pyEQL Documentation

Release v0.15.1

Ryan Kingsbury

Mar 14, 2024

CONTENTS


pyEQL

CHAPTER

ONE

DESCRIPTION

The goal of pyEQL is to provide a stable, intuitive, easy to learn python interface for water chemistry that can be connected to a variety of different modeling engines

Specifically, pyEQL defines a Solution class to represent an aqueous electrolyte solution. Virtually all of the user-facing functions in pyEQL are accessed through the Solution class. It also includes a number of other utilities to support water chemistry analysis including a database of diffusion coefficients, activity correction parameters, and other data on a variety of common electrolytes.

CHAPTER

TWO

1-MINUTE TUTORIAL

2.1 Install

```
pip install pyEQL
```

2.2 Create a Solution

```
>>> from pyEQL import Solution
>>> s1 = Solution({'Na+':'0.5 mol/kg', 'Cl-': '0.5 mol/kg'},
                          pH=8,
                          temperature = '20 degC',
                          volume='8 L')
```

2.3 Get properties

```
>>> s1.density
<Quantity(1.03710384, 'kilogram / liter')>
>>> s1.conductivity
<Quantity(8.09523295, 'siemens / meter')>
>>> s1.osmotic_pressure.to('atm')
<Quantity(46.7798197, 'standard_atmosphere')>
>>> s1.get_amount('Na+', 'ug/L')
<Quantity(22989769.3, 'microgram / liter')>
```

CHAPTER

THREE

KEY FEATURES

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.

- 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.
- Built in database containing hundreds of model parameters and physicochemical properties for different ions.
- Customizable *modeling engine system* that allows the `Solution` API to work with multiple electrolyte models.
- *Units-aware calculations* (by means of the pint library)

CHAPTER

FOUR

CONTENTS:

4.1 Quickstart

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.

4.1.1 Creating a Solution Object

Create a Solution object by invoking the Solution class:

```
>>> from pyEQL import Solution
>>> s1 = Solution()
>>> s1
<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 = Solution({'Na+':'0.5 mol/kg', 'Cl-': '0.5 mol/kg'}, pH=8, temperature = '20 degC
↪', volume='8 L')
```

See *Creating a Solution* for more details.

4.1.2 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
```

See *Getting Concentrations* for more details.

4.1.3 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
>>> test_qty.magnitude
1.0
```

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).

See the *Converting Units* for more details.

4.2 pyEQL Overview

`pyEQL` is an open-source `python` library for solution chemistry calculations and ion properties developed by the Kingsbury Lab at Princeton University.

Documentation | How to Install | GitHub

4.2.1 Installation

Uncomment and run the code cell below, if you do not already have pyEQL

[2]:
```
# pip install pyEQL
```

4.2.2 Main feature: The Solution class

[1]:
```
from pyEQL import Solution
```

[2]:
```
s1 = Solution({"Mg+2": "0.2 mol/L", "Cl-1": "0.4 mol/L"}, temperature='20 degC')
```

```
WARNING 2023-11-07 11:18:03,638 solution.py _get_property 2084 Partial molar volume for␣
↪species H[+1] not corrected for temperature
WARNING 2023-11-07 11:18:03,648 solution.py _get_property 2084 Partial molar volume for␣
↪species OH[-1] not corrected for temperature
WARNING 2023-11-07 11:18:03,922 solution.py _get_property 2084 Partial molar volume for␣
↪species Mg[+2] not corrected for temperature
WARNING 2023-11-07 11:18:03,951 activity_correction.py _debye_parameter_volume 231 Debye-
↪Huckel limiting slope for volume is approximate when T is not equal to 25 degC
```

Bulk Properties

[3]:
```
s1.density
```

[3]: $1.0138757383570756 \frac{\mathrm{kg}}{\mathrm{l}}$

[4]:
```
s1.conductivity
```

```
WARNING 2023-11-07 11:18:04,061 engines.py get_activity_coefficient 314 Ionic strength␣
↪too high to estimate activity for species H[+1]. Specify parameters for Pitzer model.␣
↪Returning unit activity coefficient
WARNING 2023-11-07 11:18:04,079 engines.py get_activity_coefficient 314 Ionic strength␣
↪too high to estimate activity for species OH[-1]. Specify parameters for Pitzer model.␣
↪Returning unit activity coefficient
```

[4]: $3.299995263108893 \frac{\mathrm{S}}{\mathrm{m}}$

[5]:
```
s1.volume
```

[5]: 1 l

[6]:
```
s1.pressure
```

[6]: 1 atm

[7]:
```
s1.temperature
```

[7]: 293.15 K

[8]:
```
s1.osmotic_pressure
```

[8]: 1286752.4185332006 Pa

Composition

[9]:
```
s1.components
```

[9]:
```
{'H2O(aq)': 55.221652761186476, 'Cl[-1]': 0.4, 'Mg[+2]': 0.2, 'H[+1]': 1e-07, 'OH[-1]':␣
↪1e-07}
```

