pyEQL Documentation

Release v0.8.1

Ryan Kingsbury

Oct 03, 2023

CONTENTS

1 Description **3**

1.1 Key Features 3

Python Module Index **29**

Index **31**

CHAPTER

ONE

DESCRIPTION

pyEQL is a Python library that provides tools for modeling aqueous electrolyte solutions. It allows the user to manipulate solutions as Python objects, providing methods to populate them with solutes, calculate species-specific properties (such as activity and diffusion coefficients), and retrieve bulk properties (such as density, conductivity, or volume).

pyeql-demo.png

pyEQL is designed to be customizable and easy to integrate into projects that require modeling of chemical thermodyanmics of aqueous solutions. It aspires to provide a flexible, extensible framework for the user, with a high level of transparency about data sources and calculation methods.

pyEQL runs on Python 3.8+ and is licensed under LGPL.

1.1 Key Features

- Build accurate solution properties using a minimum of inputs. Just specify the identity and quantity of a solute and pyEQL will do the rest.
- "Graceful Decay" from more sophisticated, data-intensive modeling approaches to simpler, less accurate ones depending on the amount of data supplied.
- Not limited to dilute solutions. pyEQL contains out of the box support for the Pitzer Model and other methods for modeling concentrated solutions.
- Extensible database system that allows one to supplement pyEQL's default parameters with project-specific data.
- Units-aware calculations (by means of the pint library)

Contents:

1.1.1 Installation

Use a conda environment

We highly recommend installing python in an isolated environment using conda (or its speedier, backward-compatible successor, mamba). In particular, we recommend the miniforge or mambaforge distributions of Python, which are lightweight distributions of conda that automatically activate the `conda-forge` channel for up-to-date scientific packages.

Note: If you are on a Windows machine, we recommend you install the Windows Subsystem for Linux (WSL) and set up your conda environments inside the WSL environment.

After installing `conda` / `mamba`, follow their instructions to create an environment. The steps should be similar to the following:

1. Open your terminal (or "Anaconda prompt" or "Miniforge prompt" on Windows)
2. Pick a name for your environment (note: you can create many environments if you want)
3. type `conda create -n <name-you-picked> python=3.10` (if you install miniforge) or `mamba create -n <name-you-picked> python=3.10` (if you installed mambaforge) and press enter
4. After the environment is installed, type `conda activate <name-you-picked> / mamba activate <name-you-picked>` and press enter

pip install

Once Python is installed and your environment is activated you can install `pyEQL` from PyPi by typing the following command:

```
pip install pyEQL
```

This should automatically pull in the required *dependencies* as well.

Important: If you are NOT using a `conda` environment, may have to run 'pip3' rather than 'pip'. This will be the case if Python 2.x and Python 3.x are installed side-by-side on your system. You can tell if this is the case by typing the following command:

```
$ python --version
Python 2.7.12
```

This means Python 2.x is installed. If you run 'pip install' it will point to the Python 2.7 installation, but pyEQL only works on Python 3. So, try this:

```
$ python3 --version
Python 3.9.7
```

To get to Python 3.x, you have to type 'python3'. In this case, you would run 'pip3 install'

Other dependencies

pyEQL also requires the following packages:

- pint - for automated unit conversion
- pymatgen - used to interpret chemical formulas
- iapws - used to calculate the properties of water
- monty - used for saving and loading `Solution` objects to files
- maggma - used by the internal property database
- scipy
- numpy

If you use pip to install pyEQL (recommended), they should be installed automatically.

Installing the development branch

If you want to use the bleeding edge (and potentially unstable!) development branch instead of the latest stable release, you can substitute the following for the above 'pip install' command:

```
pip install git+https://github.com/rkingsbury/pyEQL.git@develop
```

Manually install via Git

Simply navigate to a directory of your choice on your computer and clone the repository by executing the following terminal command:

```
git clone https://github.com/rkingsbury/pyEQL
```

Then install by executing:

```
pip install -e pyEQL
```

Note: You may have to run 'pip3' rather than 'pip'. See the note in the *pip install* section.

1.1.2 Tutorial

pyEQL creates a new type (`Solution` class) to represent a chemical solution. It also comes pre-loaded with a database of diffusion coefficients, activity correction parameters, and other data on a variety of common electrolytes. Virtually all of the user-facing functions in `pyEQL` are accessed through the `Solution` class.

Creating a Solution Object

Create a Solution object by invoking the Solution class:

```
>>> import pyEQL
>>> s1 = pyEQL.Solution()
>>> s1
<pyEQL.pyEQL.Solution at 0x7f9d188309b0>
```

If no arguments are specified, pyEQL creates a 1-L solution of water at pH 7 and 25 degC.

