equilibrator

Elad Noor

Jul 31, 2023

CONTENTS

1 Current Features **3**

2 How to cite us **5**

3 How to install **7**

4 How to use **9**
- 4.1 Installation . . . 9
- 4.2 Tutorial . . . 11
- 4.3 Covariance . . . 17
- 4.4 Code examples . . . 18
- 4.5 Local cache . . . 32
- 4.6 API Reference . . . 41

5 References **63**

Python Module Index **65**

Index **67**

`equilibrator-api` is a command-line API with minimal dependencies for calculation of standard thermodynamic potentials of biochemical reactions using the same data found on eQuilibrator website. It can be used without a network connection (after installation and initialization).

CHAPTER

ONE

CURRENT FEATURES

- Calculation of standard Gibbs potentials of reactions (together with a represnetation of uncertainties by a full covariance matrix).
- Calculation of standard Gibbs potentials of transport reactions (i.e. between cellular compartments).
- Calculation of standard reduction potentials of half-cells.
- Adjustment of Gibbs free energies to pH, ionic strength, and pMg (Magnesium ion concentration).
- Pathway analysis tools such as Max-min Driving Force and Enzyme Cost Minimization (requires the equilibrator-pathway package).
- Adding new compounds that are not among the 500,000 currently in the MetaNetX database (requires the equilibrator-assets package).

CHAPTER

TWO

HOW TO CITE US

If you plan to use results from `equilibrator-api` in a scientific publication, please cite our paper: *Consistent Estimation of Gibbs Energy Using Component Contributions*[1]

[1] Elad Noor, Hulda S. Haraldsdóttir, Ron Milo, and Ronan M. T. Fleming. Consistent Estimation of Gibbs Energy Using Component Contributions. *PLOS Computational Biology*, 9(7):e1003098, July 2013. URL: http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003098 (visited on 2017-12-13), doi:10.1371/journal.pcbi.1003098.

CHAPTER

THREE

HOW TO INSTALL

You can simply `pip install equilibrator-api` and start eQuilibrating. For more details, see *Installation*.

CHAPTER

FOUR

HOW TO USE

For all newcommers, we recommend reading the *Tutorial* and even taking a look at the *API Reference* for a full list of classes and functions.

Alternatively, if you are looking for a quick-and-dirty guide, you can take a look at the *Code examples* section.

4.1 Installation

4.1.1 Step 1

Install eQuilibrator using pip (ideally, inside a virtual environment):

```
pip install equilibrator-api
```

If you are using Windows OS, we recommend using *conda* instead of *pip*:

```
conda install -c conda-forge equilibrator-api
```

4.1.2 Step 2 (optional)

Run this command to initialize eQuilibrator:

```
python -c "from equilibrator_api import ComponentContribution; cc =␣
↪ComponentContribution()"
```

Note, that this can take minutes or even up to an hour, since about 1.3 GBytes of data need to be downloaded from a remote website (Zenodo). If this command fails, try improving the speed of your connection (e.g. disabling your VPN, or using a LAN cable to connect to your router) and running it again.

Note that you don't have to run this command before using eQuilibrator. It will simply download the database on the first time you try using it (e.g. inside the Jupyter notebook). In any case, after downloading the database the data will be locally cached and loading takes only a few seconds from then onwards.

4.1.3 Step 3 (optional)

Now, you are good to go. In case you want to see an example of how to use eQuilibrator-API in the form of a Jupyter notebook, run the following commands:

```
pip install jupyter
curl https://gitlab.com/equilibrator/equilibrator-api/-/raw/develop/scripts/equilibrator_
↪cmd.ipynb > equilibrator_cmd.ipynb
jupyter notebook
```

and select the notebook called *equilibrator_cmd.ipynb* and follow the examples in it.

4.1.4 Dependencies

- python >= 3.8
- path
- numpy
- scipy
- pandas
- pyparsing
- tqdm
- appdirs
- diskcache
- httpx
- tenacity
- python-slugify
- periodictable
- sqlalchemy
- Levenshtein
- pint
- uncertainties
- cobra (optional)
- sbtab (optional)

4.2 Tutorial

4.2.1 Introduction to Component Contributions

All of eQuilibrator's Gibbs energy estimates are based on the *Component Contribution* method[1], which combines *Group Contribution*[2345] with the more accurate *Reactant Contribution*[6] by decomposing each reaction into two parts and applying one of the methods to each of them. This method gives priority to the reactant contributions over group contributions while guaranteeing that all estimates are consistent, i.e. will not violate the first law of thermodynamics.

This tutorial focuses on usage and applications, and these do not require understanding the underlying calculations. However, a reader who is interested in learning how Component Contribution works, would benefit from reading the original publication[1].

4.2.2 Parsing chemical formulas

In this section, we will learn how to convert strings containing reaction formulae to `Reaction` objects, using the parsing functions in `equilibrator-api`.

First, one must create an object from the `ComponentContribution` class:

```
from equilibrator_api import ComponentContribution, Q_
cc = ComponentContribution()
```

Calling the constructor can take a few seconds, since it is loading the entire database of compounds into RAM. **Important notice:** when you do this for the first time after installing `equilibrator-api`, this database will be downloaded from a remote website (Zenodo) and that can take about 10 minutes (or more, depending on your bandwidth).

Notice that we also imported `equilibrator_api.Q_`, which is a shorthand to the `Quantity` object that will help us define physical values with proper units. The default aqueous conditions are pH of 7.5, ionic strenght of 250 mM, and pMg of 3. You can change any or all of them by direct assignment:

```
cc.p_h = Q_(7.0)
cc.p_mg = Q_(2.5)
cc.ionic_strength = Q_("0.1M")
```

Note that changing these aqueous conditions can be also be done later on (e.g. for checking how sensitive a reaction's Gibbs energy is to pH).

[1] Elad Noor, Hulda S. Haraldsdóttir, Ron Milo, and Ronan M. T. Fleming. Consistent Estimation of Gibbs Energy Using Component Contributions. *PLOS Computational Biology*, 9(7):e1003098, July 2013. URL: http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003098 (visited on 2017-12-13), doi:10.1371/journal.pcbi.1003098.

[2] L Michael Mavrovouniotis, Patrick Bayol, Michael Tu-Kien Lam, George Stephanopoulos, and Gregory Stephanopoulos. A group contribution method for the estimation of equilibrium constants for biochemical reactions. *Biotechnol. Tech.*, 2:23–28, March 1988.

[3] L Michael Mavrovouniotis. Group contributions for estimating standard Gibbs energies of formation of biochemical compounds in aqueous solution. *Biotechnol. Bioeng.*, 36:1070–1082, October 1990.

[4] M L Mavrovouniotis. Estimation of standard Gibbs energy changes of biotransformations. *The Journal of Biological Chemistry*, 266(22):14440–14445, August 1991. URL: http://www.ncbi.nlm.nih.gov/pubmed/1860851.

[5] Matthew D. Jankowski, Christopher S. Henry, Linda J. Broadbelt, and Vassily Hatzimanikatis. Group Contribution Method for Thermodynamic Analysis of Complex Metabolic Networks. *Biophysical Journal*, 95(3):1487–1499, August 2008. URL: http://www.sciencedirect.com/science/article/pii/S0006349508702157 (visited on 2018-10-30), doi:10.1529/biophysj.107.124784.

[6] Elad Noor, Arren Bar-Even, Avi Flamholz, Yaniv Lubling, Dan Davidi, and Ron Milo. An integrated open framework for thermodynamics of reactions that combines accuracy and coverage. *Bioinformatics*, 28(15):2037–2044, August 2012. URL: https://academic.oup.com/bioinformatics/article/28/15/2037/237811 (visited on 2018-06-20), doi:10.1093/bioinformatics/bts317.

Creating a `Compound` object

First, we will see how to find compounds in the database (which we call the `compound_cache`). The simplest way is to have an accession from on of the supported repositories:

Namespace	Pattern	Example
metanetx.chemical	MNXM{int}	metanetx.chemical:MNXM5
bigg.metabolite	{str}	bigg.metabolite:h2o
kegg	{C/D/G}{int:05d}	kegg:C00001
chebi	CHEBI:{int}	chebi:CHEBI:30616
sabiork.compound	{int}	sabiork.compound:34
metacyc.compound	{str} or CPD-{int}	metacyc.compound:ATP
hmdb	HMDB{int}	hmdb:HMDB0000538
swisslipid	SLM:{int}	swisslipid:SLM:000000002
reactome	R-ALL-{int}	reactome:R-ALL-113592
lipidmaps	LM**{int}	lipidmaps:LMFA00000001
seed	cpd{int:05d}	seed:cpd00002

For example, we are interested in the compound *ATP* and we found it in ChEBI (with the identifier `CHEBI:30616`). We can now use the following command:

```
atp_compound = cc.get_compound("chebi:CHEBI:30616")
```

The first colon separates the namespace from the ID, while the second one is just part of the identifier itself. Not all databases will have two colons, for example

```
atp_compound = cc.get_compound("kegg:C00002")
```

Now that we created a compound, we can print some of its properties:

```
print(f"InChI = {atp_compound.inchi}")
print(f"InChIKey = {atp_compound.inchi_key}")
print(f"formula = {atp_compound.formula}")
print(f"molecular mass = {atp_compound.mass}")
print(f"pKas = {atp_compound.dissociation_constants}")
```

If you don't have an database accession for your compound, you can also get it using the InChI:

```
acetate_compound = cc.get_compound_by_inchi("InChI=1S/C2H4O2/c1-2(3)4/h1H3,(H,3,4)/p-1")
```

Searching for a `Compound`

If you don't have an exact identifier (or full InChI), you have two options for searching the database: by InChIKey or by common name.

To search by name, simply use the function `search_compound`. The search is approximate (using N-grams) and tries to find the best match in terms of edit-distance (similar to the *I'm Feeling Lucky* option on Google search). However, it can be unreliable for long compound names (and a bit slow for large datasets).

```
acetate_compound = cc.search_compound("acetat")
```

The other search option is based on InChIKeys. `search_compound_by_inchi_key` returns a list of all the matching compounds whose InChIKey matches the query as their prefix. For example:

```
cc.search_compound_by_inchi_key("MNQZXJOMYWMBOU")
```

returns a list of 3 compound objects, corresponding to D-glyceraldehyde, L-glyceraldehyde, and glyceraldehyde (with unspecific chirality).

Using the `parse_reaction_formula` function

Now, we can parse our first formula using the method `parse_reaction_formula`. In the following example, we use the BiGG database for the compound IDs to create the ATPase reaction:

```
atpase_reaction = cc.parse_reaction_formula(
    "bigg.metabolite:atp + bigg.metabolite:h2o = bigg.metabolite:adp + bigg.metabolite:pi
↪"
)
```

Note that the namespace `bigg.metabolite` has to be repeated before each compound identifier, and that several namespaces can be combined within the same reaction. For example:

```
atpase_reaction = cc.parse_reaction_formula(
    "chebi:CHEBI:30616 + kegg:C00001 = hmdb:HMDB01341 + seed:cpd00009"
)
```

We highly recommend that you check that the reaction is atomically balanced (conserves atoms) and charge balanced (redox neutral). We've found that it's easy to accidentally write unbalanced reactions in this format (e.g. forgetting to balance water is a common mistake) and so we always check ourselves. Calling the function `atpase_reaction.is_balanced()` returns a boolean with the answer.

Now, try to create more reactions of your own using `parse_reaction_formula` and make sure they are balanced.

Using the `search_reaction` function

Although `parse_reaction_formula` is the most common way to create reactions, there are a few alternatives that fit some specific use-cases.

One option is to use the `search_reaction` function, which works similarly to `search_compound` by choosing the best match for each of the compound names. For example:

```
formate_dehydrogenase_reaction = cc.search_reaction("formate + NAD = CO2 + NADH")
```

In this case, the resulting reaction object is correct (and the reaction is indeed chemically balanced). However, things can become more fuzzy when dealing with compounds that have many versions with similar names, such as sugars with multiple chiral centers. For instance,

```
glucose_isomerase_reaction = cc.search_reaction("beta-D-glucose = alpha-D-glucose")
```

In this case, the -D-glucose (InChIKey = WQZGKKKJIJFFOK-VFUOTHLCSA-N) is correctly identified, but the product chosen by eQuilibrator is D-glucose (InChIKey = WQZGKKKJIJFFOK-GASJEMHNSA-N) and not -D-glucose (InChIKey = WQZGKKKJIJFFOK-DVKNGEFBSA-N). These mistakes can be unpredictable and depend on the synonym lists in various chemical databases which are not always well-curated. There is absolutely no guarantee that the first hit would be the intended compound. If you must use the reaction search option, it is vital that you carefully go through the results and make sure no mistakes have been made. As demonstrated in the last example, simply checking the chemical balancing is not enough.

Using the `Reaction` constructor

The third option for creating a `Reaction` object is by first creating your own set of `Compound` objects (e.g. using `get_compound` based on one of the namespaces). For example, one can store them in a dictionary with the IDs as the keys.

The main input argument for the constructor is a dictionary with `Compound` as keys and the stoichiometric coefficients (float) as the values.

```
from equilibrator_api import Reaction
compound_ids = ["h2o", "adp", "atp", "pi"]
compound_dict = {cid : cc.get_compound(f"bigg.metabolite:{cid}") for cid in compound_ids}
atpase_reaction = Reaction({
    compound_dict["atp"]: -1,
    compound_dict["h2o"]: -1,
    compound_dict["adp"]: 1,
    compound_dict["pi"]: 1,
})
```

4.2.3 Estimating standard Gibbs energies

Once you have a `Reaction` object, you can use it to get estimates for the Gibbs energy.

Transformed standards Gibbs energy of a single reaction

Run the following command:

```
standard_dg_prime = cc.standard_dg_prime(atpase_reaction)
```

The return value is a `Measurement` object (from the `pint` package) which contains the mean estimated value and its uncertainty (in terms of standard deviation).

Transformed standard Gibbs energies of multiple reactions

Often when working with metabolic models, we need to work with more than one reaction at a time. Although running `standard_dg_prime()` on each reaction is possible, this means that we ignore any data about the dependencies between the estimates. In reality, errors in reaction Gibbs energies are highly correlated, due to the fact that they often have metabolites in common. Furthermore, the Component Contribution method uses sub-structures to estimate Gs and these are even more coupled (for example, consider two transaminase reactions where the vector of group changes is identical, even if the reactants themselves are different).

As an example, we start by defining three additional reactions:

```
shikimate_kinase_reaction = cc.parse_reaction_formula(
    "bigg.metabolite:skm + bigg.metabolite:atp = bigg.metabolite:skm5p + bigg.metabolite:
↪adp"
)
tyramine_dehydrogenase_reaction = cc.parse_reaction_formula(
    "kegg:C00483 + bigg.metabolite:nad + bigg.metabolite:h2o = kegg:C04227 + bigg.
↪metabolite:nadh"
)
fluorene_dehydrogenase_reaction = cc.parse_reaction_formula(
```

(continues on next page)

(continued from previous page)

```
    "kegg:C07715 + bigg.metabolite:nad + bigg.metabolite:h2o = kegg:C06711 + bigg.
↪metabolite:nadh"
)
reactions = [atpase_reaction, shikimate_kinase_reaction, tyramine_dehydrogenase_reaction,
↪ fluorene_dehydrogenase_reaction]

for r in reactions:
    assert r.is_balanced(), "One of the reactions is not chemically balanced"
```

Now that we store all our reactions in a list, we can use the `standard_dg_prime_multi` function, which is a batch version of the `standard_dg_prime` function:

```
(
    standard_dg_prime, dg_uncertainty
) = cc.standard_dg_prime_multi(reactions, uncertainty_representation="cov")
```

Since the software calculates the full covariance matrix for the uncertinaty of the estimates, we cannot use the `Measurement` class (which stores only one standard deviation per estimate). Therefore, there are two return values:

- `standard_dg_prime` (Quantity) - an array containing the CC estimation of the reactions' standard transformed energies
- `dg_uncertainty` (Quantity) - the uncertainty covariance matrix, which can be given in 4 different formats (determined by the `uncertinaty_representation` flag):
  - `cov` - the full covariance matrix
  - `precision` - precision matrix (i.e. the inverse of the covariance matrix)
  - `sqrt` - a sqaure root of the covariance, based on the uncertainty vectors
  - `fullrank` - full-rank square root of the covariance which is a compressed form of the sqrt result

In the above example, we used the `cov` option, which means that `dg_uncertainty` stores the full covariance matrix:

	ATPase	Shikimate kinase	Tyramine dehydrogenase	Fluorene dehydrogenase
ATPase	0.09	0.02	0.05	0.05
Shikimate kinase	0.02	8.07	-0.42	-0.69
Tyramine dehydrogenase	0.05	-0.42	0.73	0.68
Fluorene dehydrogenase	0.05	-0.69	0.68	2.24

All values are in units of kJ^2/mol^2. The diagonal values are the uncertainties of single reactions (the square of the standard deviation), while the off-diagonal values represent the covariances between reactions. For a detailed explanation of the importance of these values and how to use them in sampling and optimization problems, see *Covariance*.