[10]:
```
s1.solvent
```

[10]:
```
'H2O(aq)'
```

[11]:
```
s1.cations
```

[11]:
```
{'Mg[+2]': 0.2, 'H[+1]': 1e-07}
```

[12]:
```
s1.anions
```

[12]:
```
{'Cl[-1]': 0.4, 'OH[-1]': 1e-07}
```

[13]:
```
s1.neutrals
```

[13]:
```
{'H2O(aq)': 55.221652761186476}
```

Species Concentrations

[14]:
```
s1.get_amount('Mg+2', 'M')
```

[14]: $0.20000000000000007\ \mathrm{M}$

[15]:
```
s1.get_amount('Cl-', '%')
```

[15]: 1.3987118404647676

[16]:
```
s1.get_amount('Mg+2', 'eq/L')
```

[16]: $0.4\ \frac{\mathrm{mol}}{\mathrm{l}}$

[17]:
```
s1.get_amount('Mg+2', 'ug/kg')
```

[17]: $4886244.60412788\ \frac{\mathrm{g}}{\mathrm{kg}}$

Transport

[18]:
```
s1.get_transport_number('Na+')
```

[18]: 0.0

[19]:
```
s1.get_transport_number('Mg+2')
```

[19]: 0.40998795156327783

[20]:
```
s1.get_transport_number('Cl-')
```

[20]: 0.5900109897851136

Speciation

[21]:
```
s1.equilibrate()
```

[22]:
```
s1.components
```

[22]:
```
{'H2O(aq)': 55.238455538403954, 'Cl[-1]': 0.38323989957700755, 'Mg[+2]': 0.
↪18323990520853942, 'MgCl[+1]': 0.016760093825573405, 'H[+1]': 1.2403230417432767e-07,
↪'OH[-1]': 9.844314720800745e-08, 'HCl(aq)': 5.431780489909693e-09, 'MgOH[+1]': 7.
↪932661202804441e-15, 'O2(aq)': 3.1477649388058775e-26, 'HClO(aq)': 8.450946375546259e-
↪29, 'ClO[-1]': 3.367672345557701e-29, 'H2(aq)': 5.442728186726209e-35, 'ClO2[-1]': 0.0,
↪ 'ClO3[-1]': 0.0, 'ClO4[-1]': 0.0, 'HClO2(aq)': 0.0}
```

Saving `Solution` to a file

[23]:
```
from monty.serialization import dumpfn
dumpfn(s1, 'test_solution.json')
```

[24]:
```
s1.as_dict()
```

[24]:
```
{'@module': 'pyEQL.solution',
 '@class': 'Solution',
 '@version': '0.9.0.post1.dev3+g22e5c4a',
 'solutes': {'H2O(aq)': '55.238455538403954 mol',
  'Cl[-1]': '0.38323989957700755 mol',
  'Mg[+2]': '0.18323990520853942 mol',
  'MgCl[+1]': '0.016760093825573405 mol',
  'H[+1]': '1.2403230417432767e-07 mol',
  'OH[-1]': '9.844314720800745e-08 mol',
  'HCl(aq)': '5.431780489909693e-09 mol',
  'MgOH[+1]': '7.932661202804441e-15 mol',
  'O2(aq)': '3.1477649388058775e-26 mol',
  'HClO(aq)': '8.450946375546259e-29 mol',
  'ClO[-1]': '3.367672345557701e-29 mol',
  'H2(aq)': '5.442728186726209e-35 mol',
  'ClO2[-1]': '0.0 mol',
  'ClO3[-1]': '0.0 mol',
  'ClO4[-1]': '0.0 mol',
  'HClO2(aq)': '0.0 mol'},
 'volume': '1 l',
 'temperature': '293.15 K',
 'pressure': '1 atm',
 'pH': 6.90646518824501,
 'pE': 8.5,
 'balance_charge': None,
 'solvent': 'H2O(aq)',
 'engine': 'native',
 'database': {'@module': 'maggma.stores.mongolike',
  '@class': 'JSONStore',
  '@version': '0.57.4',
  'paths': ['/home/ryan/mambaforge/envs/pbx/code/pyEQL/src/pyEQL/database/pyeql_db.json
↪'],
```

(continues on next page)

(continued from previous page)

```
  'read_only': True,
  'serialization_option': None,
  'serialization_default': None,
  'key': 'formula'}}
```