More usefully, you can specify solutes and bulk properties:

```
>>> s2 = pyEQL.Solution({'Na+':'0.5 mol/kg', 'Cl-': '0.5 mol/kg'}, pH=8, temperature =
↪'20 degC', volume='8 L')
```

Retrieving Solution Properties

Bulk Solution Properties

pyEQL provides a variety of methods to calculate or look up bulk properties like temperature, ionic strength, conductivity, and density.

```
>>> s2.volume
8.071524653929277 liter
>>> s2.density
1.0182802742389558 kilogram/liter
>>> s2.conductivity
4.083570230022633 siemens/meter
>>> s2.ionic_strength
0.500000505903012 mole/kilogram
```

Individual Solute Properties

You can also retrieve properties for individual solutes (or the solvent, water)

```
>>> s2.get_amount('Na+','mol/L')
0.4946847550064916 mole/liter
>>> s2.get_activity_coefficient('Na+)
0.6838526233869155
>>> s2.get_activity('Na+')
0.3419263116934578
>>> s2.get_property('Na+','transport.diffusion_coefficient')
1.1206048116287536e-05 centimeter2/second
```

Units-Aware Calculations using pint

pyEQL uses pint to perform units-aware calculations. The pint library creates Quantity objects that contain both a magnitude and a unit.

```
>>> from pyEQL import ureg
>>> test_qty = pyEQL.ureg('1 kg/m**3')
1.0 kilogram/meter3
```

Many `pyEQL` methods require physical quantities to be input as strings, then these methods return pint `Quantity` objects. A string quantity must contain both a magnitude and a unit (e.g. '0.5 mol/L'). In general, pint recognizes common abbreviations and SI prefixes. Compound units must follow Python math syntax (e.g. cm**2 not cm2).

Pint `Quantity` objects have several useful attributes. They can be converted to strings:

```
>>> str(test_qty)
'1.0 kg/m**3'
```

the magnitude, units, or dimensionality can be retrieved via attributes:

```
>>> test_qty.magnitude
1.0
>>> test_qty.units
<UnitsContainer({'kilogram': 1.0, 'meter': -3.0})>
>>> test_qty.dimensionality
<UnitsContainer({'[length]': -3.0, '[mass]': 1.0})>
```

See the pint documentation for more details on creating and manipulating `Quantity` objects.

Using `pyEQL` in your projects

To access pyEQL's main features in your project all that is needed is an import statement:

```
>>> import pyEQL
```

In order to directly create `Quantity` objects, you need to explicitly import `UnitRegistry`, which you can do as follows:

```
>>> from pyEQL import ureg
>>> test_qty = ureg('1 kg/m**3')
1.0 kilogram/meter3
```

Note: Note that the meaning of `ureg` is equivalent in the above `pyEQL` examples and in the pint documentation. `pyEQL` instantiates its own `UnitRegistry` (with custom definitions for solution chemistry) and assigns it to the variable `ureg`. In most `pint` examples, the line `ureg = UnitRegistry()` does the same thing.

Warning: if you use `pyEQL` in conjunction with another module that also uses pint for units-aware calculations, you must convert all `Quantity` objects to strings before passing them to the other module, as pint cannot perform mathematical operations on units that belong to different "registries." See the pint documentation for more details.

1.1.3 The Solution Class

The `Solution` class defines a pythonic interface for **creating**, **modifying**, and **estimating properties** of electrolyte solutions. It is the core feature of `pyEQL` and the primary user-facing class.

Creating a solution

A `Solution` created with no arguments will default to pure water at pH=7, T=25 degrees Celsius, and 1 atm pressure.

```
>>> from pyEQL import Solution
>>> s1 = Solution()
>>> s1.pH
6.998877352386266
```

Alternatively, you can use the `autogenerate()` function to easily create common solutions like seawater:

```
>>> from pyEQL.functions import autogenerate
>>> s2 = autogenerate('seawater')
<pyEQL.solution.Solution object at 0x7f057de6b0a0>
```

You can inspect the solutes present in the solution via the `components` attribute. This comprises a dictionary of solute formula: moles, where 'moles' is the number of moles of that solute in the `Solution`. Note that the solvent (water) is present in `components`, too.

```
>>> s2.components
{'H2O': 55.34455401423017,
 'H+': 7.943282347242822e-09,
 'OH-': 8.207436858780226e-06,
 'Na+': 0.46758273714962967,
 'Mg+2': 0.052661180523467986,
 'Ca+2': 0.010251594148212318,
 'K+': 0.010177468379526856,
 'Sr+2': 9.046483353663286e-05,
 'Cl-': 0.54425785619973,
 'SO4-2': 0.028151873448454025,
 'HCO3-': 0.001712651176926199,
 'Br-': 0.0008395244921424563,
 'CO3-2': 0.00023825904349479546,
 'B(OH)4': 0.0001005389715937341,
 'F-': 6.822478260456777e-05,
 'B(OH)3': 0.0003134669156396757,
 'CO2': 9.515218476861175e-06
 }
```

