78
- ## U
- unscaled (*pytfa.GenericVariable* property), 59  
 unscaled (*pytfa.optim.GenericVariable* property), 70, 78  
 unscaled\_value (*pytfa.GenericVariable* property), 59  
 unscaled\_value (*pytfa.optim.GenericVariable* property), 70, 78  
 UseOrK0Flux (class in *pytfa.redgem.lumpgem*), 83  
 UseOrK0Int (class in *pytfa.redgem.lumpgem*), 82
- ## V
- value (*pytfa.GenericVariable* property), 59  
 value (*pytfa.optim.GenericVariable* property), 70, 78  
 values() (*pytfa.MetaboliteThermo* method), 90  
 values() (*pytfa.thermo.MetaboliteThermo* method), 100, 108  
 var\_to\_dict() (in module *pytfa*), 36  
 variability\_analysis() (in module *pytfa*), 25  
 variability\_analysis() (in module *pytfa.analysis*), 31  
 variable (*pytfa.GenericVariable* property), 59  
 variable (*pytfa.optim.GenericVariable* property), 70, 78  
 varnames2ids() (in module *pytfa*), 111

`varnames2matlab()` (*in module pytfa*), [33](#)  
`varnames2matlab()` (*in module pytfa.io*), [44](#)

## W

`write_compartment_data()` (*in module pytfa*), [39](#)  
`write_compartment_data()` (*in module pytfa.io*), [46](#)  
`write_lexicon()` (*in module pytfa*), [38](#)  
`write_lexicon()` (*in module pytfa.io*), [45](#)  
`write_matlab_model()` (*in module pytfa*), [33](#)  
`write_matlab_model()` (*in module pytfa.io*), [44](#)  
`writeLP()` (*in module pytfa*), [34](#)  
`writeLP()` (*in module pytfa.io*), [45](#)