4.2.3 Units-Aware Calculations

[25]:
```
s1.volume.to('mL')
```

[25]: 1000.0000000000001 ml

[26]:
```
s1.volume.to('gal')
```

[26]: 0.26417205235814856 gal

[27]:
```
s1.osmotic_pressure.to('bar').magnitude
```

```
WARNING 2023-11-07 11:18:04,616 engines.py get_osmotic_coefficient 462 Cannot calculate␣
↪osmotic coefficient because Pitzer parameters for salt HClO3 are not specified.␣
↪Returning unit osmotic coefficient
WARNING 2023-11-07 11:18:04,621 engines.py get_osmotic_coefficient 462 Cannot calculate␣
↪osmotic coefficient because Pitzer parameters for salt HClO2 are not specified.␣
↪Returning unit osmotic coefficient
```

[27]:
```
12.630074635540739
```

4.2.4 Contribution Opportunities

Benchmarking - Compiling additional validation data for activity, conductivity, etc. - Quantifying error associated with different models - Refactoring unit tests suite to separate benchmarking

Documentation - Writing tutorials - Writing expanded docs - Cleaning up / updating docstrings

New Features - Better viscosity model - Expanded unit testing (increase test coverage to 90%) - Additional properties - Additional mixing rules / models for mixed electrolytes

Database - Expand database doverage to include additional species - More viscosity coefficients - Add 'sho' parameter - More diffusion coefficients

Software Engineering - Additional refactoring (e.g., `mypy` linting for robustness) - Bugfixes

4.3 Tutorials

Each tutorial below is presented in a Jupyter notebook. You can view the executed notebooks here in the documentation, view the raw notebooks on GitHub, or interactively run them in your web browser using Binder by clicking the respective links below.

4.3.1 Functionality Overview

View Notebook on GitHub | Try Interactive Notebook on Binder

4.3.2 Calculating Osmotic Pressure

View Notebook on GitHub | Try Interactive Notebook on Binder

pyEQL Tutorial: Calculating Osmotic Pressure

pyEQL is an open-source python library for solution chemistry calculations and ion properties developed by the Kingsbury Lab at Princeton University.

Documentation | How to Install | GitHub

Installation

Uncomment and run the code cell below, if you do not already have pyEQL

[1]:
```
# pip install pyEQL
```

First, create a Solution

pyEQL's built-in property database contains Pitzer model parameters for many simple (binary) electrolytes. If such parameters are available, pyEQL will use them by default.

[2]:
```
from pyEQL import Solution
# 2 mol/L NaCl
s1 = Solution({"Na+": "2 mol/L", "Cl-": "2 mol/L"})
```

Get the osmotic pressure

Note that the osmotic pressure (and most Solution properties) are returned as pint Quantity objects (see Converting Units).

[3]:
```
s1.osmotic_pressure
```

[3]:
```
10224795.514383134 Pa
```

If you want the osmotic pressure in different units, or you only want the magnitude, use to() and magnitude, respectively

[4]:
```
s1.osmotic_pressure.to('bar')
```

[4]:
```
102.24795514383135 bar
```

[5]:
```
s1.osmotic_pressure.to('bar').magnitude
```

[5]:
```
102.24795514383135
```

Use a `for` loop for multiple calculations

You can rapidly get estimates for multiple concentrations (or temperatures, or solutes) by using a `for` loop. Notice how in the example below, we use f-strings to insert the desired concentration (from the `for` loop) into the argument passed to `Solution` and to print the results.

[6]:
```
for conc in [0.1, 0.5, 1, 2, 4]:
    s1 = Solution({"Na+": f"{conc} mol/L", "Cl-": f"{conc} mol/L"})
    print(f"At C={conc:.1f} M, the osmotic pressure is {s1.osmotic_pressure.to('bar'):.
↪2f}.")
```

```
At C=0.1 M, the osmotic pressure is 4.63 bar.
At C=0.5 M, the osmotic pressure is 23.07 bar.
At C=1.0 M, the osmotic pressure is 47.40 bar.
At C=2.0 M, the osmotic pressure is 102.25 bar.
At C=4.0 M, the osmotic pressure is 246.95 bar.
```

Compare different modeling engines

`pyEQL` contains several different modeling engines that can calculate activity coefficients or osmotic pressures. At present, there are three options:

1. The `native` or built-in engine, which includes an implementation of the Piter model (Default).
2. the `phreeqc` engine, which utilizes the USGS PHREEQC model with the `phreeqc.dat` database.
3. An ideal solution model (`ideal`) which does not account for solution non-ideality.

You select a modeling engine using the `engine` keyword argument when you create a `Solution`. Let's compare the preditions from the three models.