To get the amount of a specific solute, use `get_amount()` and specify the units you want:

```
>>> s2.get_amount('Na+', 'g/L')
<Quantity(10.6636539, 'gram / liter')>
```

Finally, you can manually create a solution with any list of solutes, temperature, pressure, etc. that you need:

```
>>> from pyEQL import Solution
>>> s1 = Solution(solutes={'Na+':'0.5 mol/kg', 'Cl-': '0.5 mol/kg'},
```

(continues on next page)

(continued from previous page)

```
                pH=8,
                temperature = '20 degC',
                volume='8 L'
                )
```

Class reference

The remainder of this page contains detailed information on each of the methods, attributes, and properties in `Solution`. Use the sidebar on the right for easier navigation.

1.1.4 Chemical Formulas

pyEQL interprets the chemical formula of a substance to calculate its molecular weight and formal charge. The formula is also used as a key to search the database for parameters (e.g. diffusion coefficient) that are used in subsequent calculations.

How to Enter Valid Chemical Formulas

Generally speaking, type the chemical formula of your solute the "normal" way and `pyEQL` should be able to inerpret it. Internally, `pyEQL` uses a utility function `pyEQL.utils.standardize_formula` to process all formulas into a standard form. At present, this is done by passing the formula through the `pymatgen.core.ion.Ion` class. Anything that the `Ion` class can understand will be processed into a valid formula by `pyEQL`.

Here are some examples:

Substance	You enter	`pyEQL` understands
Sodium Chloride	"NaCl", "NaCl(aq)", or "ClNa"	"NaCl(aq)"
Sodium Sulfate	"Na(SO4)2" or "NaS2O8"	"Na(SO4)2(aq)"
Sodium Ion	"Na+", "Na+1", "Na1+", or "Na[+]"	"Na[+1]"
Magnesium Ion	"Mg+2", "Mg++", or "Mg[++]"	"Mg[+2]"
Methanol	"CH3OH", "CH4O"	"'CH3OH(aq)'"

Specifically, `standardize_formula` uses `Ion.from_formula(<formula>).reduced_formla` (shown in the right hand column of the table) to identify solutes. Notice that for charged species, the charges are always placed inside square brackets (e.g., `Na[+1]`) and always include the charge number (even for monovalent ions). Uncharged species are always suffixed by `(aq)` to disambiguate them from solids.

Important: When writing multivalent ion formulas, it is strongly recommended that you put the charge number AFTER the + or - sign (e.g., type "Mg+2" NOT "Mg2+"). The latter formula is ambiguous - it could mean 'Mg_2^+' or 'Mg^{+2}' and it will be processed incorrectly into `Mg[+0.5]`

Manually testing a formula

If you want to make sure pyEQL is understanding your formula correctly, you can manually test it as follows:

```
>>> from pyEQL.utils import standardize_formula
>>> standardize_formula(<your_formula>)
...
```

Formulas you will see when using `Solution`

When using the `Solution` class,

- When creating a `Solution`, you can enter chemical formulas in any format you prefer, as long as `standardize_formula` can understand it (see *manual testing*).
- The keys (solute formulas) in `Solution.components` are standardized. So if you entered `Na+` for sodium ion, it will appear in `components` as `Na[+1]`.
- However, the `components` attribute is a special dictionary that automatically standardizes formulas when accessed. So, you can still enter the formula however you want. For example, the following all access or modify the same element in `components`:

  ```
  Solution.components.get('Na+')
  Solution.components["Na+1"]
  Solution.components.update("Na[+]": 2)
  Solution.components["Na[+1]"]
  ```

- Arguments to `Solution.get_property` can be entered in any format you prefer. When `pyEQL` queries the database, it will automatically standardize the formula.
- Property data in the database is uniquely identified by the standardized ion formula (output of `Ion.from_formula(<formula>).reduced_formla`, e.g. "Na[+1]" for sodium ion).

1.1.5 Property Database

pyEQL is distributed with a database of solute properties and model parameters needed to perform it's calculations. The database includes:

- Molecular weight, charge, and other chemical informatics information for any species
- Diffusion coefficients for 104 ions
- Pitzer model activity correction coefficients for 157 salts
- Pitzer model partial molar volume coefficients for 120 salts
- Jones-Dole "B" coefficients for 83 ions
- Hydrated and ionic radii for 23 ions
- Dielectric constant model parameters for 18 ions
- Partial molar volumes for 24 ions

`pyEQL` can automatically infer basic chemical informatics such as molecular weight and charge by passing a solute's formula to `pymatgen.core.ion.Ion` (See *chemical formulas*). For other physicochemical properties, it relies on data compiled into the included database. A list of the data and species covered is available *below*

Format

The database is distributed as a `.json` file containing serialized `Solute` objects that define the schema for aggregated property data (see *below*). By default, each instance of `Solution` loads this file as a `maggma JSONStore` and queries data from it using the `Store` interface.