Multi-compartment reactions

Reactions that involve membrane transport are a bit more complicated than single-compartment reactions. First, we need to indicate the location of each reactant (i.e. in which of the two compartments it is present). In addition, we need to provide extra information, such as the electrostatic potential, the pH/I/pMg on both sides of the membrane, and the amount of protons that are being transported together with the rest of the compounds.

The calculations are based on Haraldsdottir et al.[7], specitically by adding the following term to the calculated value of $\Delta G'$

$$-N_H \cdot RT \ln\left(10^{\Delta pH}\right) - Q \cdot F \Delta\Phi$$

where *R* is the gas constant, *T* is the temperature (in Kalvin), *F* is Faraday's constant (the total electric charge of one mol of electrons – 96.5 kC/mol), *pH* is the difference in pH between initial and final compartment, N_H is the net number of hydrogen ions transported from initial to final compartment, and *Q* is the stoichiometric coefficient of the transported charges.

This code example below shows how to estimate $\Delta_r G'^{\circ}$ for glucose uptake through the phosphotransferase system. The result is -44.8 $\pm$ 0.6 kJ/mol. Note that we only account for uncertainty stemming from the component-contribution method. All other estimates (e.g. based on electrostatic forces) are assumed to be precise:

```
from equilibrator_api import ComponentContribution, Q_

cytoplasmic_p_h = Q_(7.5)
cytoplasmic_ionic_strength = Q_("250 mM")
periplasmic_p_h = Q_(7.0)
periplasmic_ionic_strength = Q_("200 mM")
e_potential_difference = Q_("0.15 V")
cytoplasmic_reaction = "bigg.metabolite:pep = bigg.metabolite:g6p + bigg.metabolite:pyr"
periplasmic_reaction = "bigg.metabolite:glc__D = "

cc = ComponentContribution()
cc.p_h = cytoplasmic_p_h
cc.ionic_strength = cytoplasmic_ionic_strength
standard_dg_prime = cc.multicompartmental_standard_dg_prime(
    cc.parse_reaction_formula(cytoplasmic_reaction),
    cc.parse_reaction_formula(periplasmic_reaction),
    e_potential_difference=e_potential_difference,
    p_h_outer=periplasmic_p_h,
    ionic_strength_outer=periplasmic_ionic_strength,
)

print(standard_dg_prime)
```

[7] H. S. Haraldsdóttir, I. Thiele, and R. M. T. Fleming. Quantitative Assignment of Reaction Directionality in a Multicompartmental Human Metabolic Reconstruction. *Biophysical Journal*, 102(8):1703–1711, April 2012. URL: http://www.sciencedirect.com/science/article/pii/S0006349512002639 (visited on 2018-07-27), doi:10.1016/j.bpj.2012.02.032.

4.2.4 References

4.3 Covariance

The uncertainties of the free energies estimated with eQuilibrator are often correlated. Sometimes we have moieties whose formation energy has high uncertainty (such as coenzyme-A, Figure 1A), but this uncertainty cancels out in reactions where the moiety is present on both sides. In contrast, there are cases where reaction energies cannot be determined because of completely uncharacterized compounds (such as flavodoxin, Figure 1B), but using explicit formation energies reveals couplings between multiple reactions. Thus, it is often unclear whether one should use the domain of reaction energies or of formation energies. With eQuilibrator 3.0, we encourage the usage of the covariance matrix of the uncertainty when modeling multiple reactions. This matrix fully captures the correlations in the uncertainty of all quantities, and always constrains the values at least as much as when using independent uncertainties. In Figure 1, only using the covariance matrix allows to determine reaction directions in both examples. Importantly, the covariance can be used in the domains of formation as well as reaction energies without loss of information on the reaction energies. In this section we summarize how the convariance matrix can be used in sampling and constraint based methods.

Fig. 1: **Figure 1**: Examples for the importance of the covariance in the estimation uncertainty found in the iML1515 *E. coli* model. **(A)** Homoserine O-succinyltransferase (HSST) and 3-oxoadipyl-CoA thiolase (3OXCOAT) both convert succinyl-CoA (succoa) to CoA. Because of the uncertainty in the $\Delta_f G'^{\circ}$ of CoA, computing reaction energies from independent $\Delta_f G'^{\circ}$ estimates results in a large uncertainty (orange). As the $\Delta_f G'^{\circ}$ of succoa and CoA are strongly correlated, direct estimates of $\Delta_r G'^{\circ}$ have smaller uncertainty (blue) comparable to the uncertainty obtained using the covariance matrix of either $\Delta_f G'^{\circ}$ or $\Delta_r G'^{\circ}$ (cyan). **(B)** Pyruvate synthase (POR5) converts acetyl-CoA (accoa) into pyruvate (pyr) by oxidizing flavodoxin which, in iML1515, can be regenerated only through oxidation of NADPH (FLDR2). The $\Delta_r G'^{\circ}$ of both reactions is unknown. However, $\Delta_f G'^{\circ}$ of flxr and flxso must have the same value in both reactions, leading to strong correlation in the uncertainty of $\Delta_r G'^{\circ}$. Thus, textit{in-vivo} synthesis of pyruvate through POR5 is unfavorable. **(C)** Covariances of $\Delta_f G'^{\circ}$ and $\Delta_r G'^{\circ}$ yield the same information about reaction energies. The small uncertainty in $\Delta_r G'^{\circ}$ for HSST and 3OXCOAT matches the correlation in $\Delta_f G'^{\circ}$ of succoa and coa, while the coupling of FLDR2 and POR5 through flavodoxin is captured by the covariance in the $\Delta_r G'^{\circ}$ of the two reactions.

4.3.1 Random Sampling

Consider a reaction network with stoichiometric matrix $\bar{X}$. The number of degrees of freedom $\bar{q}$ in the uncertainty is often smaller than the number of reactions n (note that $\bar{q} \leq q = 669$). Thus, it is convenient to represent the uncertainty with a random vector $\mathbf{m} \in \mathbb{R}^{\bar{q}}$ following the standard normal distribution $\mathcal{N}(0, I)$ and a square root $Q(\bar{X}) \in \mathbb{R}^{n \times \bar{q}}$ of the covariance $\Sigma(\bar{X})$[1], such that

$$\begin{aligned} \mathbf{m} &\sim \mathcal{N}(0, I) \\ \Delta_r G'^{\circ} &= \Delta_r G'^{\circ}(\bar{X}) + Q(\bar{X})\mathbf{m} \end{aligned}$$

where I is the $\bar{q}$-dimensional identity matrix. While $Q(\bar{X})$ can be computed from the eigenvalue decomposition of $\Sigma(\bar{X})$, this is sensitive to numerical issues if $\bar{X}$ is large. Instead, eQuilibrator computes $Q(\bar{X})$ directly, providing a numerically accurate result.

In order to draw random samples of the Gibbs free energies we can first draw samples of $\mathbf{m}$ using standard methods and then compute the corresponding free energies using the above equations.

See *Random sampling* for a code example.

[1] Mattia G Gollub, Hans-Michael Kaltenbach, and Jörg Stelling. Probabilistic Thermodynamic Analysis of Metabolic Networks. *Bioinformatics*, March 2021. URL: https://doi.org/10.1093/bioinformatics/btab194 (visited on 2021-07-27), doi:10.1093/bioinformatics/btab194.

4.3.2 Constraint-based models

In a constraint-based setting, we can use the same formulation to define a quadratic constraint to bound free energies to a desired confidence level α:

$$||\mathbf{m}||_2^2 \leq \chi^2_{\bar{q};\alpha}$$
$$\Delta_r G'^{\circ} = \Delta_r G'^{\circ}(\bar{X}) + Q(\bar{X})\mathbf{m}$$

where $\chi^2_{\bar{q};\alpha}$ is the PPF (percent point function, or quantile function) of the χ^2-distribution with $\bar{q}$ degrees of freedom. In Python it can be calculated using `scipy.stats.chi2.ppf()`.

When quadratic constraints cannot be used, one can replace the first inequality with upper and lower bounds for each m_i separately, corresponding to a confidence interval α on each individual degree of freedom in the uncertainty:

$$|m_i| \leq \sqrt{\chi^2_{1;\alpha}} \qquad \forall\ 1 \leq i \leq \bar{q}$$

Although simpler, this formulation should be used with care. Uncertainties are multivariate estimates and independent bounds can over-constrain the free energies, in particular for large networks. For example, when $\bar{q}$ = 50 and α = 0.95, the bounds define a confidence region on $\mathbf{m}$ with an overly restrictive confidence level $\alpha^{\bar{q}}$ = 0.08.

See *Constraint-based modeling* for a code example.

4.3.3 References

4.4 Code examples

We start by importing the necessary packages

[1]:
```
import numpy
import cvxpy
import scipy.stats
import matplotlib.pyplot as plt
from equilibrator_api import ComponentContribution, Q_
from numpyarray_to_latex.jupyter import to_jup
```

4.4.1 Basic *G'* calculations

Create an instance of `ComponentContribution`, which is the main interface for using eQuilibrator. This command loads all the data that is required to calculate thermodynamic potentials of reactions, and it is normal for it to take 10-20 seconds to execute. If you are running it for the first time on a new computer, it will download a 1.3 GB file, which might take a few minutes or up to an hour (depending on your bandwidth). Don't worry, it will only happen once.

[2]:
```
cc = ComponentContribution()

# optional: changing the aqueous environment parameters
cc.p_h = Q_(7.4)
cc.p_mg = Q_(3.0)
cc.ionic_strength = Q_("0.25M")
cc.temperature = Q_("298.15K")
```

You can parse a reaction formula that uses compound accessions from different databases (KEGG, ChEBI, MetaNetX, BiGG, and a few others). The following example is ATP hydrolysis to ADP and inorganic phosphate, using BiGG metabolite IDs:

[3]:
```
atpase_reaction = cc.parse_reaction_formula(
    "bigg.metabolite:atp + bigg.metabolite:h2o = "
    "bigg.metabolite:adp + bigg.metabolite:pi"
)
```

We highly recommend that you check that the reaction is atomically balanced (conserves atoms) and charge balanced (redox neutral). We've found that it's easy to accidentally write unbalanced reactions in this format (e.g. forgetting to balance water is a common mistake) and so we always check ourselves.

[4]:
```
print("The reaction is " + ("" if atpase_reaction.is_balanced() else "not ") + "balanced
↪")
```

```
The reaction is balanced
```

Now we know that the reaction is "kosher" and we can safely proceed to calculate the standard change in Gibbs potential due to this reaction.

[5]:
```
dG0_prime = cc.standard_dg_prime(atpase_reaction)
print(f"G'° = {dG0_prime}")

dGm_prime = cc.physiological_dg_prime(atpase_reaction)
print(f"G'm = {dGm_prime}")

atpase_reaction.set_abundance(cc.get_compound("bigg.metabolite:atp"), Q_("1 mM"))
atpase_reaction.set_abundance(cc.get_compound("bigg.metabolite:adp"), Q_("100 uM"))
atpase_reaction.set_abundance(cc.get_compound("bigg.metabolite:pi"), Q_("0.003 M"))

dG_prime = cc.dg_prime(atpase_reaction)
print(f"G' = {dG_prime}")
```

```
G'° = (-29.14 +/- 0.30) kilojoule / mole
G'm = (-46.26 +/- 0.30) kilojoule / mole
G' = (-49.24 +/- 0.30) kilojoule / mole
```

The return values are `pint.Measurement` objects. If you want to extract the Gibbs energy value and error as floats, you can use the following commands:

[6]:
```
dG_prime_value_in_kj_per_mol = dG_prime.value.m_as("kJ/mol")
dG_prime_error_in_kj_per_mol = dG_prime.error.m_as("kJ/mol")
print(
    f"G'° = {dG_prime_value_in_kj_per_mol:.1f} +/- "
    f"{dG_prime_error_in_kj_per_mol:.1f} kJ/mol"
)
```

```
G'° = -49.2 +/- 0.3 kJ/mol
```

4.4.2 The reversibility index

You can also calculate the reversibility index for this reaction.

[7]:
```
print(f"ln(Reversibility Index) = {cc.ln_reversibility_index(atpase_reaction)}")
```

```
ln(Reversibility Index) = (-12.45 +/- 0.08) dimensionless
```

The reversibility index is a measure of the degree of the reversibility of the reaction that is normalized for stoichiometry. If you are interested in assigning reversibility to reactions we recommend this measure because 1:2 reactions are much "easier" to reverse than reactions with 1:1 or 2:2 reactions. You can see our paper for more information.

4.4.3 Further examples for reaction parsing

Parsing reaction with non-trivial stoichiometric coefficients is simple. Just add the coefficients before each compound ID (if none is given, it is assumed to be 1)

[8]:
```
rubisco_reaction = cc.parse_reaction_formula(
    "bigg.metabolite:rb15bp + bigg.metabolite:co2 + bigg.metabolite:h2o = 2 bigg.
↪metabolite:3pg"
)
dG0_prime = cc.standard_dg_prime(rubisco_reaction)
print(f"G'° = {dG0_prime}")
```

```
G'° = (-31 +/- 4) kilojoule / mole
```

We support several compound databases, not just BiGG. One can mix between several sources in the same reaction, e.g.:

[9]:
```
atpase_reaction = cc.parse_reaction_formula(
    "bigg.metabolite:atp + CHEBI:15377 = metanetx.chemical:MNXM7 + bigg.metabolite:pi"
)
dG0_prime = cc.standard_dg_prime(atpase_reaction)
print(f"G'° = {dG0_prime}")
```

```
G'° = (-29.14 +/- 0.30) kilojoule / mole
```

Or, you can use compound names instead of identifiers. However, it is discouraged to use in batch, since we only pick the closest hit in our database, and that can often be the wrong compound. Always verify that the reaction is balanced, and preferably also that the InChIKeys are correct:

[10]:
```
glucose_isomerase_reaction = cc.search_reaction("beta-D-glucose = glucose")
for cpd, coeff in glucose_isomerase_reaction.items():
    print(f"{coeff:5.0f}  {cpd.get_common_name():15s} {cpd.inchi_key}")
dG0_prime = cc.standard_dg_prime(glucose_isomerase_reaction)
print(f"G'° = {dG0_prime}")
```

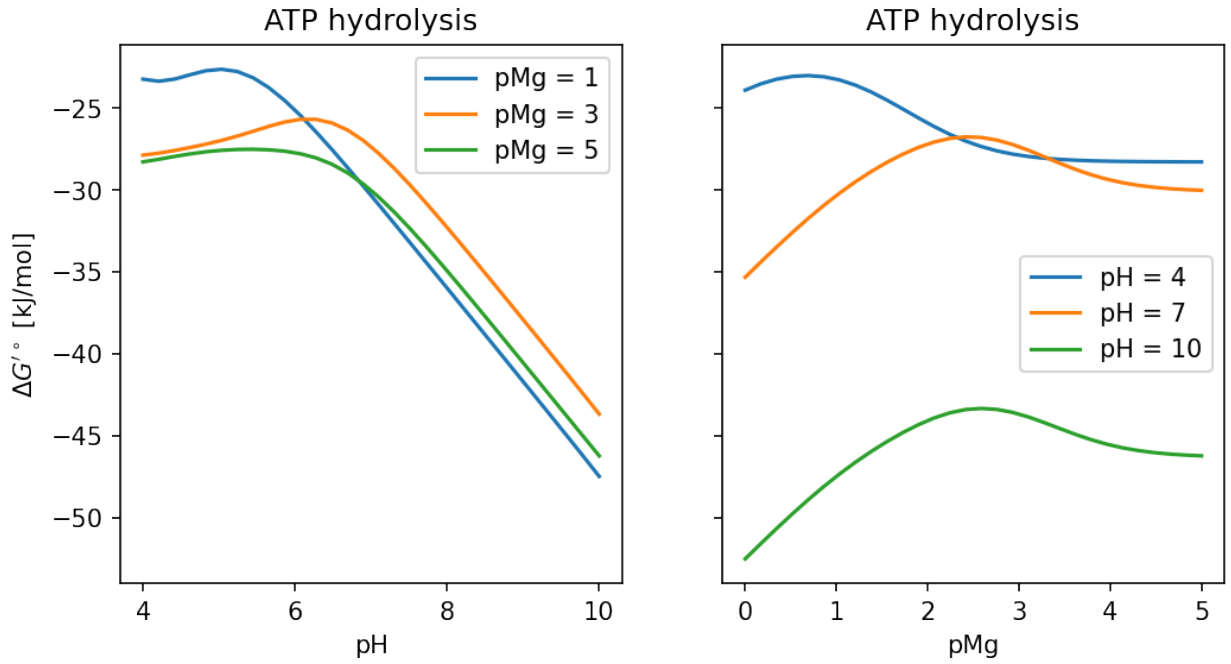
```
   -1  BETA-D-GLUCOSE  WQZGKKKJIJFFOK-VFUOTHLCSA-N
    1  D-Glucose       WQZGKKKJIJFFOK-GASJEMHNSA-N
G'° = (-1.6 +/- 1.3) kilojoule / mole
```

In this case, the matcher arbitrarily chooses α-D-glucose as the first hit for the name `glucose`. Therefore, it is always better to use the most specific synonym to avoid mis-annotations.