[7]:
```
s_ideal = Solution({"Na+": "2 mol/L", "Cl-": "2 mol/L"}, engine='ideal')
s_phreeqc = Solution({"Na+": "2 mol/L", "Cl-": "2 mol/L"}, engine='phreeqc')
s_native = Solution({"Na+": "2 mol/L", "Cl-": "2 mol/L"}, engine='native')
```

[8]:
```
s_ideal.osmotic_pressure.to('bar')
```

```
WARNING 2023-11-10 11:48:57,162 solution.py get_water_activity 1934 Pitzer parameters␣
↪not found. Water activity set equal to mole fraction
```

[8]:
```
95.73878424096024 bar
```

[9]:
```
s_phreeqc.osmotic_pressure.to('bar')
```

[9]:
```
95.73878424096024 bar
```

[10]:
```
s_native.osmotic_pressure.to('bar')
```

[10]:
```
102.24795514383135 bar
```

Plot the comparison vs. experiment

We can make a plot showing how the 3 models compare by combining the two previous steps (using a `for` loop plus changing the `engine` keyword argument. Note that this example makes use of `matplotlib` for plotting.

[11]:
```
# create empty lists to hold the results
pi_ideal = []
pi_phreeqc = []
pi_native = []

concentrations = [0.1, 0.2, 0.3, 0.4, 0.5, 1, 1.4, 2, 2.5, 3, 3.5, 4]

for conc in concentrations:
    s_ideal = Solution({"Na+": f"{conc} mol/kg", "Cl-": f"{conc} mol/kg"}, engine='ideal
↪')
    s_phreeqc = Solution({"Na+": f"{conc} mol/kg", "Cl-": f"{conc} mol/kg"}, engine=
↪'phreeqc')
    s_native = Solution({"Na+": f"{conc} mol/kg", "Cl-": f"{conc} mol/kg"}, engine=
↪'native')

    # store the osmotic pressures in the respective lists
    # note that we have to just store the .magnitude because matplotlib can't plot␣
↪Quantity
    pi_ideal.append(s_ideal.osmotic_pressure.to('bar').magnitude)
    pi_phreeqc.append(s_phreeqc.osmotic_pressure.to('bar').magnitude)
    pi_native.append(s_native.osmotic_pressure.to('bar').magnitude)
```

We will include experimental data from the IDST as a benchmark. The IDST gives us water activity, which we convert into osmotic pressure according to

$$\Pi = -\frac{RT}{V_w} \ln a_w$$

Where Π is the osmotic pressure, V_w is the partial molar volume of water (18.2 cm**3/mol), and a_w is the water activity.

[12]:
```
import math
# water activity at [0.1, 0.2, 0.3, 0.4, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4] mol/kg
water_activity_idst = [0.99664, 0.993353, 0.99008, 0.986804, 0.98352, 0.966828, 0.953166,
↪ 0.93191, 0.913072, 0.89347, 0.872859, 0.85133]

# calculate osmotic pressure as -RT/Vw ln(a_w). Factor 10 converts to bar.
pi_idst = [-8.314*298.15/18.2 * math.log(a) * 10 for a in water_activity_idst]
```