If desired, users can point a `Solution` instance to an alternate database by using the `database` keyword argument at creation. The argument should contain either 1) the path to an alternate `.json` file (as a `str`) or 2) a `maggma.Store` instance. The data in the file or `Store` must match the schema defined by `Solute`, with the field `formula` used as the key field (unique identifier).

```
s1 = Solution(database='/path/to/my_database.json')
```

or

```
from maggma.core import JSONStore

db_store = JSONStore('/path/to/my_database.json', key='formula')
s1 = Solution(database=db_store)
```

The `Solute` class

`pyEQL.Solute` is a `dataclass` that defines a schema for organizing solute property data. You can think of the schema as a structured dictionary: `Solute` defines the naming and organization of the keys. You can create a basic `Solute` from just the solute's formula as follows:

```
>>> from pyEQL.solute import Solute
>>> Solute.from_formula('Ti+2')
Solute(formula='Ti[+2]', charge=2, molecular_weight='47.867 g/mol', elements=['Ti'],␣
↪chemsys='Ti', pmg_ion=Ion: Ti1 +2, formula_html='Ti<sup>+2</sup>', formula_latex='Ti$^
↪{+2}$', formula_hill='Ti', formula_pretty='Ti^+2', oxi_state_guesses=({'Ti': 2.0},), n_
↪atoms=1, n_elements=1, size={'radius_ionic': None, 'radius_hydrated': None, 'radius_vdw
↪': None, 'molar_volume': None}, thermo={'G_hydration': None, 'G_formation': None},␣
↪transport={'diffusion_coefficient': None}, model_parameters={'activity_pitzer': {'Beta0
↪': None, 'Beta1': None, 'Beta2': None, 'Cphi': None, 'Max_C': None}, 'molar_volume_
↪pitzer': {'Beta0': None, 'Beta1': None, 'Beta2': None, 'Cphi': None, 'V_o': None, 'Max_
↪C': None}, 'viscosity_jones_dole': {'B': None}})
```

This method uses `pymatgen` to populate the `Solute` with basic chemical information like molecular weight. You can access top-level keys in the schema via attribute, e.g.

```
>>> s.molecular_weight
'47.867 g/mol'
>>> s.charge
2.0
```

Other properties that are present in the schema, but not set, are `None`. For example, here we have not specified a diffusion coefficient. If we inspect the `transport` attribute, we see

```
>>> s.transport
{'diffusion_coefficient': None}
```

You can convert a `Solute` into a regular dictionary using `Solute.as_dict()`

‘’

s.as_dict() {‘formula’: ‘Ti[+2]’, ‘charge’: 2, ‘molecular_weight’: ‘47.867 g/mol’, ‘elements’: [‘Ti’], ‘chemsys’: ‘Ti’, ‘pmg_ion’: Ion: Ti1 +2, ‘formula_html’: ‘Ti+2’, ‘formula_latex’: ‘Ti^{+2}’, ‘formula_hill’: ‘Ti’, ‘formula_pretty’: ‘Ti^+2’, ‘oxi_state_guesses’: ({‘Ti’: 2.0},), ‘n_atoms’: 1, ‘n_elements’: 1, ‘size’: {‘radius_ionic’: None, ‘radius_hydrated’: None, ‘radius_vdw’: None, ‘molar_volume’: None}, ‘thermo’: {‘G_hydration’: None, ‘G_formation’: None}, ‘transport’: {‘diffusion_coefficient’: None}, ‘model_parameters’: {‘activity_pitzer’: {‘Beta0’: None, ‘Beta1’: None, ‘Beta2’: None, ‘Cphi’: None, ‘Max_C’: None}, ‘molar_volume_pitzer’: {‘Beta0’: None, ‘Beta1’: None, ‘Beta2’: None, ‘Cphi’: None, ‘V_o’: None, ‘Max_C’: None}, ‘viscosity_jones_dole’: {‘B’: None}}} ‘”

Searching the database

Once you have a created a `Solution`, it will automatically search the database for needed parameters whenever it needs to perform a calculation. For example, if you call `get_transport_number`, `pyEQL` will search the property database for diffusion coefficient data to use in the calculation. No user action is needed.