4.4.4 *G'°* of ATP hydrolysis

as a function of pH and pMg

[11]:
```
def calc_dg(p_h, p_mg):
    cc.p_h = Q_(p_h)
    cc.p_mg = Q_(p_mg)
    return cc.standard_dg_prime(atpase_reaction).value.m_as("kJ/mol")

fig, axs = plt.subplots(1, 2, figsize=(8, 4), dpi=150, sharey=True)

ax = axs[0]
ph_range = numpy.linspace(4, 10, 30)
ax.plot(ph_range, [calc_dg(p_h, 1) for p_h in ph_range], '-', label="pMg = 1")
ax.plot(ph_range, [calc_dg(p_h, 3) for p_h in ph_range], '-', label="pMg = 3")
ax.plot(ph_range, [calc_dg(p_h, 5) for p_h in ph_range], '-', label="pMg = 5")
ax.set_xlabel('pH')
ax.set_ylabel(r"$\Delta G'^\circ$ [kJ/mol]")
ax.set_title("ATP hydrolysis")
ax.legend();

ax = axs[1]
pmg_range = numpy.linspace(0, 5, 30)
ax.plot(pmg_range, [calc_dg(4, p_mg) for p_mg in pmg_range], '-', label="pH = 4")
ax.plot(pmg_range, [calc_dg(7, p_mg) for p_mg in pmg_range], '-', label="pH = 7")
ax.plot(pmg_range, [calc_dg(10, p_mg) for p_mg in pmg_range], '-', label="pH = 10")
ax.set_xlabel('pMg')
ax.set_title("ATP hydrolysis")
ax.legend();

# don't forget to change the aqueous conditions back to the default ones
cc.p_h = Q_(7.4)
cc.p_mg = Q_(3.0)
cc.ionic_strength = Q_("0.25M")
cc.temperature = Q_("298.15K")
```



4.4.5 Multivariate calculations

We start by defining a toy model consisting of two reactions: a GTP hydrolysis reaction (NTP3) and adenylate kinase with GTP (ADK3)

[12]:
```
NTP3 = cc.parse_reaction_formula(
    "bigg.metabolite:gtp + bigg.metabolite:h2o = bigg.metabolite:gdp + bigg.metabolite:pi
↪"
)
ADK3 = cc.parse_reaction_formula(
    "bigg.metabolite:adp + bigg.metabolite:gdp = bigg.metabolite:amp + bigg.metabolite:
↪gtp"
)

reactions = [NTP3, ADK3]
```

Now we use `standard_dg_prime_multi` to obtain the mean and covariance matrix of the standard G' values. The covariance is represented by the full-rank square root $\mathbf{Q}$ (i.e. such that the covariance is given by $\mathbf{Q}\mathbf{Q}^\top$)

[13]:
```
standard_dgr_prime_mean, standard_dgr_Q = cc.standard_dg_prime_multi(
    reactions,
    uncertainty_representation="fullrank"
)

print(f"mean () in kJ / mol:")
to_jup(standard_dgr_prime_mean.m_as("kJ/mol"))
print(f"square root (Q) in kJ / mol:")
to_jup(standard_dgr_Q.m_as("kJ/mol"))
print(f"covariance in (kJ / mol)^2:")
to_jup((standard_dgr_Q @ standard_dgr_Q.T).m_as("kJ**2/mol**2"))
```

```
mean () in kJ / mol:
```

$$\begin{pmatrix} -26.46 \\ -2.95 \end{pmatrix}$$

```
square root (Q) in kJ / mol:
```

$$\begin{pmatrix} -1.35 \\ 0.00 \\ 1.30 \\ -0.31 \end{pmatrix}$$

```
covariance in (kJ / mol)^2:
```

$$\begin{pmatrix} 1.83 \\ -1.76 \\ -1.76 \\ 1.78 \end{pmatrix}$$

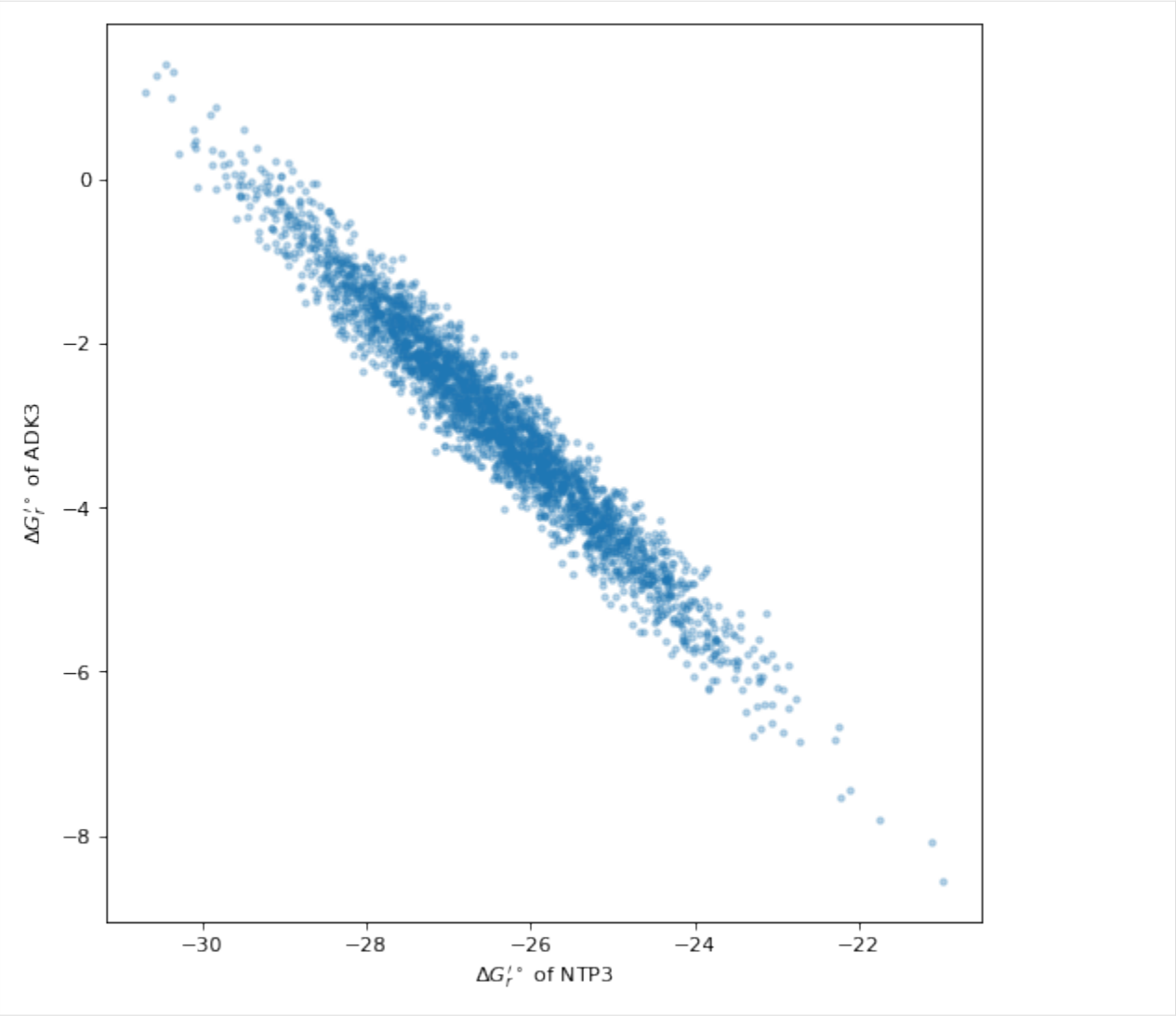
Random sampling

The following example demonstrates how one can sample from the multivariate distribution of the standard G' by drawing random normal samples $\mathbf{m}$ and using μ and $\mathbf{Q}$.

$\mathbf{m} \sim \mathcal{N}(0, \mathbf{I})$

$\Delta_r G'^{\circ} = \mu + \mathbf{Q}\,\mathbf{m}$

[14]:
```
numpy.random.seed(2019)
N = 3000
sampled_data = []
for i in range(N):
    m = numpy.random.randn(standard_dgr_Q.shape[1])
    standard_dgr_prime_sample = standard_dgr_prime_mean + standard_dgr_Q @ m
    sampled_data.append(standard_dgr_prime_sample.m_as("kJ/mol"))
sampled_data = numpy.array(sampled_data)

fig, ax = plt.subplots(1, 1, figsize=(8, 8), dpi=80)
ax.plot(sampled_data[:, 0], sampled_data[:, 1], '.', alpha=0.3)
ax.set_xlabel("$\Delta G_r'^\circ$ of NTP3")
ax.set_ylabel("$\Delta G_r'^\circ$ of ADK3");
ax.set_aspect("equal")
```



Constraint-based modeling

In this example, we use μ and $\mathbf{Q}$ in a quadratic constraints linear problem to find the metabolite concentrations that possible range of the total driving force (for both NTP3 and ADK3 combined).

$$\begin{aligned}
\text{maximize/minimize} \quad & \mathbf{g}^\top \cdot \mathbf{1} && (4.1)\\
\text{such that} \quad & && (4.2)\\
& \ln(10^{-4}) \leq x_i \leq \ln(10^{-2}) && (4.3)\\
& \mathbf{x} \leq \ln(10^{-2}) && (4.4)\\
& ||\mathbf{m}||_2 \leq \chi^2_{\alpha,q} && (4.5)\\
& \mathbf{g} = -\mu - RT \cdot \mathbf{S}^\top \mathbf{x} - \mathbf{Q}\mathbf{m} && (4.6)
\end{aligned}$$

where the desired confidence is α = 0.95, $\mathbf{x}$ represents the vector of metabolite log concentrations, and $\mathbf{g}$ represents vector of reaction driving forces.

[15]:
```
S = cc.create_stoichiometric_matrix(reactions)

alpha = 0.95
Nc, Nr = S.shape
Nq = standard_dgr_Q.shape[1]
lb = 1e-4
ub = 1e-2

ln_conc = cvxpy.Variable(shape=Nc, name="metabolite log concentration")
m = cvxpy.Variable(shape=Nq, name="covariance degrees of freedom")

constraints = [
    numpy.log(numpy.ones(Nc) * lb) <= ln_conc,  # lower bound on concentrations
    ln_conc <= numpy.log(numpy.ones(Nc) * ub),  # upper bound on concentrations
    cvxpy.norm2(m) <= scipy.stats.chi2.ppf(alpha, Nq) ** (0.5)  # quadratic bound on m␣
↪based on confidence interval
]

dg_prime = -(
    standard_dgr_prime_mean.m_as("kJ/mol") +
    cc.RT.m_as("kJ/mol") * S.values.T @ ln_conc +
    standard_dgr_Q.m_as("kJ/mol") @ m
)

prob_max = cvxpy.Problem(cvxpy.Maximize(dg_prime @ numpy.ones(Nr)), constraints)
prob_max.solve()
max_df = prob_max.value

prob_min = cvxpy.Problem(cvxpy.Minimize(dg_prime @ numpy.ones(Nr)), constraints)
prob_min.solve()
min_df = prob_min.value

print(f"* Possible Range of total driving force for NTP3 + ADK3: {min_df:.1f} - {max_df:.
↪1f} kJ/mol")
```

```
* Possible Range of total driving force for NTP3 + ADK3: 5.8 - 53.0 kJ/mol
```

4.4.6 Using formation energies to calculate reaction energies

Warning: using formation energies is highly discouraged

Avoid using formation energies, unless it is absolutely necessary. The function `standard_dg_formation` is difficult to use by design and requires good knowledge of thermodynamics. Please use `standard_dg_prime` instead whenever possible.

In the following example, we consider two reactions that share several reactants: ATPase and adenylate kinase

[16]:
```
metabolite_names = ["atp", "adp", "amp", "pi", "h2o"]
print("order of compounds: " + str(metabolite_names) + "\n")

# obtain a list of compound objects using `get_compound`
```

(continues on next page)

(continued from previous page)

```
compound_list = [cc.get_compound(f"bigg.metabolite:{cname}") for cname in metabolite_
↪names]

# appply standard_dg_formation on each one, and pool the results in 3 lists
standard_dgf_mu, sigmas_fin, sigmas_inf = zip(*map(cc.standard_dg_formation, compound_
↪list))
standard_dgf_mu = numpy.array(standard_dgf_mu)
sigmas_fin = numpy.array(sigmas_fin)
sigmas_inf = numpy.array(sigmas_inf)

# we now apply the Legendre transform to convert from the standard Gf to the standard G'f
delta_dgf_list = numpy.array([
    cpd.transform(cc.p_h, cc.ionic_strength, cc.temperature, cc.p_mg).m_as("kJ/mol")
    for cpd in compound_list
])
standard_dgf_prime_mu = standard_dgf_mu + delta_dgf_list

# to create the formation energy covariance matrix, we need to combine the two outputs
# sigma_fin and sigma_inf
standard_dgf_cov = sigmas_fin @ sigmas_fin.T + 1e6 * sigmas_inf @ sigmas_inf.T

print("(Gf'0) in kJ / mol:")
to_jup(standard_dgf_prime_mu)
print("(Gf'0) in kJ^2 / mol^2:")
to_jup(standard_dgf_cov)
```

```
order of compounds: ['atp', 'adp', 'amp', 'pi', 'h2o']

(Gf'0) in kJ / mol:
```

$$\begin{pmatrix} -2270.48 \\ -1395.63 \\ -521.04 \\ -1056.08 \\ -152.08 \end{pmatrix}$$

```
(Gf'0) in kJ^2 / mol^2:
```

$$\begin{pmatrix} 2.22 \\ 1.70 \\ 1.17 \\ 0.26 \\ -0.24 \\ 1.70 \\ 1.50 \\ 1.30 \\ 0.12 \\ -0.09 \\ 1.17 \\ 1.30 \\ 1.43 \\ -0.02 \\ 0.07 \\ 0.26 \\ 0.12 \\ -0.02 \\ 0.56 \\ 0.40 \\ -0.24 \\ -0.09 \\ 0.07 \\ 0.40 \\ 0.58 \end{pmatrix}$$

Now we define S and use it to calculate the reaction G'0 values

[17]:
```
S = numpy.array([
    [-1, 1, 0, 1, -1],  # ATPase
    [-1, 2, -1, 0, 0]   # adenylate kinase
], dtype=int).T
print("stoichiometric matrix:")
to_jup(S, fmt="{:d}")

standard_dgr_prime_mu = S.T @ standard_dgf_prime_mu
standard_dgr_cov = S.T @ standard_dgf_cov @ S
print(f"(Gr'0) in kJ / mol:")
to_jup(standard_dgr_prime_mu, fmt="{:.1f}")
print(f"(Gr'0) in kJ^2 / mol^2:")
to_jup(standard_dgr_cov, fmt="{:.3f}")
```

```
stoichiometric matrix:
```

$$\begin{pmatrix} -1 \\ -1 \\ 1 \\ 2 \\ 0 \\ -1 \\ 1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

```
(Gr'0) in kJ / mol:
```

$$\begin{pmatrix} -29.1 \\ 0.3 \end{pmatrix}$$

```
(Gr'0) in kJ^2 / mol^2:
```

$$\begin{pmatrix} 0.093 \\ 0.010 \\ 0.010 \\ 0.026 \end{pmatrix}$$

Compare this result to the output of `standard_dg_multi` and confirm that they are the same

[18]:
```
ADK = cc.parse_reaction_formula(
    "bigg.metabolite:atp + bigg.metabolite:amp = 2 bigg.metabolite:adp"
)
standard_dgr_prime_mu2, standard_dgr_cov2 = cc.standard_dg_prime_multi([atpase_reaction,␣
↪ADK])

print(f"(Gr'0) in kJ / mol:")
to_jup(standard_dgr_prime_mu2.m_as("kJ/mol"), fmt="{:.1f}")
print(f"(Gr'0) in kJ^2 / mol^2:")
to_jup(standard_dgr_cov2.m_as("kJ**2/mol**2"), fmt="{:.3f}")
```

```
(Gr'0) in kJ / mol:
```

$$\begin{pmatrix} -29.1 \\ 0.3 \end{pmatrix}$$

```
(Gr'0) in kJ^2 / mol^2:
```

$$\begin{pmatrix} 0.093 \\ 0.010 \\ 0.010 \\ 0.026 \end{pmatrix}$$

4.4.7 Estimating energies for multi-compartment reactions

Our first example is for showing how to use `multicompartmental_standard_dg_prime` for a single reaction.