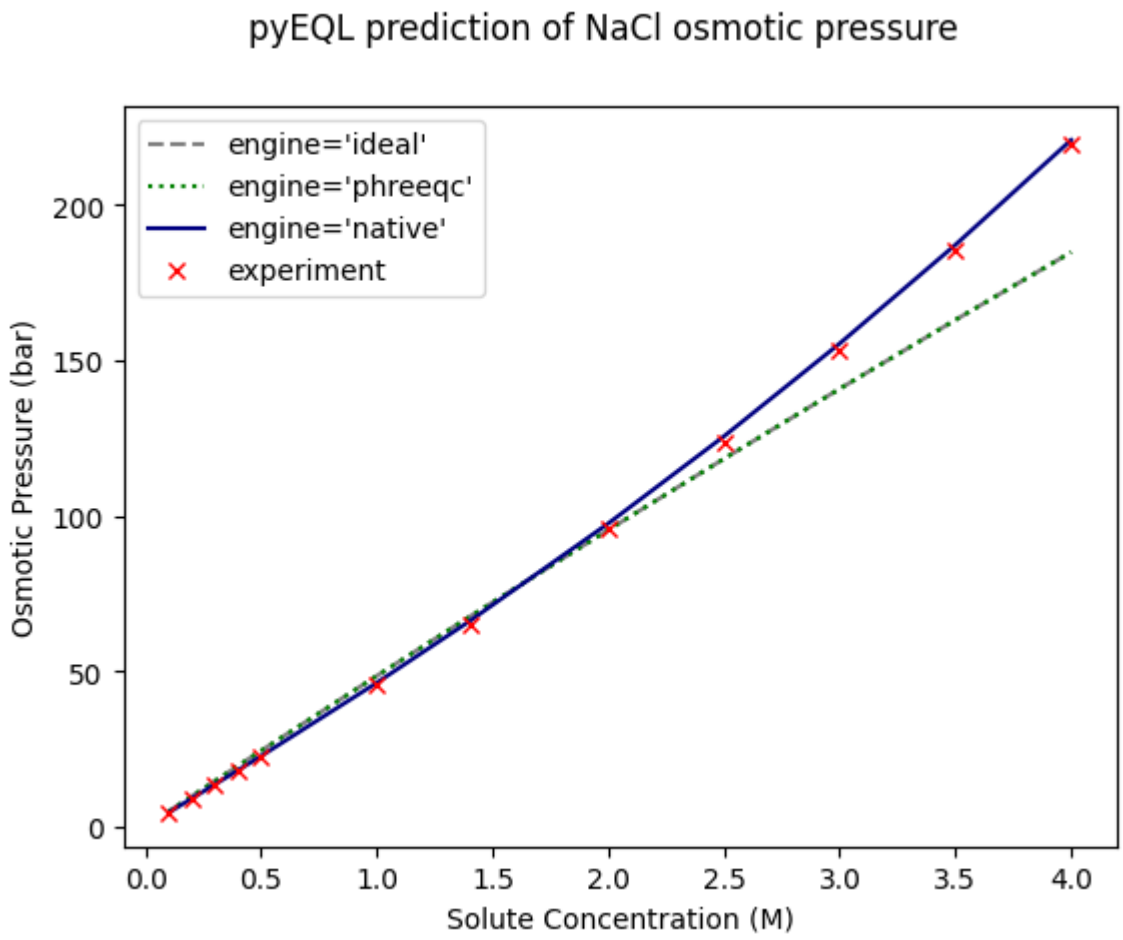
[13]:
```
# plot the results!
from matplotlib import pyplot as plt

fig, ax = plt.subplots()
ax.plot(concentrations, pi_ideal, label="engine='ideal'", ls='--', color='gray')
ax.plot(concentrations, pi_phreeqc, label="engine='phreeqc'", ls=':', color='green')
```

(continues on next page)

(continued from previous page)

```
ax.plot(concentrations, pi_native, label="engine='native'", ls='-', color='navy')
ax.plot(concentrations, pi_idst, label="experiment", ls='', color='red', marker="x")
ax.legend()
ax.set_xlabel('Solute Concentration (M)')
ax.set_ylabel('Osmotic Pressure (bar)')
fig.suptitle('pyEQL prediction of NaCl osmotic pressure')
```

[13]:
```
Text(0.5, 0.98, 'pyEQL prediction of NaCl osmotic pressure')
```



[ ]:

4.3.3 Accessing the Property Database

View Notebook on GitHub | Try Interactive Notebook on Binder

pyEQL Tutorial: Searching the Property Database

pyEQL is an open-source python library for solution chemistry calculations and ion properties developed by the Kingsbury Lab at Princeton University.

Documentation | How to Install | GitHub

Installation

Uncomment and run the code cell below, if you do not already have pyEQL

[1]:
```
# pip install pyEQL
```

First, import the property database

pyEQL's built-in property database contains physichochemical, transport, and model parameters for hundreds of solutes. This information is used behind the scenes when you interact with a Solution object, but it can also be accessed directly.

[2]:
```
from pyEQL import IonDB
```

How to Search the Database

Query an example document

You can think of the database like list of dict that contain structure data. More specifically, the database is a list of `Solute objects <https://pyeql.readthedocs.io/en/latest/database.html#the-solute-class>`__ that have been serialized to dictionaries. We refer to each of these dict as **"documents"** (consistent with MongoDB terminology) or "records"

To see what one document looks like, use query_one(), which retrieves a single record from the database. The record is a dict.

[3]:
```
IonDB.query_one()
```

[3]:
```
{'_id': ObjectId('654e5f131ed012c187817e6a'),
 'formula': 'Ac[+3]',
 'charge': 3,
 'molecular_weight': '227.0 g/mol',
 'elements': ['Ac'],
```

(continues on next page)

(continued from previous page)


```
'chemsys': 'Ac',
'pmg_ion': {'Ac': 1,
 'charge': 3,
 '@module': 'pymatgen.core.ion',
 '@class': 'Ion',
 '@version': None},
'formula_html': 'Ac<sup>+3</sup>',
'formula_latex': 'Ac$^{+3}$',
'formula_hill': 'Ac',
'formula_pretty': 'Ac^+3',
'oxi_state_guesses': {'Ac': 3},
'n_atoms': 1,
'n_elements': 1,
'size': {'radius_ionic': {'value': '1.26 Å',
  'reference': 'pymatgen',
  'data_type': 'experimental'},
 'radius_hydrated': None,
 'radius_vdw': {'value': '2.47 ',
  'reference': 'pymatgen',
  'data_type': 'experimental'},
 'molar_volume': None,
 'radius_ionic_marcus': {'value': '1.18 ± 0.02 ',
  'reference': 'Marcus2015',
  'data_type': 'experimental'}},
'thermo': {'G_hydration': {'value': '-3086.0 ± 10 kJ/mol',
  'reference': '10.1021/acs.jpca.9b05140',
  'data_type': 'experimental'},
 '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}}}
```

