equilibrator

Elad Noor

Apr 14, 2021

CONTENTS

`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 confidence intervals).
- 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

You can find a simple example in the Example Usage section. Or take a look at the *API Reference* for a full list of classes and functions.

4.1 Installation

4.1.1 Step 1

The easiest way to get eQuilibrator-API up and running is using virtualenv, PyPI, and Jupyter notebooks

```
virtualenv -p python3 equilibrator
source equilibrator/bin/activate
pip install equilibrator-api jupyter
```

If you are using a Windows environment, 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:

```
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 >= 3.6
- equilibrator-cache
- component-contribution
- numpy
- scipy
- pandas
- python-Levenshtein-wheels
- pint
- path
- appdirs
- diskcache
- httpx
- tenacity
- periodictable
- uncertainties
- pyzenodo3
- python-slugify
- cobra (optional)

4.2 Code examples

We start by importing the necessary packages

[1]:
```
import numpy
import matplotlib.pyplot as plt
from equilibrator_api import ComponentContribution, Q_
```

4.2.1 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()
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]:
```
reaction1 = 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 reaction1.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(reaction1)
print(f"G'° = {dG0_prime}")

dGm_prime = cc.physiological_dg_prime(reaction1)
print(f"G'm = {dGm_prime}")

reaction1.set_abundance(cc.get_compound("bigg.metabolite:atp"), Q_("1 mM"))
reaction1.set_abundance(cc.get_compound("bigg.metabolite:adp"), Q_("100 uM"))
reaction1.set_abundance(cc.get_compound("bigg.metabolite:pi"), Q_("0.003 M"))

dG_prime = cc.dg_prime(reaction1)
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} +/- "
```

(continues on next page)

(continued from previous page)

```
    f"{dG_prime_error_in_kj_per_mol:.1f} kJ/mol"
)
```

```
G'° = -49.2 +/- 0.3 kJ/mol
```

4.2.2 The reversibility index

You can also calculate the reversibility index for this reaction.

[7]:
```
print(f"ln(Reversibility Index) = {cc.ln_reversibility_index(reaction1)}")
```

```
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.

4.2.3 Further examples for reaction parsing

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

[8]:
```
reaction2 = cc.parse_reaction_formula(
    "kegg:C00002 + CHEBI:15377 = metanetx.chemical:MNXM7 + bigg.metabolite:pi"
)
dG0_prime = cc.standard_dg_prime(reaction2)
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:

[9]:
```
reaction3 = cc.search_reaction("beta-D-glucose = glucose")
for cpd, coeff in reaction3.items():
    print(f"{coeff:5.0f}  {cpd.get_common_name():15s} {cpd.inchi_key}")
dG0_prime = cc.standard_dg_prime(reaction3)
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.

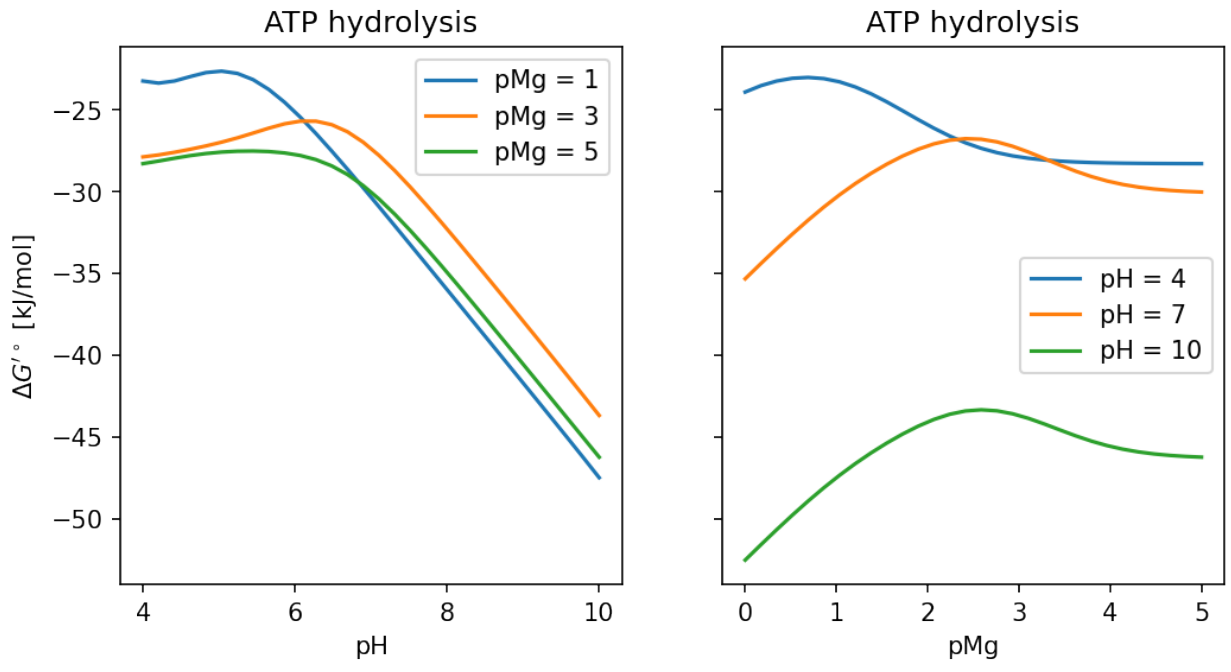
4.2.4 *G'°* of ATP hydrolysis as a function of pH and pMg