[27]:
```
cc.p_h = Q_(7.4)
cc.p_mg = Q_(3.0)
cc.ionic_strength = Q_("0.25M")
cc.temperature = Q_("298.15K")

periplasmic_p_h = Q_(6.5)
periplasmic_ionic_strength = Q_("200 mM")
periplasmic_p_mg = Q_(3.0)
e_potential_difference = Q_(0.17, "V")

cytoplasmic_reaction = f"bigg.metabolite:adp + bigg.metabolite:pi + 3 bigg.metabolite:h␣
↪= bigg.metabolite:h2o + bigg.metabolite:atp"
periplasmic_reaction = f"= 3 bigg.metabolite:h"
standard_dg_prime = cc.multicompartmental_standard_dg_prime(
```

(continues on next page)

(continued from previous page)

```
    reaction_inner=cc.parse_reaction_formula(cytoplasmic_reaction),
    reaction_outer=cc.parse_reaction_formula(periplasmic_reaction),
    e_potential_difference=e_potential_difference,
    p_h_outer=periplasmic_p_h,
    ionic_strength_outer=periplasmic_ionic_strength,
    p_mg_outer=periplasmic_p_mg
)
print("G' = ", standard_dg_prime)
```

```
G' =  (-4.66 +/- 0.30) kilojoule / mole
```

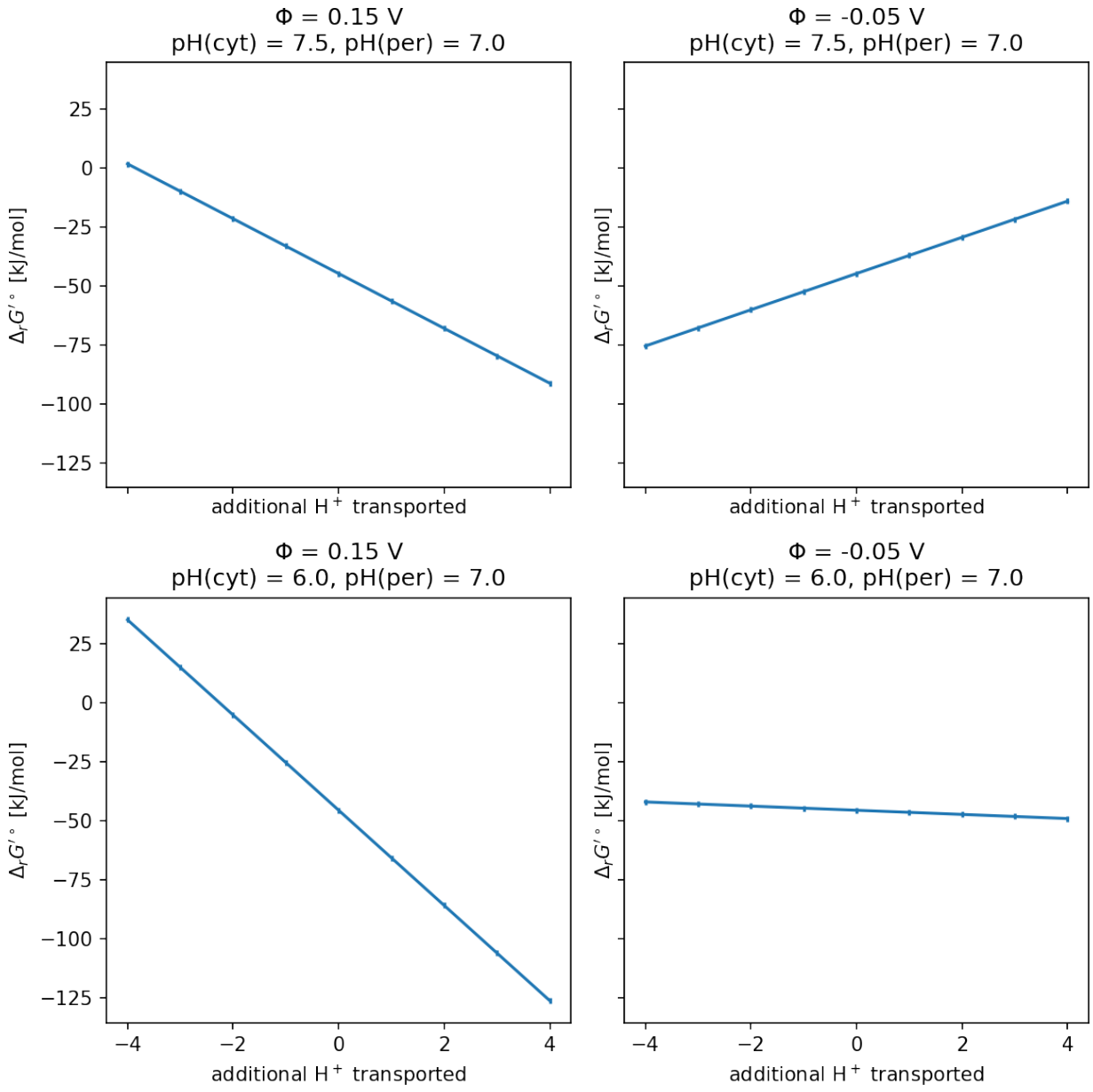
The following example demonstrated how to use `multi_compartmental_standard_dg_prime` for estimating the standard G' of the phosphotransferase (PTS) system which imports glucose into the cell while phosphorylating it using PEP. We compare the response to the number of extra transported protons, in several physiological conditions.

[20]:
```
fig, axs = plt.subplots(2, 2, figsize=(8, 8), dpi=150, sharey=True, sharex=True)

PMF_RANGE = 4
for (psi, ph), ax in zip([(0.15, 7.5), (-0.05, 7.5), (0.15, 6.0), (-0.05, 6.0)], axs.
↪flat):
    e_potential_difference = Q_(psi, "V")

    cytoplasmic_p_h = Q_(ph)
    cytoplasmic_ionic_strength = Q_("250 mM")
    cytoplasmic_p_mg = Q_(3.0)

    periplasmic_p_h = Q_(7.0)
    periplasmic_ionic_strength = Q_("200 mM")
    periplasmic_p_mg = Q_(3.0)

    data = []
    for n_pmf in numpy.arange(-PMF_RANGE, PMF_RANGE+1):

        cytoplasmic_reaction = f"bigg.metabolite:pep + {n_pmf} bigg.metabolite:h = bigg.
↪metabolite:g6p + bigg.metabolite:pyr"
        periplasmic_reaction = f"bigg.metabolite:glc__D = {n_pmf} bigg.metabolite:h"

        cc.p_h = cytoplasmic_p_h
        cc.ionic_strength = cytoplasmic_ionic_strength
        cc.p_mg = cytoplasmic_p_mg

        standard_dg_prime = cc.multicompartmental_standard_dg_prime(
            reaction_inner=cc.parse_reaction_formula(cytoplasmic_reaction),
            reaction_outer=cc.parse_reaction_formula(periplasmic_reaction),
            e_potential_difference=e_potential_difference,
            p_h_outer=periplasmic_p_h,
            ionic_strength_outer=periplasmic_ionic_strength,
            p_mg_outer=periplasmic_p_mg
        )
        data.append((n_pmf, standard_dg_prime.value.m_as("kJ/mol"), 1.96 * standard_dg_
↪prime.error.m_as("kJ/mol")))

    pmf, dg, dg_conf_interval = zip(*data)
```

(continues on next page)

(continued from previous page)

```
    ax.errorbar(pmf, dg, yerr=dg_conf_interval)
    ax.set_title(f"$\Phi$ = {psi} V\npH(cyt) = {cytoplasmic_p_h.m_as('')}, pH(per) =
↪{periplasmic_p_h.m_as('')}")
    ax.set_xlabel("additional H$^+$ transported")
    ax.set_ylabel("$\Delta_r G'^\circ$ [kJ/mol]")
fig.tight_layout()
```



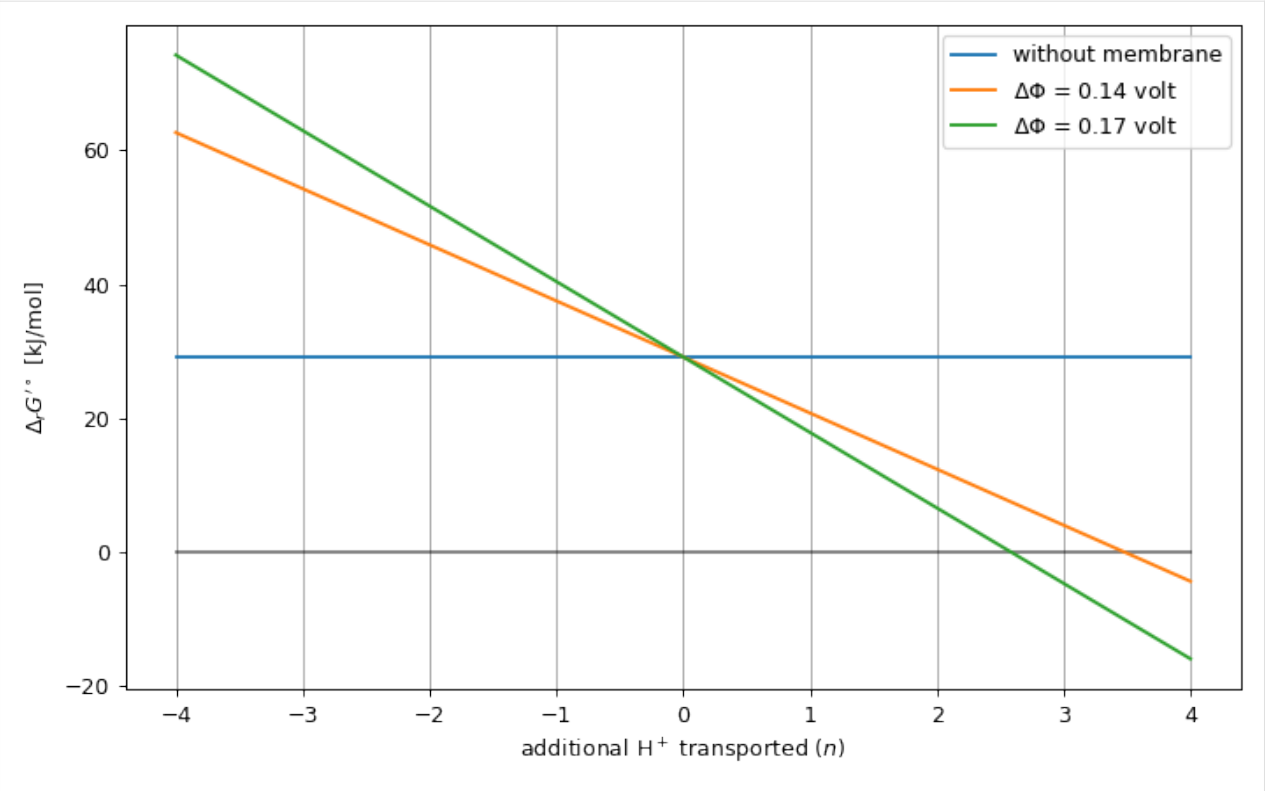
In the next example, we estimate the standard G' of ATP synthase as a function of the number of extra transported protons, in either = 0.14V and = 0.17V.

[21]:
```
data_series = []
PMF_RANGE = 4
```

(continues on next page)

(continued from previous page)

```
cc.p_h = Q_(7.4) # set the cytoplasmic pH
cc.ionic_strength = Q_("250 mM") # set the cytoplasmic I
cc.p_mg = Q_(3.0) # set the cytoplasmic pMg
periplasmic_p_h = Q_(6.5)
periplasmic_ionic_strength = Q_("200 mM")
periplasmic_p_mg = Q_(3.0)
for phi in [0.14, 0.17]:
    e_potential_difference = Q_(phi, "V")
    data = []
    for n_pmf in numpy.arange(-PMF_RANGE, PMF_RANGE+1):
        cytoplasmic_reaction = (
            f"bigg.metabolite:adp + bigg.metabolite:pi + {n_pmf} bigg.metabolite:h = "
            "bigg.metabolite:h2o + bigg.metabolite:atp"
        )
        periplasmic_reaction = f" = {n_pmf} bigg.metabolite:h"
        standard_dg_prime = cc.multicompartmental_standard_dg_prime(
            reaction_inner=cc.parse_reaction_formula(cytoplasmic_reaction),
            reaction_outer=cc.parse_reaction_formula(periplasmic_reaction),
            e_potential_difference=e_potential_difference,
            p_h_outer=periplasmic_p_h,
            ionic_strength_outer=periplasmic_ionic_strength,
            p_mg_outer=periplasmic_p_mg
        )
        data.append((n_pmf, standard_dg_prime.value.m_as("kJ/mol")))
    pmf, dg = zip(*data)
    data_series.append((
        e_potential_difference, cytoplasmic_p_h, periplasmic_p_h, pmf, dg
    ))
fig, ax = plt.subplots(1, 1, figsize=(8, 5), dpi=90, sharey=True, sharex=True)
dG0_prime = -cc.standard_dg_prime(atpase_reaction).value.m_as("kJ/mol")
ax.plot([-PMF_RANGE, PMF_RANGE], [dG0_prime, dG0_prime],
        label="without membrane")
ax.plot([-PMF_RANGE, PMF_RANGE], [0, 0], "k-", alpha=0.5, label=None)
for e_potential_difference, _, _, pmf, dg in data_series:
    ax.plot(
        pmf, dg,
        label=f"$\Delta\Phi$ = {e_potential_difference}"
    )
ax.set_xlabel("additional H$^+$ transported ($n$)")
ax.set_ylabel("$\Delta_r G'^\circ$ [kJ/mol]")
ax.set_xticks(numpy.arange(-PMF_RANGE, PMF_RANGE+1))
ax.axes.xaxis.grid(True)
ax.legend(loc="best")
fig.tight_layout()
fig.savefig("pmf.pdf")
```



4.5 Local cache

The LocalCompoundCache class, found in local_compound_cache, provides methods to generate Compound objects as well as storing and retrieving these compounds from a local component contribution database.

This notebook will highlight the following use-cases:

1. Adding compounds and retrieving them from the coco namespace using `add_compounds`
2. Adding compounds and retrieving them using `get_compounds`
3. Options to control behavior for `get_compounds` and `add_compounds`

4.5.1 Requirements

- equilibrator-assets: `!pip install equilibrator-assets`
- openbabel: `!pip install openbabel-wheel` or `!conda install -c conda-forge openbabel`
- chemaxon (including license): `cxcalc` must be in "PATH"

4.5.2 Initialize the local compound cache

[1]:
```
import pandas as pd
from equilibrator_assets.local_compound_cache import LocalCompoundCache
lc = LocalCompoundCache()
```

4.5.3 Generating a new local cache

A copy of the default zenodo cache must be used for the local_cache.