Query a specific document

The IonDB is a `maggma.Store <https://materialsproject.github.io/maggma/getting_started/stores/>`__ that can be queried using a MongoDB-like syntax. The basic syntax is

```
IonDB.query_one({field: value})
```

where `field` is a top-level key in the `Solute` `dict`, such as `formula`, `charge`, or `elements`. See this page and the maggma documentation (link WIP) for more detailed examples.

[4]:
```
# a document with the formula "Na[+1]"
IonDB.query_one({"formula":'Na[+1]'})
```

[4]:
```
{'_id': ObjectId('654e5f131ed012c187817f46'),
 'formula': 'Na[+1]',
 'charge': 1,
 'molecular_weight': '22.98976928 g/mol',
 'elements': ['Na'],
 'chemsys': 'Na',
 'pmg_ion': {'Na': 1,
  'charge': 1,
  '@module': 'pymatgen.core.ion',
  '@class': 'Ion',
  '@version': None},
 'formula_html': 'Na<sup>+1</sup>',
 'formula_latex': 'Na$^{+1}$',
 'formula_hill': 'Na',
 'formula_pretty': 'Na^+1',
 'oxi_state_guesses': {'Na': 1},
 'n_atoms': 1,
 'n_elements': 1,
 'size': {'radius_ionic': {'value': '1.16 Å',
   'reference': 'pymatgen',
   'data_type': 'experimental'},
  'radius_hydrated': {'value': '3.58 ',
   'reference': 'Nightingale1959',
   'data_type': 'experimental'},
  'radius_vdw': {'value': '2.27 ',
   'reference': 'pymatgen',
   'data_type': 'experimental'},
  'molar_volume': {'value': '-5.0 cm**3/mol',
   'reference': 'Calculation of the Partial Molal Volume of Organic Compounds and␣
↪Polymers. Progress in Colloid & Polymer Science (94), 20-39.',
   'data_type': 'experimental'},
  'radius_ionic_marcus': {'value': '1.02 ± 0.02 ',
   'reference': 'Marcus2015',
   'data_type': 'experimental'}},
 'thermo': {'G_hydration': {'value': '-427.0 ± 6 kJ/mol',
   'reference': 'Marcus2015',
   'data_type': 'experimental'},
  'G_formation': None},
 'transport': {'diffusion_coefficient': {'value': '1.334e-05 cm**2/s',
   'reference': 'CRC',
   'data_type': 'experimental'}},
```

(continues on next page)

(continued from previous page)

```
 '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': {'value': '0.085 dm**3/mol',
    'reference': 'https://doi.org/10.1021/cr00040a004',
    'data_type': 'fitted'}},
  'dielectric_zuber': {'value': '3.62 dimensionless',
   'reference': 'https://doi.org/10.1016/j.fluid.2014.05.037',
   'data_type': 'fitted'}}}
```

Only return a subset of the document

If you don't need to see the entire document, you can restrict the data returned by the query (in MongoDB, this is called "projection"). To use this feature, pass a second argument that is a `list` containing *only the fields that you want returned*. Note that there is a unique identified (field name `_id`) that is always returned.

[5]:
```
# a document with the formula "Na[+1]", where we only want the formula, charge, and␣
↪molecular_weight
IonDB.query_one({"formula":'Na[+1]'}, ["formula","charge","molecular_weight"])
```

[5]:
```
{'formula': 'Na[+1]',
 'charge': 1,
 'molecular_weight': '22.98976928 g/mol',
 '_id': ObjectId('654e5f131ed012c187817f46')}
```

[6]:
```
# a document with the charge -1, where we only want the formula, charge, and molecular_
↪weight
IonDB.query_one({"charge":-1}, ["formula","charge","molecular_weight"])
```

[6]:
```
{'formula': 'Ag(CN)2[-1]',
 'charge': -1,
 'molecular_weight': '159.903 g/mol',
 '_id': ObjectId('654e5f131ed012c187817e6b')}
```

NOTE: Be mindful of data types when querying. `charge` is an `int`. If we tried to query `charge` as if it were a `str`, we would get no results:

[7]:
```
# a document with the charge -1, where we only want the formula, charge, and molecular_
↪weight
IonDB.query_one({"charge":"-1"}, ["formula","charge","molecular_weight"])
```

Query nested fields

If you want to query a field that is not a top-level key (such as transport / diffusion_coefficient), you can place a `.` between the field names at each level, e.g.