[10]:
```
def calc_dg(p_h, p_mg):
    cc.p_h = Q_(p_h)
    cc.p_mg = Q_(p_mg)
    return cc.standard_dg_prime(reaction1).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();
```



4.3 API Reference

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

4.3.1 equilibrator_api

Subpackages

equilibrator_api.data

equilibrator_api.model

Submodules

equilibrator_api.model.bounds

Define lower and upper bounds on compounds.

Module Contents

class equilibrator_api.model.bounds.**BaseBounds**
Bases: object

A base class for declaring bounds on things.

abstract copy(*self*)
Return a (deep) copy of self.

abstract get_lower_bound(*self*, *compound: equilibrator_cache.Compound*)
Get the lower bound for this key.

Parameters **key** – a compound

abstract get_upper_bound(*self*, *compound: equilibrator_cache.Compound*)
Get the upper bound for this key.

Parameters **key** – a compound

get_lower_bounds(*self*, *compounds: Iterable[equilibrator_cache.Compound]*) → Iterable[equilibrator_api.Q_]
Get the bounds for a set of keys in order.

Parameters **compounds** – an iterable of Compounds

Returns an iterable of the lower bounds

get_upper_bounds(*self*, *compounds: Iterable[equilibrator_cache.Compound]*) → Iterable[equilibrator_api.Q_]
Get the bounds for a set of keys in order.

Parameters **compounds** – an iterable of Compounds

Returns an iterable of the upper bounds

[1] Created with sphinx-autoapi

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

Get both upper and lower bounds for this key.

Parameters **compound** – a Compound object

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

get_bounds(*self*, *compounds: Iterable[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(*self*, *compounds: Iterable[equilibrator_cache.Compound]*) → Tuple[Iterable[float], Iterable[float]]

Get the log-bounds for a set of compounds.

Parameters **compounds** – an iterable of Compounds

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

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

Get the log lower bounds for a set of compounds.

Parameters **compounds** – an iterable of Compounds

Returns an iterable of log lower bounds

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

Get the log upper bounds for a set of compounds.

Parameters **compounds** – an iterable of Compounds

Returns an iterable of log upper bounds

set_bounds(*self*, *compound: equilibrator_cache.Compound*, *lb: equilibrator_api.Q_*, *ub: equilibrator_api.Q_*) → None

Set bounds for a specific key.

Parameters **key** – a string representation of a KEGG compound ID,

i.e. C00001 for water :param lb: the lower bound value :param ub: the upper bound value

class equilibrator_api.model.bounds.**Bounds**(*lower_bounds: Dict[equilibrator_cache.Compound, equilibrator_api.Q_] = None*, *upper_bounds: Dict[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(*cls*, *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(*self*) → pandas.DataFrame

Convert the list of bounds to a Pandas DataFrame.

check_bounds(*self*) → None

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

copy(*self*) → *Bounds*

Return a deep copy of self.

get_lower_bound(*self*, *compound: equilibrator_cache.Compound*) → equilibrator_api.Q_

Get the lower bound for this compound.

get_upper_bound(*self*, *compound: 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) –

Returns

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**(*reactions: List[equilibrator_api.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, compound_id_mapping: Callable[[equilibrator_cache.Compound], 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(*self*) → None
Configure the Component Contribution aqueous conditions.

update_standard_dgs(*self*) → None
Calculate the standard G' values and uncertainties.