You can skip this cell if the local cache already exists

[2]:
```
# Copies the default zenodo compounds.sqlite cache to file location
# If that location already exists, user is prompted to delete
lc.generate_local_cache_from_default_zenodo('compounds.sqlite')
```

```
compounds.sqlite already exists.
Delete existing file and replace?
```

```
Proceed? (yes/no): yes
```

```
Deleting compounds.sqlite
Copying default Zenodo compound cache to compounds.sqlite
```

4.5.4 Loading an already existing local cache

[3]:
```
# load the local cache from the .sqlite database
lc.load_cache('compounds.sqlite')
```

```
Loading compounds from compounds.sqlite
```

4.5.5 Creating and adding compounds to the coco namespace

`add_compounds` provides a method to take a data frame consisting of compound information and generating and adding new compounds into the database. When generated, three compound properties must be defined:

1. `struct` - a SMILES string representing the compound structure
2. `coco_id` - an ID (string) enabling use with the equilibrator-api parser, e.g. `my_compound` can be accessed using `coco:my_compound`
3. `name` - the name of a compound that will appear when creating plots for analyses such as Max-min Driving Force (MDF)

To generate compounds, a DataFrame must be provided following this example:

struct	coco_id	name
CCO	etoh	Ethanol
C/C1=CC(Ø)=C/C(=O)O1	TAL	Triacetic Acid Lactone

[4]:
```
# Generating an example .csv for adding compounds
# 3A4HA is already present, but custom names can be added
# to the coco namespace
compound_df = pd.DataFrame(
    data=[
        ["OC(=O)C1=CC(NC(=O)C2=CC=CC=C2)=C(O)C=C1", "3B4HA", "3-Benzamido-4-
↪hydroxybenzoic acid"],
        ["NC1=C(O)C=CC(=C1)C(O)=O", "3A4HA", "3-Amino-4-hydroxybenzoic acid"]
    ],
    columns=["struct","coco_id", "name"]
)

lc.add_compounds(compound_df, mol_format="smiles")
# added compound has the ID 3B4HA that can be access as coco:3B4HA
# and prints as 3-Amino-4-hydroxybenzoic acid in plots
coco3B4HA = lc.get_compounds("OC(=O)C1=CC(NC(=O)C2=CC=CC=C2)=C(O)C=C1")
print(coco3B4HA)
```

```
Compound(id=694325, inchi_key=RKCVLDMDZASBEO-UHFFFAOYSA-N)
```

4.5.6 Using the coco namespace to define reactions with `equilibrator-api`

This method uses the `equilibrator_api` and the `LocalCompoundCache` to enable custom-compound use.

[5]:
```
from equilibrator_api import ComponentContribution, Q_
# the local cache is passed to ComponentContribution
cc = ComponentContribution(ccache = lc.ccache)
```

[6]:
```
# use coco:ID to access user-specified coco namespace
rxn = cc.parse_reaction_formula("coco:3B4HA + kegg:C00001 = coco:3A4HA + kegg:C00180")
if not rxn.is_balanced():
    print('%s is not balanced' % rxn)

cc.p_h = Q_(7)  # set pH
cc.ionic_strength = Q_("100 mM")  # set I

print(f"G0 = {cc.standard_dg(rxn)}")
print(f"G'0 = {cc.standard_dg_prime(rxn)}")
print(f"G'm = {cc.physiological_dg_prime(rxn)}")
```

```
G0 = (39.1 +/- 3.4) kilojoule / mole
G'0 = (-2.0 +/- 3.4) kilojoule / mole
G'm = (-19.1 +/- 3.4) kilojoule / mole
```

4.5.7 Using `get_compounds` to directly generate `Compound` objects

The `get_compounds` method accepts a single string or a list of strings that are molecule structures in either smiles or inchi form. The database is queried for each molecule and any misses are generated and inserted into the database. A list of compounds is returned.

Generated compounds are assigned an ID that is one greater than the current largest ID.

[7]:
```
compound_list = lc.get_compounds(["CC(=O)O", "CC(O)C(=O)O", 'CCCOP(=O)(O)O',
↪"OCC(N)C(O)CO"])

for c in compound_list:
    print("-" * 80)
    print(c)
    print(f"pK_a: {c.dissociation_constants}")
    print(f"pK_Mg: {c.magnesium_dissociation_constants}")
    print("Microspecies: ")
    for ms in c.microspecies:
        print(f"{ms}, ddg_over_rt = {ms.ddg_over_rt:.1f}")
```

```
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
```

```
--------------------------------------------------------------------------------
Compound(id=28, inchi_key=QTBSBXVTEAMEQO-UHFFFAOYSA-M)
pK_a: [4.54]
pK_Mg: [MagnesiumDissociationConstant(compound_id=28, number_protons=3, number_
↪magnesiums=1)]
Microspecies:
CompoundMicrospecies(compound_id=28, z=-1, nH=3, nMg=0), ddg_over_rt = 0.0
CompoundMicrospecies(compound_id=28, z=0, nH=4, nMg=0), ddg_over_rt = -10.5
CompoundMicrospecies(compound_id=28, z=1, nH=3, nMg=1), ddg_over_rt = -186.1
--------------------------------------------------------------------------------
Compound(id=2667, inchi_key=JVTAAEKCZFNVCJ-UHFFFAOYSA-M)
pK_a: [3.78]
pK_Mg: []
Microspecies:
CompoundMicrospecies(compound_id=2667, z=-1, nH=5, nMg=0), ddg_over_rt = 0.0
CompoundMicrospecies(compound_id=2667, z=0, nH=6, nMg=0), ddg_over_rt = -8.7
--------------------------------------------------------------------------------
Compound(id=694326, inchi_key=MHZDONKZSXBOGL-UHFFFAOYSA-N)
pK_a: [6.84, 1.82]
pK_Mg: []
Microspecies:
CompoundMicrospecies(compound_id=694326, z=-2, nH=7, nMg=0), ddg_over_rt = 0.0
CompoundMicrospecies(compound_id=694326, z=-1, nH=8, nMg=0), ddg_over_rt = -15.7
CompoundMicrospecies(compound_id=694326, z=0, nH=9, nMg=0), ddg_over_rt = -19.9
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Compound(id=694327, inchi_key=PMLGQXIKBPFHJZ-UHFFFAOYSA-N)
pK_a: [13.69, 8.92]
pK_Mg: []
Microspecies:
CompoundMicrospecies(compound_id=694327, z=-1, nH=10, nMg=0), ddg_over_rt = 52.1
CompoundMicrospecies(compound_id=694327, z=0, nH=11, nMg=0), ddg_over_rt = 20.5
CompoundMicrospecies(compound_id=694327, z=1, nH=12, nMg=0), ddg_over_rt = 0.0
```

Highlighting local cache persistence

Compounds remain in the local cache between runs. To highlight this, two compounds are added to local cache and given ids. The cache is reloaded and the compounds are queried in reverse, showing the ids remain with the specific compound.

[8]:
```
# get two new compounds
cpds_before = lc.get_compounds(["C(CC)CCOP(=O)(O)O", "C(CCC)CCOP(=O)(O)O"])

print('Before Reload')
for cpd in cpds_before:
    print(f"\tID: {cpd.id}, InChI Key: {cpd.inchi_key}")

print("\n")
# reload cache
lc.ccache.session.close()
lc.load_cache('compounds.sqlite')
print("\n")

# query compounds in reverse
# ids stay with inchi keys, indicating compound persistence in the local cache
cpds_after = lc.get_compounds(["C(CCC)CCOP(=O)(O)O", "C(CC)CCOP(=O)(O)O"])

print('After Reload')
for cpd in cpds_after:
    print(f"\tID: {cpd.id}, InChI Key: {cpd.inchi_key}")
```

```
Before Reload
        ID: 694328, InChI Key: NVTPMUHPCAUGCB-UHFFFAOYSA-N
        ID: 694329, InChI Key: PHNWGDTYCJFUGZ-UHFFFAOYSA-N


Loading compounds from compounds.sqlite


After Reload
        ID: 694329, InChI Key: PHNWGDTYCJFUGZ-UHFFFAOYSA-N
        ID: 694328, InChI Key: NVTPMUHPCAUGCB-UHFFFAOYSA-N
```

4.5.8 Exploring More Options of add_compounds and get_compounds

There are a number of options to further control the behavior of get_compounds that will be explained below:

1. Varying the inchi-key connectivity for searches
2. Handling compound creation errors
    - Investigting Log
    - Bypassing Chemaxon
    - Inserting Empty Compounds
    - Returning Failed Compounds

Inchi-key block control over searches

The connectivity_only option in get_compounds allows for the use of only the first block in the InChI key to be used in a search, otherwise the first two blocks will be used.

An example is shown with D-Glucose and L-Glucose. The connectivity-only searches yield the same results, as is expected.

[9]:
```
cc = ComponentContribution()
TRAINING_IDS = cc.predictor.params.train_G.index

d_glucose_con = lc.get_compounds('C([C@@H]1[C@H]([C@@H]([C@H]([C@H](O1)O)O)O)O)O',␣
↪connectivity_only=True)
d_glucose = lc.get_compounds('C([C@@H]1[C@H]([C@@H]([C@H]([C@H](O1)O)O)O)O)O',␣
↪connectivity_only=False)
l_glucose_con = lc.get_compounds('O[C@@H]1[C@@H](O)[C@@H](OC(O)[C@H]1O)CO', connectivity_
↪only=True)
l_glucose = lc.get_compounds('O[C@@H]1[C@@H](O)[C@@H](OC(O)[C@H]1O)CO', connectivity_
↪only=False)

print("D-Glucose Search")
print(f"Two InChI Key blocks: {d_glucose}\nIn training data: {d_glucose.id in TRAINING_
↪IDS}")
print(f"\nConnectivity Only: {d_glucose_con}\nIn training data: {d_glucose_con.id in␣
↪TRAINING_IDS}")

print('\n')
print("L-Glucose Search")
print(f"Two InChI Key blocks: {l_glucose}\nIn training data: {l_glucose.id in TRAINING_
↪IDS}")
print(f"\nConnectivity Only: {l_glucose_con}\nIn training data: {l_glucose_con.id in␣
↪TRAINING_IDS}")
```

```
D-Glucose Search
Two InChI Key blocks: Compound(id=93, inchi_key=WQZGKKKJIJFFOK-DVKNGEFBSA-N)
In training data: False

Connectivity Only: Compound(id=43, inchi_key=WQZGKKKJIJFFOK-GASJEMHNSA-N)
In training data: True
```

(continues on next page)

(continued from previous page)

```
L-Glucose Search
Two InChI Key blocks: Compound(id=11639, inchi_key=WQZGKKKJIJFFOK-ZZWDRFIYSA-N)
In training data: False

Connectivity Only: Compound(id=43, inchi_key=WQZGKKKJIJFFOK-GASJEMHNSA-N)
In training data: True
```

Handling Compound Creation Errors

Sometimes compounds fail to be decomposed. This is due to chemaxon errors or the structure being invalid. As a result, there are a few workarounds to this problem. Users can specify two options, `bypass_chemaxon` and `save_empty_compounds`, to get around these errors.

`bypass_chemxon` will attempt to create a compound from the user-specified structure. If the compound cannot be decomposed even without `bypass_chemaxon=True` then it can still be saved as an empty compound by specifying `save_empty_compounds=True`.

There are two ways to log results, an error_log, which saves to a .tsv, or through a pandas dataframe.

Viewing error log

[ ]:
```
log_df = pd.DataFrame()
smiles = [
        # Decomposed with chemaxon
        "OC(=O)C1=CC(NC(=O)C2=CC=CC=C2)=C(O)C=C1",
        # Decomposed with bypass_chemaxon
        "C1(CC(OC(C(C1)=O)CO)O)=O",
        # Cannot be decomposed
        "CC(=O)OC1C=C2C3Cc4ccc(c(c4C2(CCN3C)C=C1OC)O)OC",
    ]

compounds = lc.get_compounds(
    smiles,
    bypass_chemaxon=False,
    save_empty_compounds=False,
    error_log="compound_creation_log.tsv",
    log_df=log_df
)
```

compound_creation_log

The log gives the structure, the method that can insert the compound, and the status.

[ ]:
```
# succesfully insert with chemaxon
cc_log = pd.read_csv("compound_creation_log.tsv", sep="\t", index_col=[0])
print(f"----------\nSaved Log\n{cc_log.to_string()}\n")
print(f"----------\nDataFrame Log\n{log_df.to_string()}")
```

[12]:
```
smiles = [
        # Decomposed with chemaxon
        "OC(=O)C1=CC(NC(=O)C2=CC=CC=C2)=C(O)C=C1",
        # Decomposed with bypass_chemaxon
        "C1(CC(OC(C(C1)=O)CO)O)=O",
        # Cannot be decomposed
        "CC(=O)OC1C=C2C3Cc4ccc(c(c4C2(CCN3C)C=C1OC)O)OC",
    ]

compounds = lc.get_compounds(
    smiles,
    bypass_chemaxon=True,
    save_empty_compounds=False,
)
# Succesfully insert empty compound
```

```
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
WARNING:equilibrator_assets.generate_compound:One or more compounds were unable to be␣
↪decomposed. Rerun specifying error_log or log_df to view details.
```

[13]:
```
smiles = [
        # Decomposed with chemaxon
        "OC(=O)C1=CC(NC(=O)C2=CC=CC=C2)=C(O)C=C1",
        # Decomposed with bypass_chemaxon
        "C1(CC(OC(C(C1)=O)CO)O)=O",
        # Cannot be decomposed
        "CC(=O)OC1C=C2C3Cc4ccc(c(c4C2(CCN3C)C=C1OC)O)OC",
    ]

compounds = lc.get_compounds(
    smiles,
    bypass_chemaxon=True,
    save_empty_compounds=True,
    log_df=log_df
)
# Succesfully insert empty compound
print(f"{log_df.to_string()}")
```

```
==============================
*** Open Babel Warning  in InChI code
```

(continues on next page)

(continued from previous page)

```
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
```

	struct	inchi_key	method	status
0	OC(=O)C1=CC(NC(=O)C2=CC=CC=C2)=C(O)C=C1	RKCVLDMDZASBEO-UHFFFAOYSA-N	database	valid
1	C1(CC(OC(C(C1)=O)CO)O)=O	UBJAOLUWBPQQJC-UHFFFAOYSA-N	database	valid
2	CC(=O)OC1C=C2C3Cc4ccc(c(c4C2(CCN3C)C=C1OC)O)OC	DNOMLUPMYHAJIY-UHFFFAOYSA-N	empty	valid

Returning failed compounds

If the shape of the output list of get_compounds must be the same as the input, set `return_failures==True`. This returns None for compounds that failed.

The below only returns 2 compounds, as the third one cannot be decomposed.

[14]:
```
smiles = [
        # Already in database
        "CCO",
        # Decomposed with bypass_chemaxon
        "C1(CCC(OC(C(C1)=O)CO)O)=O",
        # Cannot be decomposed
        "CCC(=O)OC1C=C2C3Cc4ccc(c(c4C2(CCN3C)C=C1OC)O)OC",
    ]
lc.get_compounds(smiles)
```

```
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
WARNING:equilibrator_assets.generate_compound:One or more compounds were unable to be␣
↪decomposed. Rerun specifying error_log or log_df to view details.
```

[14]:
```
[Compound(id=287, inchi_key=LFQSCWFLJHTTHZ-UHFFFAOYSA-N),
 Compound(id=694332, inchi_key=XNYRWWMIBJOLGA-UHFFFAOYSA-N)]
```

Set `return_fails=True` to return None for failed compounds

[15]:
```
compounds = lc.get_compounds(smiles, return_fails=True)
smiles_dict = dict(zip(smiles, compounds))
print("\nCompounds:")
print(compounds)
print("\nDict:")
print(smiles_dict)
```

```
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
==============================
*** Open Babel Warning  in InChI code
  #1 :Omitted undefined stereo
WARNING:equilibrator_assets.generate_compound:One or more compounds were unable to be␣
↪decomposed. Rerun specifying error_log or log_df to view details.


Compounds:
[Compound(id=287, inchi_key=LFQSCWFLJHTTHZ-UHFFFAOYSA-N), Compound(id=694332, inchi_
↪key=XNYRWWMIBJOLGA-UHFFFAOYSA-N), None]

Dict:
{'CCO': Compound(id=287, inchi_key=LFQSCWFLJHTTHZ-UHFFFAOYSA-N),
↪'C1(CCC(OC(C(C1)=O)CO)O)=O': Compound(id=694332, inchi_key=XNYRWWMIBJOLGA-UHFFFAOYSA-
↪N), 'CCC(=O)OC1C=C2C3Cc4ccc(c(c4C2(CCN3C)C=C1OC)O)OC': None}
```

4.6 API Reference

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

4.6.1 equilibrator_api

Subpackages

equilibrator_api.data

equilibrator_api.model

Submodules

equilibrator_api.model.bounds

Define lower and upper bounds on compounds.

[1] Created with sphinx-autoapi

Module Contents

class equilibrator_api.model.bounds.**BaseBounds**

Bases: object

A base class for declaring bounds on things.

abstract copy()

Return a (deep) copy of self.

abstract get_lower_bound(*compound: Union[str, equilibrator_cache.Compound]*)

Get the lower bound for this key.

Parameters
key – a compound

abstract get_upper_bound(*compound: Union[str, equilibrator_cache.Compound]*)

Get the upper bound for this key.

Parameters
key – a compound

get_lower_bounds(*compounds: Iterable[Union[str, equilibrator_cache.Compound]]*) → Iterable[equilibrator_api.Q_]

Get the bounds for a set of keys in order.