[8]:
```
IonDB.query_one({"size.radius_vdw.value": "2.27 "}, ["formula", "size.radius_vdw.value"])
```

[8]:
```
{'formula': 'Na2CO3(aq)',
 'size': {'radius_vdw': {'value': '2.27 '}},
 '_id': ObjectId('654e5f131ed012c187817f31')}
```

Note that in the `Solute` documents, **most quantitative data are stored as ``str`` so that there is no ambiguity about their units**. In the example above, the value of the van der Waals radius is `"2.27 "` (a `str`, including a unit), NOT `2.27` (a `float`).

You can easily extract the value by turning the `str` into a `Quantity` (see Converting Units), or by using `python` string operations to split the value and the units, e.g.

[9]:
```
# string operations
print(float("2.27 ".split(" ")[0]))
```

```
2.27
```

[10]:
```
# pint Quantity
from pyEQL import ureg
print(ureg.Quantity("2.27 ").magnitude)
```

```
2.27
```

Query multiple documents

`query_one` only returns a single document (a single `dict`). You can instead use `query` with exactly the same syntax to return a generator of all documents that match your query.

[11]:
```
# all documents with a charge of +2, returning only the formulas
IonDB.query({"charge":2}, ["formula","molecular_weight"])
```

[11]:
```
<generator object MongoStore.query at 0x7f0e84427ed0>
```

A generator is not very useful unless we turn it into a `list`. You can do this with `list()` or with a list comprehension

[12]:
```
# using list()
list(IonDB.query({"charge":2}, ["formula","molecular_weight"]))
```

[12]:
```
[{'formula': 'Ag[+2]',
  'molecular_weight': '107.8682 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e6e')},
 {'formula': 'Au[+2]',
  'molecular_weight': '196.966569 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e76')},
 {'formula': 'Ba[+2]',
  'molecular_weight': '137.327 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e83')},
 {'formula': 'Be[+2]',
  'molecular_weight': '9.012182 g/mol',
```

(continues on next page)

(continued from previous page)

```
 '_id': ObjectId('654e5f131ed012c187817e85')},
{'formula': 'Ca[+2]',
 'molecular_weight': '40.078 g/mol',
 '_id': ObjectId('654e5f131ed012c187817e96')},
{'formula': 'Cd[+2]',
 'molecular_weight': '112.411 g/mol',
 '_id': ObjectId('654e5f131ed012c187817e9b')},
{'formula': 'Co[+2]',
 'molecular_weight': '58.933195 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ea9')},
{'formula': 'Cr[+2]',
 'molecular_weight': '51.9961 g/mol',
 '_id': ObjectId('654e5f131ed012c187817eae')},
{'formula': 'Cu[+2]',
 'molecular_weight': '63.546 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ebe')},
{'formula': 'Dy[+2]',
 'molecular_weight': '162.5 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ec0')},
{'formula': 'Eu[+2]',
 'molecular_weight': '151.964 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ec5')},
{'formula': 'Fe[+2]',
 'molecular_weight': '55.845 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ecc')},
{'formula': 'Ge[+2]',
 'molecular_weight': '72.64 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ed1')},
{'formula': 'Hg[+2]',
 'molecular_weight': '200.59 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ef1')},
{'formula': 'In[+2]',
 'molecular_weight': '114.818 g/mol',
 '_id': ObjectId('654e5f131ed012c187817ef7')},
{'formula': 'Mg[+2]',
 'molecular_weight': '24.305 g/mol',
 '_id': ObjectId('654e5f131ed012c187817f26')},
{'formula': 'Mn[+2]',
 'molecular_weight': '54.938045 g/mol',
 '_id': ObjectId('654e5f131ed012c187817f2a')},
{'formula': 'Nd[+2]',
 'molecular_weight': '144.242 g/mol',
 '_id': ObjectId('654e5f131ed012c187817f4a')},
{'formula': 'Ni[+2]',
 'molecular_weight': '58.6934 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f4f')},
{'formula': 'Pb[+2]',
 'molecular_weight': '207.2 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f62')},
{'formula': 'Pd[+2]',
 'molecular_weight': '106.42 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f63')},
```

(continues on next page)