If you want to search the database yourself, or to inspect the values that `pyEQL` uses for a particular parameter, you can do so via the `get_property` method. First, create a `Solution`

```
>>> from pyEQL import Solution
>>> s1 = pyEQL.Solution
```

Next, call `get_property` with a solute name and the name of the property you need. Valid property names are any key in the `Solute` schema. Nested keys can be separated by periods, e.g. “model_parameters.activity_pitzer”:

```
>>> s1.get_property('Mg+2', 'transport.diffusion_coefficient')
<Quantity(0.00705999997, 'centimeter ** 2 * liter * pascal * second / kilogram / meter␣
↪** 2')>
```

If the property exists, it will be returned as a `pint` `Quantity` object, which you can convert to specific units if needed, e.g.

```
>>> s1.get_property('Mg+2', 'transport.diffusion_coefficient').to('m**2/s')
<Quantity(7.05999997e-10, 'meter ** 2 / second')>
```

If the property does not exist in the database, `None` will be returned.

```
>>> s1.get_property('Mg+2', 'transport.randomproperty')
>>>
```

Although the database contains additional context about each and every property value, such as a citation, this information is not currently exposed via the `Solution` interface. Richer methods for exploring and adding to the database may be added in the future.

Species included

The database currently contains one or more physichochemical properties for each of the solutes listed below. More detailed information about which properties are available for which solutes may be added in the future.

- Ac[+3]
- Ag(CN)2[-1]
- AgNO3(aq)
- Ag[+1]
- Ag[+2]
- Ag[+3]
- Al2(SO4)3(aq)
- Al[+3]
- AsO4[-3]
- Au(CN)2[-1]
- Au(CN)4[-1]
- Au[+1]
- Au[+2]
- Au[+3]
- B(H5C6)4[-1]
- B(OH)3(aq)
- B(OH)4[-1]
- BF4[-1]
- BO2[-1]
- Ba(ClO4)2(aq)
- Ba(NO3)2(aq)
- BaBr2(aq)
- BaC4O.3H2O(aq)
- BaCl2(aq)
- BaI2(aq)
- Ba[+2]
- BeSO4(aq)
- Be[+2]
- Bi[+3]
- BrO3[-1]
- Br[-0.33333333]
- Br[-1]
- C2N3[-1]

- CH3COO[-1]
- CNO[-1]
- CN[-1]
- CO3[-2]
- CSN[-1]
- CSeN[-1]
- Ca(ClO4)2(aq)
- Ca(NO3)2(aq)
- CaBr2(aq)
- CaCl2(aq)
- CaI2(aq)
- Ca[+2]
- Cd(ClO4)2(aq)
- Cd(NO2)2(aq)
- Cd(NO3)2(aq)
- CdSO4(aq)
- Cd[+2]
- CeCl3(aq)
- Ce[+3]
- Ce[+4]
- ClO2[-1]
- ClO3[-1]
- ClO4[-1]
- Cl[-1]
- Co(CN)6[-3]
- Co(H3N)6[-3]
- Co(NO3)2(aq)
- CoBr2(aq)
- CoCl2(aq)
- CoI2(aq)
- Co[+2]
- Co[+3]
- Cr(NO3)3(aq)
- CrCl3(aq)
- CrO4[-2]
- Cr[+2]

- Cr[+3]
- Cs2SO4(aq)
- CsBr(aq)
- CsCl(aq)
- CsF(aq)
- CsHC2O.1H2O(aq)
- CsI(aq)
- CsNO2(aq)
- CsNO3(aq)
- CsOH(aq)
- Cs[+1]
- Cu(NO3)2(aq)
- CuCl2(aq)
- CuSO4(aq)
- Cu[+1]
- Cu[+2]
- Cu[+3]
- Dy[+2]
- Dy[+3]
- Er[+2]
- Er[+3]
- Eu(NO3)3(aq)
- EuCl3(aq)
- Eu[+2]
- Eu[+3]
- F[-1]
- Fe(CN)6[-3]
- Fe(CN)6[-4]
- FeCl2(aq)
- FeCl3(aq)
- Fe[+2]
- Fe[+3]
- Ga[+3]
- GdCl3(aq)
- Gd[+3]
- Ge[+2]

- H2CO3(aq)
- H2O(aq)
- H2SNO3[-1]
- H2SO4(aq)
- H3O[+1]
- H4BrN(aq)
- H4IN(aq)
- H4N2O3(aq)
- H4NCl(aq)
- H4NClO4(aq)
- H4N[+1]
- H4SNO4(aq)
- H5C6O7[-3]
- H5N2[+1]
- H8S(NO2)2(aq)
- HBr(aq)
- HCO2[-1]
- HCO3[-1]
- HCl(aq)
- HClO4(aq)
- HF2[-1]
- HI(aq)
- HNO3(aq)
- HO2[-1]
- HOsO5[-1]
- HSO3[-1]
- HSO4[-1]
- HS[-1]
- HSeO3[-1]
- H[+1]
- Hf[+4]
- Hg[+2]
- Ho[+2]
- Ho[+3]
- IO3[-1]
- IO4[-1]