Parameters
compounds – an iterable of Compounds or strings

Returns
an iterable of the lower bounds

get_upper_bounds(*compounds: Iterable[Union[str, equilibrator_cache.Compound]]*) → Iterable[equilibrator_api.Q_]

Get the bounds for a set of keys in order.

Parameters
compounds – an iterable of Compounds or strings

Returns
an iterable of the upper bounds

get_bound_tuple(*compound: Union[str, equilibrator_cache.Compound]*) → Tuple[equilibrator_api.Q_, equilibrator_api.Q_]

Get both upper and lower bounds for this key.

Parameters
compound – a Compound object or string

Returns
a 2-tuple (lower bound, upper bound)

get_bounds(*compounds: Iterable[Union[str, equilibrator_cache.Compound]]*) → Tuple[Iterable[equilibrator_api.Q_], Iterable[equilibrator_api.Q_]]

Get the bounds for a set of compounds.

Parameters
compounds – an iterable of Compounds

Returns
a 2-tuple (lower bounds, upper bounds)

static conc2ln_conc(*b: equilibrator_api.Q_*) → float

Convert a concentration to log-concentration.

Parameters
b – a concentration

Returns
the log concentration

get_ln_bounds(*compounds: Iterable[Union[str, equilibrator_cache.Compound]]*) → Tuple[Iterable[float], Iterable[float]]

Get the log-bounds for a set of compounds.

Parameters
compounds – an iterable of Compounds or strings

Returns
a 2-tuple (log lower bounds, log upper bounds)

get_ln_lower_bounds(*compounds: Iterable[Union[str, equilibrator_cache.Compound]]*) → Iterable[float]

Get the log lower bounds for a set of compounds.

Parameters
compounds – an iterable of Compounds or strings

Returns
an iterable of log lower bounds

get_ln_upper_bounds(*compounds: Iterable[Union[str, equilibrator_cache.Compound]]*) → Iterable[float]

Get the log upper bounds for a set of compounds.

Parameters
compounds – an iterable of Compounds or strings

Returns
an iterable of log upper bounds

set_bounds(*compound: Union[str, equilibrator_cache.Compound]*, *lb: equilibrator_api.Q_*, *ub: equilibrator_api.Q_*) → None

Set bounds for a specific key.

Parameters

- **key** – a Compounds or string
- **lb** – the lower bound value
- **ub** – the upper bound value

class equilibrator_api.model.bounds.**Bounds**(*lower_bounds: Dict[Union[str, equilibrator_cache.Compound], equilibrator_api.Q_] = None*, *upper_bounds: Dict[Union[str, equilibrator_cache.Compound], equilibrator_api.Q_] = None*, *default_lb: equilibrator_api.Q_ = default_conc_lb*, *default_ub: equilibrator_api.Q_ = default_conc_ub*)

Bases: *BaseBounds*

Contains upper and lower bounds for various keys.

Allows for defaults.

DEFAULT_BOUNDS

classmethod from_csv(*f: TextIO*, *comp_contrib: equilibrator_api.ComponentContribution*, *default_lb: equilibrator_api.Q_ = default_conc_lb*, *default_ub: equilibrator_api.Q_ = default_conc_ub*) → *Bounds*

Read Bounds from a CSV file.

Parameters

- **f** (*TextIO*) – an open .csv file stream
- **comp_contrib** (ComponentContribution) – used for parsing compound accessions
- **default_lb** (Q_) – the default lower bound
- **default_ub** (Q_) – the default upper bound

to_data_frame() → pandas.DataFrame

Convert the list of bounds to a Pandas DataFrame.

check_bounds() → None

Assert the bounds are valid (i.e. that lb <= ub).

copy() → *Bounds*

Return a deep copy of self.

get_lower_bound(*compound: Union[str, equilibrator_cache.Compound]*) → equilibrator_api.Q_

Get the lower bound for this compound.

get_upper_bound(*compound: Union[str, equilibrator_cache.Compound]*) → equilibrator_api.Q_

Get the upper bound for this compound.

static get_default_bounds(*comp_contrib: equilibrator_api.ComponentContribution*) → *Bounds*

Return the default lower and upper bounds for a pre-determined list.

Parameters
comp_contrib (ComponentContribution) –

Return type
a Bounds object with the default values

equilibrator_api.model.model

a basic stoichiometric model with thermodynamics.

Module Contents

class equilibrator_api.model.model.**StoichiometricModel**(*S: pandas.DataFrame*, *compound_dict: Dict[str, equilibrator_cache.Compound]*, *reaction_dict: Dict[str, equilibrator_cache.Reaction]*, *comp_contrib: Optional[equilibrator_api.ComponentContribution] = None*, *standard_dg_primes: Optional[equilibrator_api.Q_] = None*, *dg_sigma: Optional[equilibrator_api.Q_] = None*, *bounds: Optional[equilibrator_api.model.Bounds] = None*, *config_dict: Optional[Dict[str, str]] = None*)

Bases: object

A basic stoichiometric model with thermodynamics.

Designed as a base model for 'Pathway' which also includes flux directions and magnitudes.

MINIMAL_STDEV = 0.001

configure() → None

Configure the Component Contribution aqueous conditions.

property compound_ids → Iterable[str]

Get the list of compound IDs.

property compounds → Iterable[equilibrator_cache.Compound]

Get the list of Compound objects.

property compound_df → pandas.DataFrame

Get a DataFrame with all the compound data.

The columns are:
compound_id lower_bound upper_bound

property reaction_ids → Iterable[str]

Get the list of reaction IDs.

property reactions → Iterable[equilibrator_cache.Reaction]

Get the list of Reaction objects.

property reaction_formulas → Iterable[str]

Iterate through all the reaction formulas.

Returns
the reaction formulas

property reaction_df → pandas.DataFrame

Get a DataFrame with all the reaction data.

The columns are:
reaction_id reaction_formula standard_dg_prime

update_standard_dgs() → None

Calculate the standard G' values and uncertainties.

Use the Component Contribution method.

set_bounds(*cid: str*, *lb: Optional[equilibrator_api.Q_] = None*, *ub: Optional[equilibrator_api.Q_] = None*) → None

Set the lower and upper bound of a compound.

Parameters

- **compound_id** (*str*) – the compound ID
- **lb** (*Quantity, optional*) – the new concentration lower bound (ignored if the value is None)
- **ub** (*Quantity, optional*) – the new concentration upper bound (ignored if the value is None)

get_bounds(*cid: str*) → Tuple[equilibrator_api.Q_, equilibrator_api.Q_]

Get the lower and upper bound of a compound.

Parameters
compound_id (*str*) – the compound ID

Returns

- **lb** (*Quantity, optional*) – the new concentration lower bound (ignored if the value is None)
- **ub** (*Quantity, optional*) – the new concentration upper bound (ignored if the value is None)

property bounds → Tuple[Iterable[equilibrator_api.Q_], Iterable[equilibrator_api.Q_]]

Get the concentration bounds.

The order of compounds is according to the stoichiometric matrix index.

Return type
tuple of (lower bounds, upper bounds)

property bound_df → pandas.DataFrame

Get a DataFrame with all the bounds data.

property ln_conc_lb → numpy.array

Get the log lower bounds on the concentrations.

The order of compounds is according to the stoichiometric matrix index.

Return type
a NumPy array of the log lower bounds

property ln_conc_ub → numpy.ndarray

Get the log upper bounds on the concentrations.

The order of compounds is according to the stoichiometric matrix index.

Return type
a NumPy array of the log upper bounds

property ln_conc_mu → numpy.array

Get mean of log concentration distribution based on the bounds.

The order of compounds is according to the stoichiometric matrix index.

Return type
a NumPy array with the mean of the log concentrations

property ln_conc_sigma → numpy.array

Get stdev of log concentration distribution based on the bounds.

Return type
a NumPy array with the stdev of the log concentrations

static read_thermodynamics(*thermo_sbtab: equilibrator_api.model.SBtabTable*, *config_dict: Dict[str, str]*) → Dict[str, equilibrator_api.Q_]

Read the 'thermodynamics' table from an SBtab.

Parameters

- **thermo_sbtab** (*SBtabTable*) – A SBtabTable containing the thermodynamic data
- **config_dict** (*dict*) – A dictionary containing the configuration arguments

Return type
A dictionary mapping reaction IDs to standard G' values.

classmethod from_network_sbtab(*filename: Union[str, equilibrator_api.model.SBtabDocument]*, *comp_contrib: Optional[equilibrator_api.ComponentContribution] = None*, *freetext: bool = True*, *bounds: Optional[equilibrator_api.model.Bounds] = None*) → *StoichiometricModel*

Initialize a Pathway object using a 'network'-only SBtab.

Parameters

- **filename** (*str, SBtabDocument*) – a filename containing an SBtabDocument (or the SBtabDocument object itself) defining the network (topology) only
- **comp_contrib** (ComponentContribution, *optional*) – a ComponentContribution object needed for parsing and searching the reactions. also used to set the aqueous parameters (pH, I, etc.)
- **freetext** (*bool, optional*) – a flag indicating whether the reactions are given as free-text (i.e. common names for compounds) or by standard database accessions (Default value: *True*)
- **bounds** (Bounds, *optional*) – bounds on metabolite concentrations (by default uses the "data/cofactors.csv" file in *equilibrator-api*)

Return type
a Pathway object

classmethod from_sbtab(*filename: Union[str, equilibrator_api.model.SBtabDocument]*, *comp_contrib: Optional[equilibrator_api.ComponentContribution] = None*) → *StoichiometricModel*

Parse and SBtabDocument and return a StoichiometricModel.

Parameters

- **filename** (*str or SBtabDocument*) – a filename containing an SBtabDocument (or the SBtabDocument object itself) defining the pathway
- **comp_contrib** (ComponentContribution, *optional*) – a ComponentContribution object needed for parsing and searching the reactions. also used to set the aqueous parameters (pH, I, etc.)

Returns
stoich_model – A StoichiometricModel object based on the configuration SBtab

Return type
StoichiometricModel

`to_sbtab`() → equilibrator_api.model.SBtabDocument

Export the model to an SBtabDocument.

`write_sbtab`(*filename: str*) → None

Write the pathway to an SBtab file.

Package Contents

equilibrator_api.model.**open_sbtabdoc**(*filename: Union[str, sbtab.SBtab.SBtabDocument]*) → sbtab.SBtab.SBtabDocument

Open a file as an SBtabDocument.

Checks whether it is already an SBtabDocument object, otherwise reads the CSV file and returns the parsed object.

Submodules

equilibrator_api.compatibility

Provide functions for compatibility with COBRA.

Module Contents

equilibrator_api.compatibility.**map_cobra_reactions**(*cache: equilibrator_cache.CompoundCache*, *reactions: List[cobra.Reaction]*, ***kwargs*) → Dict[str, *equilibrator_api.phased_reaction.PhasedReaction*]

Translate COBRA reactions to eQuilibrator phased reactions.

Parameters

- **cache** (*equilibrator_cache.CompoundCache*) –
- **reactions** (*iterable of cobra.Reaction*) – A list of reactions to map to equilibrator phased reactions.
- **kwargs** – Any further keyword arguments are passed to the underlying function for mapping metabolites.

Returns
A mapping from COBRA reaction identifiers to equilibrator phased reactions where such a mapping can be established.

Return type
dict

See also:

`equilibrator_cache.compatibility.map_cobra_metabolites`

equilibrator_api.component_contribution

A wrapper for the GibbeEnergyPredictor in component-contribution.

Module Contents

equilibrator_api.component_contribution.**find_most_abundant_ms**(*cpd: equilibrator_cache.Compound*, *p_h: equilibrator_api.Q_*, *p_mg: equilibrator_api.Q_*, *ionic_strength: equilibrator_api.Q_*, *temperature: equilibrator_api.Q_*) → equilibrator_cache.CompoundMicrospecies

Find the most abundant microspecies based on transformed energies.

equilibrator_api.component_contribution.**predict_protons_and_charge**(*rxn: equilibrator_api.phased_reaction.PhasedReaction*, *p_h: equilibrator_api.Q_*, *p_mg: equilibrator_api.Q_*, *ionic_strength: equilibrator_api.Q_*, *temperature: equilibrator_api.Q_*) → Tuple[float, float, float]

Find the #protons and charge of a transport half-reaction.

class equilibrator_api.component_contribution.**ComponentContribution**(*rmse_inf: equilibrator_api.Q_ = default_rmse_inf*, *ccache: Optional[equilibrator_cache.CompoundCache] = None*, *predictor: Optional[component_contribution.predict.GibbsE = None*)

Bases: object

A wrapper class for GibbsEnergyPredictor.

Also holds default conditions for compounds in the different phases.

property p_h → equilibrator_api.Q_

Get the pH.

property p_mg → equilibrator_api.Q_

Get the pMg.

property ionic_strength → equilibrator_api.Q_

Get the ionic strength.

property temperature → equilibrator_api.Q_

Get the temperature.

static legacy() → *ComponentContribution*

Initialize a ComponentContribution object with the legacy version.

The legacy version is intended for compatibility with older versions of equilibrator api (0.2.x - 0.3.1). Starting from 0.3.2, there is a significant change in the predictions caused by an improved Mg2+ concentration model.

Return type
A ComponentContribution object

static initialize_custom_version(*rmse_inf: equilibrator_api.Q_ = default_rmse_inf*, *ccache_settings: component_contribution.ZenodoSettings = DEFAULT_COMPOUND_CACHE_SETTINGS*, *cc_params_settings: component_contribution.ZenodoSettings = DEFAULT_CC_PARAMS_SETTINGS*) → *ComponentContribution*

Initialize ComponentContribution object with custom Zenodo versions.

Parameters

- **rmse_inf** (*Quantity, optional*) – Set the factor by which to multiply the error covariance matrix for reactions outside the span of Component Contribution. (Default value: 1e-5 kJ/mol)
- **settings** (*ZenodoSettings*) – The doi, filename and md5 of the

Return type
A ComponentContribution object

get_compound(*compound_id: str*) → Union[equilibrator_cache.Compound, None]

Get a Compound using the DB namespace and its accession.

Returns
cpd

Return type
Compound

get_compound_by_inchi(*inchi: str*) → Union[equilibrator_cache.Compound, None]

Get a Compound using InChI.

Returns
cpd

Return type
Compound

search_compound_by_inchi_key(*inchi_key: str*) → List[equilibrator_cache.Compound]

Get a Compound using InChI.

Returns
cpd

Return type
Compound

search_compound(*query: str*) → Union[None, equilibrator_cache.Compound]

Try to find the compound that matches the name best.

Parameters
query (*str*) – an (approximate) compound name

Returns
cpd – the best match

Return type
Compound

parse_reaction_formula(*formula: str*) → *equilibrator_api.phased_reaction.PhasedReaction*

Parse reaction text using exact match.

Parameters
formula (*str*) – a string containing the reaction formula

Returns
rxn

Return type
PhasedReaction

search_reaction(*formula: str*) → *equilibrator_api.phased_reaction.PhasedReaction*

Search a reaction written using compound names (approximately).

Parameters
formula (*str*) – a string containing the reaction formula

Returns
rxn

Return type
PhasedReaction

balance_by_oxidation(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → *equilibrator_api.phased_reaction.PhasedReaction*

Convert an unbalanced reaction into an oxidation reaction.

By adding H2O, O2, Pi, CO2, and NH4+ to both sides.

get_oxidation_reaction(*compound: equilibrator_cache.Compound*) → *equilibrator_api.phased_reaction.PhasedReaction*

Generate an oxidation Reaction for a single compound.

Generate a Reaction object which represents the oxidation reaction of this compound using O2. If there are N atoms, the product must be NH3 (and not N2) to represent biological processes. Other atoms other than C, N, H, and O will raise an exception.

property RT → equilibrator_api.Q_

Get the value of RT.

standard_dg_formation(*compound: equilibrator_cache.Compound*) → Tuple[Optional[float], Optional[numpy.ndarray]]

Get the (mu, sigma) predictions of a compound's formation energy.

Parameters
compound (*Compound*) – a Compound object

Returns

- **mu** (*float*) – the mean of the standard formation Gibbs energy estimate
- **sigma_fin** (*array*) – a vector representing the square root of the covariance matrix (uncertainty)

- **sigma_inf** (*array*) – a vector representing the infinite-uncertainty eigenvalues of the covariance matrix

standard_dg(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the chemical reaction energies of a reaction.