Use the Component Contribution method.

set_compound_ids(*self*, *mapping: Callable[[equilibrator_cache.Compound], str] = None*) → None
Use alternative compound names for outputs such as plots.

Parameters **mapping** (*callable, optional*) – a function mapping compounds to their names in the model

set_bounds(*self*, *compound_id: 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(*self*, *compound_id: 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 (*self*) → Tuple[Iterable[equilibrator_api.Q_], Iterable[equilibrator_api.Q_]]
Get the concentration bounds.

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

Returns

Return type tuple of (lower bounds, upper bounds)

property ln_conc_lb (*self*) → numpy.array
Get the log lower bounds on the concentrations.

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

Returns

Return type a NumPy array of the log lower bounds

property ln_conc_ub (*self*) → numpy.ndarray
Get the log upper bounds on the concentrations.

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

Returns

Return type a NumPy array of the log upper bounds

property ln_conc_mu (*self*) → numpy.array
Get the mean of the log concentration distribution based on the bounds.

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

Returns

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

property ln_conc_sigma (*self*) → numpy.array
Get the stdev of the log concentration distribution based on the bounds.

Returns

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

static get_compounds (*reactions: Iterable[equilibrator_api.Reaction]*) → List[equilibrator_cache.Compound]
Get a unique list of all compounds in all reactions.

Parameters **reactions** – an iterator of reactions

Returns a list of unique compounds

property reaction_ids (*self*) → Iterable[str]
Iterate through all the reaction IDs.

Returns the reaction IDs

property reaction_formulas (*self*) → Iterable[str]
Iterate through all the reaction formulas.

Returns the reaction formulas

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

Returns

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

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

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*)

Returns

Return type a Pathway object

classmethod from_sbtab(*cls*, *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(*self*) → equilibrator_api.model.SBtabDocument

Export the model to an SBtabDocument.

write_sbtab(*self*, *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.

Other Parameters **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_abundance_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.PhasedRea*, *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.Compo = None*, *predictor: Optional[component_contribution.p = None*)

Bases: object

A wrapper class for GibbsEnergyPredictor.

Also holds default conditions for compounds in the different phases.

property p_h (*self*) → equilibrator_api.Q_
Get the pH.

property p_mg (*self*) → equilibrator_api.Q_
Get the pMg.

property ionic_strength (*self*) → equilibrator_api.Q_
Get the ionic strength.

property temperature (*self*) → 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.

Returns

Return type A ComponentContribution object

static initialize_custom_version (*rmse_inf: equilibrator_api.Q_ = default_rmse_inf, zenodo_doi_cache: str = DEFAULT_ZENODO_DOI, zenodo_doi_params: str = ZENODO_DOI_PARAMETERS*) → *ComponentContribution*
Initialize a ComponentContribution object with custom quilt 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)
- **zenodo_doi_cache** (*str, optional*) – (Default value: "10.5281/zenodo.4128543")
- **zenodo_doi_params** (*str, optional*) – (Default value: "10.5281/zenodo.4013789")

Returns

Return type A ComponentContribution object

get_compound (*self*, *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 (*self*, *inchi: str*) → Union[equilibrator_cache.Compound, None]
Get a Compound using InChI.

Returns cpd

Return type Compound

search_compound_by_inchi_key (*self*, *inchi_key: str*) → List[equilibrator_cache.Compound]
Get a Compound using InChI.

Returns **cpd**

Return type Compound

search_compound (*self*, *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 (*self*, *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 Reaction

search_reaction (*self*, *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 Reaction

balance_by_oxidation (*self*, *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 (*self*, *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 (*self*) → equilibrator_api.Q_
Get the value of RT.

standard_dg_formation (*self*, *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 Gibbs energy of formation
- **sigma** (*array*) – the uncertainty vector

standard_dg (*self*, *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 (*self*, *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. '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 (*self*, *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 (*self*, *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 (*self*, *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. '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(*self*, *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).