- I[-1]
- In[+1]
- In[+2]
- In[+3]
- IrO4[-1]
- Ir[+3]
- K2CO3(aq)
- K2PHO4(aq)
- K2SO4(aq)
- K3Fe(CN)6(aq)
- K3PO4(aq)
- K4Fe(CN)6(aq)
- KBr(aq)
- KBrO3(aq)
- KCSN(aq)
- KCl(aq)
- KClO3(aq)
- KClO4(aq)
- KCrO4(aq)
- KF(aq)
- KHC2O.1H2O(aq)
- KHCO3(aq)
- KI(aq)
- KNO2(aq)
- KNO3(aq)
- KOH(aq)
- KPO3.1H2O(aq)
- K[+1]
- La(NO3)3(aq)
- LaCl3(aq)
- La[+3]
- Li2SO4(aq)
- LiBr(aq)
- LiCl(aq)
- LiClO4(aq)
- LiHC2O.1H2O(aq)

- LiI(aq)
- LiNO2(aq)
- LiNO3(aq)
- LiOH(aq)
- Li[+1]
- Lu[+3]
- Mg(ClO4)2(aq)
- Mg(NO3)2(aq)
- MgBr2(aq)
- MgC4O.3H2O(aq)
- MgCl2(aq)
- MgI2(aq)
- MgSO4(aq)
- Mg[+2]
- MnCl2(aq)
- MnO4[-1]
- MnSO4(aq)
- Mn[+2]
- Mn[+3]
- MoO4[-2]
- Mo[+3]
- NO2[-1]
- NO3[-1]
- N[-0.33333333]
- Na2CO3(aq)
- Na2PHO4(aq)
- Na2S2O3(aq)
- Na2SO4(aq)
- Na3PO4(aq)
- NaBr(aq)
- NaBrO3(aq)
- NaCSN(aq)
- NaCl(aq)
- NaClO4(aq)
- NaCrO4(aq)
- NaF(aq)

- NaHC2O.1H2O(aq)
- NaHC3.2H2O(aq)
- NaHCO2(aq)
- NaHCO3(aq)
- NaI(aq)
- NaNO2(aq)
- NaNO3(aq)
- NaOH(aq)
- NaPO3.1H2O(aq)
- Na[+1]
- Nb[+3]
- Nd(NO3)3(aq)
- NdCl3(aq)
- Nd[+2]
- Nd[+3]
- Ni(NO3)2(aq)
- NiCl2(aq)
- NiSO4(aq)
- Ni[+2]
- Ni[+3]
- Np[+3]
- Np[+4]
- OH[-1]
- Os[+3]
- P(HO2)2[-1]
- P(OH)2[-1]
- P2O7[-4]
- P3O10[-5]
- PF6[-1]
- PH9(NO2)2(aq)
- PHO4[-2]
- PO3F[-2]
- PO3[-1]
- PO4[-3]
- Pa[+3]
- Pb(ClO4)2(aq)

- Pb(NO3)2(aq)
- Pb[+2]
- Pd[+2]
- Pm[+2]
- Pm[+3]
- Po[+2]
- PrCl3(aq)
- Pr[+2]
- Pr[+3]
- Pt[+2]
- Pu[+2]
- Pu[+4]
- Ra[+2]
- Rb2SO4(aq)
- RbBr(aq)
- RbCl(aq)
- RbF(aq)
- RbHC2O.1H2O(aq)
- RbI(aq)
- RbNO2(aq)
- RbNO3(aq)
- RbOH(aq)
- Rb[+1]
- ReO4[-1]
- Re[+1]
- Re[+3]
- Re[-1]
- Rh[+3]
- Ru[+2]
- Ru[+3]
- S2O3[-2]
- SO2[-1]
- SO3[-1]
- SO3[-2]
- SO4[-1]
- SO4[-2]

- S[-2]
- Sb(HO2)2[-1]
- Sb(OH)6[-1]
- ScCl3(aq)
- Sc[+2]
- Sc[+3]
- SeO3[-1]
- SeO4[-1]
- SeO4[-2]
- SiF6[-2]
- SmCl3(aq)
- Sm[+2]
- Sm[+3]
- Sn[+2]
- Sn[+4]
- Sr(ClO4)2(aq)
- Sr(NO3)2(aq)
- SrBr2(aq)
- SrCl2(aq)
- SrI2(aq)
- Sr[+2]
- Ta[+3]
- Tb[+3]
- TcO4[-1]
- Tc[+2]
- Tc[+3]
- Th(NO3)4(aq)
- Th[+4]
- Ti[+2]
- Ti[+3]
- Tl(ClO4)3(aq)
- Tl(NO2)3(aq)
- Tl(NO3)3(aq)
- TlH(C3O)2.4H2O(aq)
- Tl[+1]
- Tl[+3]

- Tm[+2]
- Tm[+3]
- U(ClO)2(aq)
- U(ClO5)2(aq)
- U(NO4)2(aq)
- UO2[+1]
- UO2[+2]
- USO6(aq)
- U[+3]
- U[+4]
- VO2[+1]
- V[+2]
- V[+3]
- WO4[-1]
- WO4[-2]
- W[+3]
- YCl3(aq)
- YNO3(aq)
- Y[+3]
- Yb[+2]
- Yb[+3]
- Zn(ClO4)2(aq)
- Zn(NO3)2(aq)
- ZnBr2(aq)
- ZnCl2(aq)
- ZnI2(aq)
- ZnSO4(aq)
- Zn[+2]
- Zr[+4]

1.1.6 Contributing to pyEQL

Reporting Issues

You can help the project simply by using pyEQL and comparing the output to experimental data and/or other models and tools. If you encounter any bugs, packaging issues, feature requests, comments, or questions, please report them using the issue tracker on github.