Returns
standard_dg – the dG0 in kJ/mol and standard error. To calculate the 95% confidence interval, use the range -1.96 to 1.96 times this value

Return type
ureg.Measurement

standard_dg_multi(*reactions: List[*equilibrator_api.phased_reaction.PhasedReaction*]*, *uncertainty_representation: str = 'cov'*) → Tuple[numpy.ndarray, numpy.ndarray]

Calculate the chemical reaction energies of a list of reactions.

Using the major microspecies of each of the reactants.

Parameters

- **reactions** (*List[*PhasedReaction*]*) – a list of PhasedReaction objects to estimate
- **uncertainty_representation** (*str*) – which representation to use for the uncertainties. *cov* would return a full covariance matrix. *precision* would return the precision matrix (i.e. the inverse of the covariance matrix). *sqrt* would return a sqaure root of the covariance, based on the uncertainty vectors. *fullrank* would return a full-rank square root of the covariance which is a compressed form of the *sqrt* result. (Default value: *cov*)

Returns

- **standard_dg** (*Quantity*) – the estimated standard reaction Gibbs energies based on the the major microspecies
- **dg_uncertainty** (*Quantity*) – the uncertainty matrix (in either 'cov', 'sqrt' or 'fullrank' format)

standard_dg_prime(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the transformed reaction energies of a reaction.

Returns
standard_dg – the dG0_prime in kJ/mol and standard error. To calculate the 95% confidence interval, use the range -1.96 to 1.96 times this value

Return type
ureg.Measurement

dg_prime(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the dG'0 of a single reaction.

Returns
dg – the dG_prime in kJ/mol and standard error. To calculate the 95% confidence interval, use the range -1.96 to 1.96 times this value

Return type
ureg.Measurement

standard_dg_prime_multi(*reactions: List[*equilibrator_api.phased_reaction.PhasedReaction*]*, *uncertainty_representation: str = 'cov'*, *minimize_norm: bool = False*) → Tuple[equilibrator_api.Q_, equilibrator_api.Q_]

Calculate the transformed reaction energies of a list of reactions.

Parameters

- **reactions** (*List[*PhasedReaction*]*) – a list of PhasedReaction objects to estimate
- **uncertainty_representation** (*str*) – which representation to use for the uncertainties. *cov* would return a full covariance matrix. *precision* would return the precision matrix (i.e. the inverse of the covariance matrix). *sqrt* would return a sqaure root of the covariance, based on the uncertainty vectors. *fullrank* would return a full-rank square root of the covariance which is a compressed form of the *sqrt* result. (Default value: *cov*)
- **minimize_norm** (*bool*) – if True, use an orthogonal projection to minimize the norm2 of the result vector (keeping it within the finite-uncertainty sub-space, i.e. only moving along eigenvectors with infinite uncertainty).

Returns

- **standard_dg_prime** (*Quantity*) – the CC estimation of the reactions' standard transformed energies
- **dg_uncertainty** (*Quantity*) – the uncertainty co-variance matrix (in either 'cov', 'sqrt' or 'fullrank' format)

physiological_dg_prime(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the dG'm of a single reaction.

Assume all aqueous reactants are at 1 mM, gas reactants at 1 mbar and the rest at their standard concentration.

Returns

- **standard_dg_primes** (*ndarray*) – a 1D NumPy array containing the CC estimates for the reactions' physiological dG'
- **dg_sigma** (*ndarray*) – the second is a 2D numpy array containing the covariance matrix of the standard errors of the estimates. one can use the eigenvectors of the matrix to define a confidence high-dimensional space, or use dg_sigma as the covariance of a Gaussian used for sampling (where 'standard_dg_primes' is the mean of that Gaussian).

dgf_prime_sensitivity_to_p_h(*compound: equilibrator_cache.Compound*) → equilibrator_api.ureg.Quantity

Calculate the sensitivity of the chemical formation energy to pH.

Returns
The derivative of ΔG_f with respect to pH, in kJ/mol.

Return type
Quantity

dg_prime_sensitivity_to_p_h(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Quantity

Calculate the sensitivity of the chemical reaction energy to pH.

Returns
The derivative of ΔG_r with respect to pH, in kJ/mol.

Return type
Quantity

ln_reversibility_index(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the reversibility index (ln Gamma) of a single reaction.

Returns
ln_RI – the reversibility index (in natural log scale).

Return type
ureg.Measurement

standard_e_prime(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the E'0 of a single half-reaction.

Returns

- **standard_e_prime** (*ureg.Measurement*)
- *the estimated standard electrostatic potential of reaction and*
- *E0_uncertainty is the standard deviation of estimation. Multiply it*
- *by 1.96 to get a 95% confidence interval (which is the value shown on*
- *eQuilibrator).*

physiological_e_prime(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the E'0 of a single half-reaction.

Returns

- **physiological_e_prime** (*ureg.Measurement*)
- *the estimated physiological electrostatic potential of reaction and*
- *E0_uncertainty is the standard deviation of estimation. Multiply it*
- *by 1.96 to get a 95% confidence interval (which is the value shown on*
- *eQuilibrator).*

e_prime(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → equilibrator_api.ureg.Measurement

Calculate the E'0 of a single half-reaction.

Returns

- **e_prime** (*ureg.Measurement*)
- *the estimated electrostatic potential of reaction and*
- *E0_uncertainty is the standard deviation of estimation. Multiply it*
- *by 1.96 to get a 95% confidence interval (which is the value shown on*
- *eQuilibrator).*

dg_analysis(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → List[Dict[str, object]]

Get the analysis of the component contribution estimation process.

Return type
the analysis results as a list of dictionaries

is_using_group_contribution(*reaction:* equilibrator_api.phased_reaction.PhasedReaction) → bool

Check whether group contribution is needed to get this reactions' dG.

Return type
true iff group contribution is needed

multicompartmental_standard_dg_prime(*reaction_inner:* equilibrator_api.phased_reaction.PhasedReaction, *reaction_outer:* equilibrator_api.phased_reaction.PhasedReaction, *e_potential_difference: equilibrator_api.Q_*, *p_h_outer: equilibrator_api.Q_*, *ionic_strength_outer: equilibrator_api.Q_*, *p_mg_outer: equilibrator_api.Q_ = default_physiological_p_mg*, *tolerance: float = 0.0*) → equilibrator_api.ureg.Measurement

Calculate the transformed energies of a multi-compartmental reaction.

Based on the equations from Harandsdottir et al. 2012 (https://doi.org/10.1016/j.bpj.2012.02.032)

Parameters

- **reaction_inner** (PhasedReaction) – the inner compartment half-reaction
- **reaction_outer** (PhasedReaction) – the outer compartment half-reaction
- **e_potential_difference** (*Quantity*) – the difference in electro-static potential between the outer and inner compartments
- **p_h_outer** (*Quantity*) – the pH in the outside compartment
- **ionic_strength_outer** (*Quantity*) – the ionic strength outside
- **p_mg_outer** (*Quantity (optional)*) – the pMg in the outside compartment
- **tolerance** (*Float (optional)*) – tolerance for identifying inbalance between inner and outer reactions (default = 0)

Returns
standard_dg_prime – the transport reaction Gibbs free energy change

Return type
Measurement

static parse_formula_side(*s: str*) → Dict[str, float]

Parse one side of the reaction formula.

static parse_formula(*formula: str*) → Dict[str, float]

Parse a two-sided reaction formula.

static create_stoichiometric_matrix_from_reaction_formulas(*formulas: Iterable[str]*) → pandas.DataFrame

Build a stoichiometric matrix.

Parameters
formulas (*Iterable[str]*) – String representations of the reactions.

Returns
The stoichiometric matrix as a DataFrame whose indexes are the compound IDs and its columns are the reaction IDs (in the same order as the input).

Return type
DataFrame

create_stoichiometric_matrix_from_reaction_objects(*reactions: Iterable[equilibrator_api.phased_reaction.PhasedReaction]*) → pandas.DataFrame

Build a stoichiometric matrix.

Parameters
reactions (*Iterable[PhasedReaction]*) – The collection of reactions to build a stoichiometric matrix from.

Returns
The stoichiometric matrix as a DataFrame whose indexes are the compounds and its columns are the reactions (in the same order as the input).

Return type
DataFrame

equilibrator_api.phased_compound

inherit from equilibrator_cache.models.compound.Compound an add phases.

Module Contents

equilibrator_api.phased_compound.**AQUEOUS_PHASE_NAME = aqueous**

equilibrator_api.phased_compound.**GAS_PHASE_NAME = gas**

equilibrator_api.phased_compound.**LIQUID_PHASE_NAME = liquid**

equilibrator_api.phased_compound.**SOLID_PHASE_NAME = solid**

equilibrator_api.phased_compound.**REDOX_PHASE_NAME = redox**

equilibrator_api.phased_compound.**PhaseInfo**

equilibrator_api.phased_compound.**PHASE_INFO_DICT**

equilibrator_api.phased_compound.**NON_AQUEOUS_COMPOUND_DICT**

equilibrator_api.phased_compound.**MicroSpecie**

equilibrator_api.phased_compound.**PHASED_COMPOUND_DICT**

equilibrator_api.phased_compound.**CARBONATE_INCHIS**

class equilibrator_api.phased_compound.**Condition**(*phase: str*, *abundance: equilibrator_api.ureg.Quantity = None*)

Bases: object

A class for defining the conditions of a compound.

I.e. the phase and the abundance.

property phase → str
Return the phase.

property abundance → equilibrator_api.ureg.Quantity
Return the abundance.

property standard_abundance → equilibrator_api.ureg.Quantity

Return the standard abundance in this phase.

property physiolical_abundance → equilibrator_api.ureg.Quantity

Return the default physiological abundance in this phase.

property dimensionality → str

Return the dimensionality of the abundance in this phase.

E.g. [concentration] for aqueous phase, or [pressure] for gas phase. :return: the dimensionality in this phase, or None if abundance is fixed.

property ln_abundance → float

Return the log of the ratio between given and std abundances.

property ln_physiological_abundance → float

Return the log of the ratio between phys and std abundances.

reset_abundance() → None

Reset the abundance to standard abundance.

property is_physiological → bool

Return True iff the abundance is the same as the physiological.

Returns
True if the abundance is in physiological conditions,

or if the abundance if fixed in this phase anyway.

class equilibrator_api.phased_compound.**PhasedCompound**(*compound: equilibrator_cache.Compound*, *condition:* Condition = *None*)

Bases: object

A class that combines a equilibrator_api Compound and a Condition.

static get_default(*compound: equilibrator_cache.Compound*) → *Condition*

Get the default phase of a compound.

Parameters
compound – a Compound

Returns
the default phase

property atom_bag → dict

Get the compound's atom bag.

property smiles → str

Get the compound's InChI.

property inchi → str

Get the compound's InChI.

property inchi_key → str

Get the compound's InChIKey.

property id → int

Get the compound's equilibrator internal ID.

property formula → str

Get the chemical formula.

property mass → float

Get the chemical molecular mass.

property phase → str

Get the phase.

property html_formula → str

Get the chemical formula.

property phase_shorthand → str

Get the phase shorthand (i.e. 'l' for liquid).

property possible_phases → Tuple[str]

Get the possible phases for this compound.

property abundance → equilibrator_api.ureg.Quantity

Get the abundance.

property ln_abundance → float

Return the log of the abundance (for thermodynamic calculations).

property ln_physiological_abundance → float

Return the log of the default physiological abundance.

property is_physiological → bool

Check if the abundance is physiological.

get_stored_standard_dgf_prime(*p_h: equilibrator_api.ureg.Quantity*, *ionic_strength: equilibrator_api.ureg.Quantity*, *temperature: equilibrator_api.ureg.Quantity*, *p_mg: equilibrator_api.ureg.Quantity*) → equilibrator_api.ureg.Quantity

Return the stored formation energy of this phased compound.

Only if it exists, otherwise return None (and we will use component-contribution later to get the reaction energy).

Parameters

- **p_h** –
- **ionic_strength** –
- **temperature** –
- **p_mg** –

Returns

standard_dgf_prime (in kJ/mol)

get_stored_standard_dgf() → equilibrator_api.ureg.Quantity

Return the stored formation energy of this phased compound.

Only if it exists, otherwise return None (and we will use component-contribution later to get the reaction energy).

Returns

standard_dgf (in kJ/mol)

get_stored_microspecie() → MicroSpecie

Get the stored microspecies (from the PHASED_COMPOUND_DICT).

Returns
The MicroSpecie namedtuple with the stored formation energy,

or None if this compound has no stored value at this phase.

serialize() → dict

Return a serialized version of all the compound thermo data.

Returns
a list of dictionaries with all the microspecies data

class equilibrator_api.phased_compound.**Proton**(*compound: equilibrator_cache.Compound*)

Bases: *PhasedCompound*

A class specifically for protons.

property abundance → equilibrator_api.ureg.Quantity

Get the abundance.

property ln_physiological_abundance → float

Return the log of the default physiological abundance.

property ln_abundance → float

Return the log of the abundance (for thermodynamic calculations).

class equilibrator_api.phased_compound.**RedoxCarrier**(*compound: equilibrator_cache.Compound*, *potential: Optional[equilibrator_api.ureg.Quantity] = None*)

Bases: *PhasedCompound*

A class specifically for redox carriers (with a given potential).

get_stored_standard_dgf_prime(*p_h: equilibrator_api.ureg.Quantity*, *ionic_strength: equilibrator_api.ureg.Quantity*, *temperature: equilibrator_api.ureg.Quantity*, *p_mg: equilibrator_api.ureg.Quantity*) → equilibrator_api.ureg.Quantity

Get the standard formation G'.

get_stored_standard_dgf() → equilibrator_api.ureg.Quantity

Get the standard formation G.

property atom_bag → dict

Get the compound's atom bag.

property ln_abundance → float

Return the log of the abundance (for thermodynamic calculations).

property ln_physiological_abundance → float

Return the log of the default physiological abundance.

property is_physiological → bool

Check if the abundance is physiological.

equilibrator_api.phased_reaction

inherit from equilibrator_cache.reaction.Reaction an add phases.

Module Contents

class equilibrator_api.phased_reaction.**PhasedReaction**(*sparse: Dict[equilibrator_cache.Compound, float]*, *arrow: str = '<=>'*, *rid: str = None*, *sparse_with_phases: Dict[*equilibrator_api.phased_compound.PhasedCompound, *float] = None*)

Bases: equilibrator_cache.Reaction

A daughter class of Reaction that adds phases and abundances.

REACTION_COUNTER = 0

static to_phased_compound(*cpd: equilibrator_cache.Compound*) → *equilibrator_api.phased_compound.PhasedCompound*

Convert a Compound object to PhasedCompound.

clone() → *PhasedReaction*

Clone this reaction object.

reverse() → *PhasedReaction*

Return a PhasedReaction with the reverse reaction.

get_element_data_frame() → pandas.DataFrame

Tabulate the elemental composition of all reactants.

Returns
A data frame where the columns are the compounds and the indexes are atomic elements.

Return type
DataFrame

hash_md5(*reversible: bool = True*) → str

Return a MD5 hash of the PhasedReaction.

This hash is useful for finding reactions with the exact same stoichiometry. We create a unique formula string based on the Compound IDs and coefficients.

Parameters
reversible (*bool*) – a flag indicating whether the directionality of the reaction matters or not. if *True*, the same value will be returned for both the forward and backward versions.

Returns
hash – a unique hash string representing the Reaction.

Return type
str

set_abundance(*compound: equilibrator_cache.Compound*, *abundance: equilibrator_api.ureg.Quantity*)

Set the abundance of the compound.

reset_abundances()

Reset the abundance to standard levels.

set_phase(*compound: equilibrator_cache.Compound*, *phase: str*)

Set the phase of the compound.

get_phased_compound(*compound: equilibrator_cache.Compound*) → Tuple[*equilibrator_api.phased_compound.PhasedCompound*, float]

Get the PhasedCompound object by the Compound object.

get_phase(*compound: equilibrator_cache.Compound*) → str

Get the phase of the compound.

get_abundance(*compound: equilibrator_cache.Compound*) → equilibrator_api.ureg.Quantity

Get the abundance of the compound.

property is_physiological → bool

Check if all concentrations are physiological.

This function is used by eQuilibrator to know if to present the adjusted dG' or not (since the physiological dG' is always displayed and it would be redundant).

Returns

True if all compounds are at physiological abundances.

get_stoichiometry(*compound: equilibrator_cache.Compound*) → float

Get the abundance of the compound.

add_stoichiometry(*compound: equilibrator_cache.Compound*, *coeff: float*) → None

Add to the stoichiometric coefficient of a compound.