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

Get the analysis of the component contribution estimation process.

Returns

Return type the analysis results as a list of dictionaries

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

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

Returns

Return type true iff group contribution is needed

multicompartmental_standard_dg_prime (*self*, *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*) → 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

Returns **standard_dg_prime** – the transport reaction Gibbs free energy change

Return type Measurement

create_stoichiometric_matrix(*self*, *reactions: Iterable[*equilibrator_api.phased_reaction.PhasedReaction*]*) → pandas.DataFrame

Build a stoichiometric matrix.

Parameters **reactions** (*Iterable[Reaction]*) – 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.**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**

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(*self*) → str
Return the phase.

property abundance(*self*) → equilibrator_api.ureg.Quantity
Return the abundance.

property standard_abundance(*self*) → equilibrator_api.ureg.Quantity
Return the standard abundance in this phase.

property physiolical_abundance(*self*) → equilibrator_api.ureg.Quantity
Return the default physiological abundance in this phase.

property dimensionality(*self*) → 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(*self*) → float
Return the log of the ratio between given and std abundances.

property ln_physiological_abundance(*self*) → float
Return the log of the ratio between phys and std abundances.

reset_abundance(*self*) → None
Reset the abundance to standard abundance.

property is_physiological(*self*) → 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 inchi(*self*) → str
Get the compound's InChI.

property inchi_key(*self*) → str
Get the compound's InChIKey.

property is_proton(*self*) → bool
Return True if this compound is a proton.

property id(*self*) → int
Get the compound's equilibrator internal ID.

property formula(*self*) → str
Get the chemical formula.

property html_formula(*self*) → str
Get the chemical formula.

property mass(*self*) → float
Get the chemical molecular mass.

property phase(*self*) → str
Get the phase.

property phase_shorthand(*self*) → str
Get the phase shorthand (i.e. 'l' for liquid).

property possible_phases(*self*) → Tuple[str]
Get the possible phases for this compound.

property abundance(*self*) → equilibrator_api.ureg.Quantity
Get the abundance.

property ln_abundance(*self*) → float
Return the log of the abundance (for thermodynamic calculations).

property ln_physiological_abundance (*self*) → float
Return the log of the default physiological abundance.

property is_physiological (*self*) → bool
Check if the abundance is physiological.

get_stored_standard_dgf_prime (*self*, *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 (*self*) → 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 (*self*) → *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 (*self*) → dict
Return a serialized version of all the compound thermo data.

Returns a list of dictionaries with all the microspecies data

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.PhasedComp *float] = None*)

Bases: equilibrator_cache.Reaction

A daughter class of Reaction that adds phases and abundances.

REACTION_COUNTER = 0

clone(*self*) → *PhasedReaction*
Clone this reaction object.

reverse(*self*) → *PhasedReaction*
Return a PhasedReaction with the reverse reaction.

hash_md5(*self*, *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(*self*, *compound: equilibrator_cache.Compound*, *abundance: equilibrator_api.ureg.Quantity*)
Set the abundance of the compound.

reset_abundances(*self*)
Reset the abundance to standard levels.

set_phase(*self*, *compound: equilibrator_cache.Compound*, *phase: str*)
Set the phase of the compound.

get_phased_compound(*self*, *compound: equilibrator_cache.Compound*) → Tuple[*equilibrator_api.phased_compound.PhasedCompound*, float]
Get the PhasedCompound object by the Compound object.

get_phase(*self*, *compound: equilibrator_cache.Compound*) → str
Get the phase of the compound.

get_abundance(*self*, *compound: equilibrator_cache.Compound*) → equilibrator_api.ureg.Quantity
Get the abundance of the compound.

property is_physiological(*self*) → 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 (*self*, *compound: equilibrator_cache.Compound*) → float
Get the abundance of the compound.

add_stoichiometry (*self*, *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 (*self*, *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 (*self*) → 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 (*self*) → 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 (*self*) → 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 (*self*) → 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

INDEX

F

G

N

O

P

R

S

T

U

W