Contributing Code

To contribute bug fixes, documentation enhancements, or new code, please fork pyEQL and send us a pull request. It's not as hard as it sounds! Beginning with version 0.6.0, we follow the GitHub flow workflow model.

Hacking pyEQL, step by step

1. Fork the pyEQL repository on Github
2. Clone your repository to a directory of your choice:

   ```
   git clone https://github.com/<username>/pyEQL
   ```

3. Install the package and the test dependencies by running the following command from the repository directory:

   ```
   pip install -e '.[testing]``
   ```

4. Create a branch for your work. Preferably, start your branch name with "feature-", "fix-", or "doc-" depending on whether you are contributing **bug fixes**, **documentation** or a **new feature**, e.g. prefix your branch with "fix-" or "doc-" as appropriate:

   ```
   git checkout -b mybranch
   ```

   or

   ```
   git checkout -b doc-mydoc
   ```

   or

   ```
   git checkout -b feature-myfeature
   ```

5. Make changes to the code until you're satisfied.
6. Push your work back to Github:

   ```
   git push origin feature-myfeature
   ```

7. Create a pull request with your changes. See this tutorial for instructions.

Guidelines

Please abide by the following guidelines when contributing code to `pyEQL`:

- All changes you make to quacc should be accompanied by unit tests and should not break existing tests. To run the full test suite, run `pytest tests/` from the repository directory.
- Code coverage should be maintained or increase. Each PR will report code coverage after the tests pass, but you can check locally using pytest-cov, by running `pytest --cov tests/`
- All code should include type hints and have internally consistent documentation for the inputs and outputs.
- Use Google style docstrings
- Lint your code with `ruff` by running `ruff check --fix src/` from the repo directory. Alternatively, you can install the `pre-commit` hooks by running `pre-commit install` from the repository directory. This will prevent committing new changes until all linting errors are fixed.
- Update the `CHANGELOG.md` file.
- Ask questions and be open to feedback!

Documentation

Improvements to the documentation are most welcome! Our documentation system uses `sphinx` with the Materials for Sphinx theme. To edit the documentation locally, run `tox -e autodocs` from the repository root directory. This will serve the documents to http://localhost:8000/ so you can view them in your web browser. When you make changes to the files in the `docs/` directory, the documentation will automatically rebuild and update in your browser (you might have to refresh the page to see changes).

Changelog

We keep a `CHANGELOG.md` file in the base directory of the repository. Before submitting your PR, be sure to update the `CHANGELOG.md` file under the "Unreleased" section with a brief description of your changes. Our `CHANGELOG.md` file lossely follows the Keep a Changelog format, beginning with `v0.6.0`.

1.1.7 Functions Module

pyEQL functions that take Solution objects as inputs or return Solution objects.

copyright
: 2013-2023 by Ryan S. Kingsbury

license
: LGPL, see LICENSE for more details.

`pyEQL.functions.`**`autogenerate`**`(`*solution=''*`)`

This method provides a quick way to create Solution objects representing commonly-encountered solutions, such as seawater, rainwater, and wastewater.

Parameters

`solution` (*`str`*) – String representing the desired solution Valid entries are 'seawater', 'rainwater', 'wastewater',and 'urine'

Returns

A pyEQL Solution object.

Return type
Solution

Notes

The following sections explain the different solution options:

- ‘’ - empty solution, equivalent to pyEQL.Solution()
- ‘rainwater’ - pure water in equilibrium with atmospheric CO2 at pH 6
- ‘seawater’ or ‘SW’- Standard Seawater. See Table 4 of the Reference for Composition [1]_
- ‘wastewater’ or ‘WW’ - medium strength domestic wastewater. See Table 3-18 of [2]_
- ‘urine’ - typical human urine. See Table 3-15 of [2]_
- ‘normal saline’ or ‘NS’ - normal saline solution used in medicine [3]_
- ‘Ringers lacatate’ or ‘RL’ - Ringer’s lactate solution used in medicine [4]_

References:

`pyEQL.functions.donnan_eql(`*solution*, *fixed_charge*`)`

Return a solution object in equilibrium with fixed_charge.