If this compound is not already in the reaction, add it.

separate_stored_dg_prime(*p_h: equilibrator_api.ureg.Quantity*, *ionic_strength: equilibrator_api.ureg.Quantity*, *temperature: equilibrator_api.ureg.Quantity*, *p_mg: equilibrator_api.ureg.Quantity*) → Tuple[equilibrator_cache.Reaction, equilibrator_api.ureg.Quantity]

Split the PhasedReaction to aqueous phase and all the rest.

Parameters

- **p_h** –
- **ionic_strength** –
- **temperature** –
- **p_mg** –

Returns

a tuple (residual_reaction, stored_dg_prime) where

residual_reaction is a Reaction object (excluding the compounds that had stored values), and stored_dg_prime is the total dG' of the compounds with stored values (in kJ/mol).

separate_stored_dg() → Tuple[equilibrator_cache.Reaction, equilibrator_api.ureg.Quantity]

Split the PhasedReaction to aqueous phase and all the rest.

Returns

a tuple (residual_reaction, stored_dg) where

residual_reaction is a Reaction object (excluding the compounds that had stored values), and stored_dg is the total dG of the compounds with stored values (in kJ/mol).

dg_correction() → equilibrator_api.ureg.Quantity

Calculate the concentration adjustment in the dG' of reaction.

Returns
the correction for delta G in units of RT

physiological_dg_correction() → equilibrator_api.ureg.Quantity

Calculate the concentration adjustment in the dG' of reaction.

Assuming all reactants are in the default physiological concentrations (i.e. 1 mM)

Returns
the correction for delta G in units of RT

serialize() → List[dict]

Return a serialized version of all the reaction thermo data.

equilibrator_api.reaction_parser

A parser for reaction formulae.

Module Contents

equilibrator_api.reaction_parser.**POSSIBLE_REACTION_ARROWS = ['<=>', '<->', '-->', '<--', '=>', '<=', '->', '<-', '=', '', '', '', '']**

equilibrator_api.reaction_parser.**make_reaction_parser**() → pyparsing.Forward

Build pyparsing-based recursive descent parser for chemical reactions.

Returns
parser

Return type
pyparsing.Forward

Package Contents

equilibrator_api.**default_phase = aqueous**

equilibrator_api.**default_physiological_p_h**

equilibrator_api.**default_physiological_p_mg**

equilibrator_api.**default_physiological_ionic_strength**

equilibrator_api.**default_physiological_temperature**

equilibrator_api.**default_conc_lb**

equilibrator_api.**default_conc_ub**

equilibrator_api.**default_e_potential**

equilibrator_api.**default_rmse_inf**

CHAPTER

FIVE

REFERENCES

PYTHON MODULE INDEX

e

equilibrator_api, 41
equilibrator_api.compatibility, 48
equilibrator_api.component_contribution, 49
equilibrator_api.data, 41
equilibrator_api.model, 41
equilibrator_api.model.bounds, 41
equilibrator_api.model.model, 44
equilibrator_api.phased_compound, 56
equilibrator_api.phased_reaction, 60
equilibrator_api.reaction_parser, 62

INDEX

A

abundance (*equilibrator_api.phased_compound.Condition property*), 56
abundance (*equilibrator_api.phased_compound.PhasedCompound property*), 58
abundance (*equilibrator_api.phased_compound.Proton property*), 59
add_stoichiometry() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
AQUEOUS_PHASE_NAME (*in module equilibrator_api.phased_compound*), 56
atom_bag (*equilibrator_api.phased_compound.PhasedCompound property*), 57
atom_bag (*equilibrator_api.phased_compound.RedoxCarrier property*), 59

B

balance_by_oxidation() (*equilibrator_api.component_contribution.ComponentContribution method*), 51
BaseBounds (*class in equilibrator_api.model.bounds*), 42
bound_df (*equilibrator_api.model.model.StoichiometricModel property*), 46
Bounds (*class in equilibrator_api.model.bounds*), 43
bounds (*equilibrator_api.model.model.StoichiometricModel property*), 46

C

CARBONATE_INCHIS (*in module equilibrator_api.phased_compound*), 56
check_bounds() (*equilibrator_api.model.bounds.Bounds method*), 44
clone() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
ComponentContribution (*class in equilibrator_api.component_contribution*), 49
compound_df (*equilibrator_api.model.model.StoichiometricModel property*), 45
compound_ids (*equilibrator_api.model.model.StoichiometricModel property*), 45
compounds (*equilibrator_api.model.model.StoichiometricModel property*), 45
conc2ln_conc() (*equilibrator_api.model.bounds.BaseBounds static method*), 43
Condition (*class in equilibrator_api.phased_compound*), 56
configure() (*equilibrator_api.model.model.StoichiometricModel method*), 45
copy() (*equilibrator_api.model.bounds.BaseBounds method*), 42
copy() (*equilibrator_api.model.bounds.Bounds method*), 44
create_stoichiometric_matrix_from_reaction_formulas() (*equilibrator_api.component_contribution.ComponentContributi static method*), 55
create_stoichiometric_matrix_from_reaction_objects() (*equilibrator_api.component_contribution.ComponentContributi method*), 55

D

DEFAULT_BOUNDS (*equilibrator_api.model.bounds.Bounds attribute*), 43
default_conc_lb (*in module equilibrator_api*), 62
default_conc_ub (*in module equilibrator_api*), 62
default_e_potential (*in module equilibrator_api*), 62
default_phase (*in module equilibrator_api*), 62
default_physiological_ionic_strength (*in module equilibrator_api*), 62
default_physiological_p_h (*in module equilibrator_api*), 62
default_physiological_p_mg (*in module equilibrator_api*), 62
default_physiological_temperature (*in module equilibrator_api*), 62
default_rmse_inf (*in module equilibrator_api*), 62
dg_analysis() (*equilibra-*

tor_api.component_contribution.ComponentContribution method), 54
dg_correction() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
dg_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 52
dg_prime_sensitivity_to_p_h() (*equilibrator_api.component_contribution.ComponentContribution method*), 53
dgf_prime_sensitivity_to_p_h() (*equilibrator_api.component_contribution.ComponentContribution method*), 53
dimensionality (*equilibrator_api.phased_compound.Condition property*), 57

E

e_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 54
equilibrator_api
    module, 41
equilibrator_api.compatibility
    module, 48
equilibrator_api.component_contribution
    module, 49
equilibrator_api.data
    module, 41
equilibrator_api.model
    module, 41
equilibrator_api.model.bounds
    module, 41
equilibrator_api.model.model
    module, 44
equilibrator_api.phased_compound
    module, 56
equilibrator_api.phased_reaction
    module, 60
equilibrator_api.reaction_parser
    module, 62

F

find_most_abundant_ms() (*in module equilibrator_api.component_contribution*), 49
formula (*equilibrator_api.phased_compound.PhasedCompound property*), 57
from_csv() (*equilibrator_api.model.bounds.Bounds class method*), 44
from_network_sbtab() (*equilibrator_api.model.model.StoichiometricModel class method*), 47
from_sbtab() (*equilibrator_api.model.model.StoichiometricModel class method*), 47

G

GAS_PHASE_NAME (*in module equilibrator_api.phased_compound*), 56
get_abundance() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
get_bound_tuple() (*equilibrator_api.model.bounds.BaseBounds method*), 42
get_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 42
get_bounds() (*equilibrator_api.model.model.StoichiometricModel method*), 46
get_compound() (*equilibrator_api.component_contribution.ComponentContribution method*), 50
get_compound_by_inchi() (*equilibrator_api.component_contribution.ComponentContribution method*), 50
get_default() (*equilibrator_api.phased_compound.PhasedCompound static method*), 57
get_default_bounds() (*equilibrator_api.model.bounds.Bounds static method*), 44
get_element_data_frame() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
get_ln_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 43
get_ln_lower_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 43
get_ln_upper_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 43
get_lower_bound() (*equilibrator_api.model.bounds.BaseBounds method*), 42
get_lower_bound() (*equilibrator_api.model.bounds.Bounds method*), 44
get_lower_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 42
get_oxidation_reaction() (*equilibrator_api.component_contribution.ComponentContribution method*), 51
get_phase() (*equilibrator_api.phased_reaction.PhasedReaction*

method), 61
get_phased_compound() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
get_stoichiometry() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
get_stored_microspecie() (*equilibrator_api.phased_compound.PhasedCompound method*), 58
get_stored_standard_dgf() (*equilibrator_api.phased_compound.PhasedCompound method*), 58
get_stored_standard_dgf() (*equilibrator_api.phased_compound.RedoxCarrier method*), 59
get_stored_standard_dgf_prime() (*equilibrator_api.phased_compound.PhasedCompound method*), 58
get_stored_standard_dgf_prime() (*equilibrator_api.phased_compound.RedoxCarrier method*), 59
get_upper_bound() (*equilibrator_api.model.bounds.BaseBounds method*), 42
get_upper_bound() (*equilibrator_api.model.bounds.Bounds method*), 44
get_upper_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 42

H

hash_md5() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
html_formula (*equilibrator_api.phased_compound.PhasedCompound property*), 58

I

id (*equilibrator_api.phased_compound.PhasedCompound property*), 57
inchi (*equilibrator_api.phased_compound.PhasedCompound property*), 57
inchi_key (*equilibrator_api.phased_compound.PhasedCompound property*), 57
initialize_custom_version() (*equilibrator_api.component_contribution.ComponentContribution static method*), 50
ionic_strength (*equilibrator_api.component_contribution.ComponentContribution property*), 49
is_physiological (*equilibrator_api.phased_compound.Condition property*), 57
is_physiological (*equilibrator_api.phased_compound.PhasedCompound property*), 58
is_physiological (*equilibrator_api.phased_compound.RedoxCarrier property*), 59
is_physiological (*equilibrator_api.phased_reaction.PhasedReaction property*), 61
is_using_group_contribution() (*equilibrator_api.component_contribution.ComponentContribution method*), 54

L

legacy() (*equilibrator_api.component_contribution.ComponentContribut static method*), 49
LIQUID_PHASE_NAME (*in module equilibrator_api.phased_compound*), 56
ln_abundance (*equilibrator_api.phased_compound.Condition property*), 57
ln_abundance (*equilibrator_api.phased_compound.PhasedCompound property*), 58
ln_abundance (*equilibrator_api.phased_compound.Proton property*), 59
ln_abundance (*equilibrator_api.phased_compound.RedoxCarrier property*), 59
ln_conc_lb (*equilibrator_api.model.model.StoichiometricModel property*), 46
ln_conc_mu (*equilibrator_api.model.model.StoichiometricModel property*), 46
ln_conc_sigma (*equilibrator_api.model.model.StoichiometricModel property*), 46
ln_conc_ub (*equilibrator_api.model.model.StoichiometricModel property*), 46
ln_physiological_abundance (*equilibrator_api.phased_compound.Condition property*), 57
ln_physiological_abundance (*equilibrator_api.phased_compound.PhasedCompound property*), 58
ln_physiological_abundance (*equilibrator_api.phased_compound.Proton property*), 59
ln_physiological_abundance (*equilibrator_api.phased_compound.RedoxCarrier*

property), 59
ln_reversibility_index() (*equilibrator_api.component_contribution.ComponentContribution method*), 54

M

make_reaction_parser() (*in module equilibrator_api.reaction_parser*), 62
map_cobra_reactions() (*in module equilibrator_api.compatibility*), 48
mass (*equilibrator_api.phased_compound.PhasedCompound property*), 58
MicroSpecie (*in module equilibrator_api.phased_compound*), 56
MINIMAL_STDEV (*equilibrator_api.model.model.StoichiometricModel attribute*), 45
module
    equilibrator_api, 41
    equilibrator_api.compatibility, 48
    equilibrator_api.component_contribution, 49
    equilibrator_api.data, 41
    equilibrator_api.model, 41
    equilibrator_api.model.bounds, 41
    equilibrator_api.model.model, 44
    equilibrator_api.phased_compound, 56
    equilibrator_api.phased_reaction, 60
    equilibrator_api.reaction_parser, 62
multicompartmental_standard_dg_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 55

N

NON_AQUEOUS_COMPOUND_DICT (*in module equilibrator_api.phased_compound*), 56

O

open_sbtabdoc() (*in module equilibrator_api.model*), 48

P

p_h (*equilibrator_api.component_contribution.ComponentContribution property*), 49
p_mg (*equilibrator_api.component_contribution.ComponentContribution property*), 49
parse_formula() (*equilibrator_api.component_contribution.ComponentContribution static method*), 55
parse_formula_side() (*equilibrator_api.component_contribution.ComponentContribution static method*), 55
parse_reaction_formula() (*equilibrator_api.component_contribution.ComponentContribution method*), 51
phase (*equilibrator_api.phased_compound.Condition property*), 56
phase (*equilibrator_api.phased_compound.PhasedCompound property*), 58
PHASE_INFO_DICT (*in module equilibrator_api.phased_compound*), 56
phase_shorthand (*equilibrator_api.phased_compound.PhasedCompound property*), 58
PHASED_COMPOUND_DICT (*in module equilibrator_api.phased_compound*), 56
PhasedCompound (*class in equilibrator_api.phased_compound*), 57
PhasedReaction (*class in equilibrator_api.phased_reaction*), 60
PhaseInfo (*in module equilibrator_api.phased_compound*), 56
physiolical_abundance (*equilibrator_api.phased_compound.Condition property*), 57
physiological_dg_correction() (*equilibrator_api.phased_reaction.PhasedReaction method*), 62
physiological_dg_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 53
physiological_e_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 54
possible_phases (*equilibrator_api.phased_compound.PhasedCompound property*), 58
POSSIBLE_REACTION_ARROWS (*in module equilibrator_api.reaction_parser*), 62
predict_protons_and_charge() (*in module equilibrator_api.component_contribution*), 49
Proton (*class in equilibrator_api.phased_compound*), 59

R

REACTION_COUNTER (*equilibrator_api.phased_reaction.PhasedReaction attribute*), 60
reaction_df (*equilibrator_api.model.model.StoichiometricModel property*), 45
reaction_formulas (*equilibrator_api.model.model.StoichiometricModel property*), 45
reaction_ids (*equilibrator_api.model.model.StoichiometricModel property*), 45

reactions (*equilibrator_api.model.model.StoichiometricModel property*), 45
read_thermodynamics() (*equilibrator_api.model.model.StoichiometricModel static method*), 47
REDOX_PHASE_NAME (*in module equilibrator_api.phased_compound*), 56
RedoxCarrier (*class in equilibrator_api.phased_compound*), 59
reset_abundance() (*equilibrator_api.phased_compound.Condition method*), 57
reset_abundances() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
reverse() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
RT (*equilibrator_api.component_contribution.ComponentContribution property*), 51

S

search_compound() (*equilibrator_api.component_contribution.ComponentContribution method*), 50
search_compound_by_inchi_key() (*equilibrator_api.component_contribution.ComponentContribution method*), 50
search_reaction() (*equilibrator_api.component_contribution.ComponentContribution method*), 51
separate_stored_dg() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
separate_stored_dg_prime() (*equilibrator_api.phased_reaction.PhasedReaction method*), 61
serialize() (*equilibrator_api.phased_compound.PhasedCompound method*), 59
serialize() (*equilibrator_api.phased_reaction.PhasedReaction method*), 62
set_abundance() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
set_bounds() (*equilibrator_api.model.bounds.BaseBounds method*), 43
set_bounds() (*equilibrator_api.model.model.StoichiometricModel method*), 45
set_phase() (*equilibrator_api.phased_reaction.PhasedReaction method*), 60
smiles (*equilibrator_api.phased_compound.PhasedCompound property*), 57
SOLID_PHASE_NAME (*in module equilibrator_api.phased_compound*), 56
standard_abundance (*equilibrator_api.phased_compound.Condition property*), 56
standard_dg() (*equilibrator_api.component_contribution.ComponentContribution method*), 52
standard_dg_formation() (*equilibrator_api.component_contribution.ComponentContribution method*), 51
standard_dg_multi() (*equilibrator_api.component_contribution.ComponentContribution method*), 52
standard_dg_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 52
standard_dg_prime_multi() (*equilibrator_api.component_contribution.ComponentContribution method*), 52
standard_e_prime() (*equilibrator_api.component_contribution.ComponentContribution method*), 54
StoichiometricModel (*class in equilibrator_api.model.model*), 44

T

temperature (*equilibrator_api.component_contribution.ComponentContribution property*), 49
to_data_frame() (*equilibrator_api.model.bounds.Bounds method*), 44
to_phased_compound() (*equilibrator_api.phased_reaction.PhasedReaction static method*), 60
to_sbtab() (*equilibrator_api.model.model.StoichiometricModel method*), 48

U

update_standard_dgs() (*equilibrator_api.model.model.StoichiometricModel method*), 45

W

write_sbtab() (*equilibrator_api.model.model.StoichiometricModel method*), 48