Parameters

- **solution** (*Solution object*) – The external solution to be brought into equilibrium with the fixed charges
- **fixed_charge** (*str quantity*) – String representing the concentration of fixed charges, including sign. May be specified in mol/L or mol/kg units. e.g. ‘1 mol/kg’

Returns
A solution that has established Donnan equilibrium with the external (input) Solution

Return type
Solution

Notes

The general equation representing the equilibrium between an external electrolyte solution and an ion-exchange medium containing fixed charges is

In addition, electroneutrality must prevail within the membrane phase:

$$\bar{C}_+ z_+ + \bar{X} + \bar{C}_- z_- = 0$$

Where C represents concentration and X is the fixed charge concentration in the membrane or ion exchange phase.

This function solves these two equations simultaneously to arrive at the concentrations of the cation and anion in the membrane phase. It returns a solution equal to the input solution except that the concentrations of the predominant cation and anion have been adjusted according to this equilibrium.

NOTE that this treatment is only capable of equilibrating a single salt. This salt is identified by the get_salt() method.

References

Strathmann, Heiner, ed. *Membrane Science and Technology* vol. 9, 2004. Chapter 2, p. 51.
http://dx.doi.org/10.1016/S0927-5193(04)80033-0

See Also:

get_salt()

`pyEQL.functions.`**`entropy_mix`**(*Solution1*, *Solution2*)

Return the ideal mixing entropy associated with mixing two solutions.

Parameters

- **`Solution1`** (*`Solution objects`*) – The two solutions to be mixed.
- **`Solution2`** (*`Solution objects`*) – The two solutions to be mixed.

Returns
The ideal mixing entropy associated with complete mixing of the Solutions, in Joules.

Return type
Quantity

Notes

The ideal entropy of mixing is calculated as follows

$$Delta_{mix}S =$$
$$sum_i(n_c + n_d)RT$$
$$lnx_b -$$
$$sum_i n_c RT$$
$$lnx_c -$$
$$sum_i n_d RT$$
$$lnx_d$$

Where n is the number of moles of substance, T is the temperature in kelvin, and subscripts b, c, and d refer to the concentrated, dilute, and blended Solutions, respectively.

Note that dissociated ions must be counted as separate components, so a simple salt dissolved in water is a three component solution (cation, anion, and water).

References

Koga, Yoshikata, 2007. *Solution Thermodynamics and its Application to Aqueous Solutions:
A differential approach.* Elsevier, 2007, pp. 23-37.

`pyEQL.functions.`**`gibbs_mix`**(*Solution1*, *Solution2*)

Return the Gibbs energy change associated with mixing two solutions.

Parameters

- **`Solution1`** (*`Solution objects`*) – The two solutions to be mixed.

- **Solution2** (*Solution objects*) – The two solutions to be mixed.

Returns
The change in Gibbs energy associated with complete mixing of the Solutions, in Joules.

Return type
Quantity

Notes

The Gibbs energy of mixing is calculated as follows

$$Delta_{mix}G =$$
$$sum_i(n_c + n_d)RT$$
$$lna_b -$$
$$sum_i n_c RT$$
$$lna_c -$$
$$sum_i n_d RT$$
$$lna_d$$

Where n is the number of moles of substance, T is the temperature in kelvin, and subscripts b, c, and d refer to the concentrated, dilute, and blended Solutions, respectively.

Note that dissociated ions must be counted as separate components, so a simple salt dissolved in water is a three component solution (cation, anion, and water).

References

Koga, Yoshikata, 2007. *Solution Thermodynamics and its Application to Aqueous Solutions:
A differential approach.* Elsevier, 2007, pp. 23-37.

`pyEQL.functions.mix(s1, s2)`
Mix two solutions together.

Parameters

- **s1** – The two solutions to be mixed.
- **s2** – The two solutions to be mixed.

Returns
A Solution object that represents the result of mixing s1 and s2.

Notes

The initial volume of the mixed solution is set as the sum of the volumes of s1 and s2. The pressure and temperature are volume-weighted averages. The pH and pE values are currently APPROXIMATE because they are calculated assuming H+ and e- mix conservatively (i.e., the mixing process does not incorporate any equilibration reactions or buffering). Such support is planned in a future release.

1.1.8 Internal module reference

These internal modules are used by `Solution` but typically are not directly accessed by the user.

Salt analysis module

Activity Correction module

Speciation Engines module

PYTHON MODULE INDEX

p

pyEQL.functions, 24

INDEX

A

`autogenerate()` (*in module pyEQL.functions*), 24

D

`donnan_eql()` (*in module pyEQL.functions*), 25

E

`entropy_mix()` (*in module pyEQL.functions*), 26

G

`gibbs_mix()` (*in module pyEQL.functions*), 26

M

`mix()` (*in module pyEQL.functions*), 27
`module`
    `pyEQL.functions`, 24

P

`pyEQL.functions`
    `module`, 24