(continued from previous page)

```
{'formula': 'Po[+2]',
 'molecular_weight': '210.0 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f65')},
{'formula': 'Pr[+2]',
 'molecular_weight': '140.90765 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f67')},
{'formula': 'Pt[+2]',
 'molecular_weight': '195.084 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f69')},
{'formula': 'Ra[+2]',
 'molecular_weight': '226.0 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f6b')},
{'formula': 'Ru[+2]',
 'molecular_weight': '101.07 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f7b')},
{'formula': 'Sc[+2]',
 'molecular_weight': '44.955912 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f87')},
{'formula': 'Sm[+2]',
 'molecular_weight': '150.36 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f8e')},
{'formula': 'Sn[+2]',
 'molecular_weight': '118.71 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f90')},
{'formula': 'Sr[+2]',
 'molecular_weight': '87.62 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f97')},
{'formula': 'Tc[+2]',
 'molecular_weight': '98.0 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f9b')},
{'formula': 'Ti[+2]',
 'molecular_weight': '47.867 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f9f')},
{'formula': 'Tm[+2]',
 'molecular_weight': '168.93421 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fa7')},
{'formula': 'UO2[+2]',
 'molecular_weight': '270.02771 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fad')},
{'formula': 'V[+2]',
 'molecular_weight': '50.9415 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fb2')},
{'formula': 'Yb[+2]',
 'molecular_weight': '173.04 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fba')},
{'formula': 'Zn[+2]',
 'molecular_weight': '65.409 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fc2')}]
```

[13]:
```
# using a comprehension
[doc for doc in IonDB.query({"charge":2}, ["formula","molecular_weight"])]
```

[13]:
```
[{'formula': 'Ag[+2]',
  'molecular_weight': '107.8682 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e6e')},
 {'formula': 'Au[+2]',
  'molecular_weight': '196.966569 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e76')},
 {'formula': 'Ba[+2]',
  'molecular_weight': '137.327 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e83')},
 {'formula': 'Be[+2]',
  'molecular_weight': '9.012182 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e85')},
 {'formula': 'Ca[+2]',
  'molecular_weight': '40.078 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e96')},
 {'formula': 'Cd[+2]',
  'molecular_weight': '112.411 g/mol',
  '_id': ObjectId('654e5f131ed012c187817e9b')},
 {'formula': 'Co[+2]',
  'molecular_weight': '58.933195 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ea9')},
 {'formula': 'Cr[+2]',
  'molecular_weight': '51.9961 g/mol',
  '_id': ObjectId('654e5f131ed012c187817eae')},
 {'formula': 'Cu[+2]',
  'molecular_weight': '63.546 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ebe')},
 {'formula': 'Dy[+2]',
  'molecular_weight': '162.5 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ec0')},
 {'formula': 'Eu[+2]',
  'molecular_weight': '151.964 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ec5')},
 {'formula': 'Fe[+2]',
  'molecular_weight': '55.845 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ecc')},
 {'formula': 'Ge[+2]',
  'molecular_weight': '72.64 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ed1')},
 {'formula': 'Hg[+2]',
  'molecular_weight': '200.59 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ef1')},
 {'formula': 'In[+2]',
  'molecular_weight': '114.818 g/mol',
  '_id': ObjectId('654e5f131ed012c187817ef7')},
 {'formula': 'Mg[+2]',
  'molecular_weight': '24.305 g/mol',
  '_id': ObjectId('654e5f131ed012c187817f26')},
 {'formula': 'Mn[+2]',
  'molecular_weight': '54.938045 g/mol',
  '_id': ObjectId('654e5f131ed012c187817f2a')},
 {'formula': 'Nd[+2]',
  'molecular_weight': '144.242 g/mol',
```

(continues on next page)

(continued from previous page)

```
 '_id': ObjectId('654e5f131ed012c187817f4a')},
{'formula': 'Ni[+2]',
 'molecular_weight': '58.6934 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f4f')},
{'formula': 'Pb[+2]',
 'molecular_weight': '207.2 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f62')},
{'formula': 'Pd[+2]',
 'molecular_weight': '106.42 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f63')},
{'formula': 'Po[+2]',
 'molecular_weight': '210.0 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f65')},
{'formula': 'Pr[+2]',
 'molecular_weight': '140.90765 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f67')},
{'formula': 'Pt[+2]',
 'molecular_weight': '195.084 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f69')},
{'formula': 'Ra[+2]',
 'molecular_weight': '226.0 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f6b')},
{'formula': 'Ru[+2]',
 'molecular_weight': '101.07 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f7b')},
{'formula': 'Sc[+2]',
 'molecular_weight': '44.955912 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f87')},
{'formula': 'Sm[+2]',
 'molecular_weight': '150.36 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f8e')},
{'formula': 'Sn[+2]',
 'molecular_weight': '118.71 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f90')},
{'formula': 'Sr[+2]',
 'molecular_weight': '87.62 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f97')},
{'formula': 'Tc[+2]',
 'molecular_weight': '98.0 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f9b')},
{'formula': 'Ti[+2]',
 'molecular_weight': '47.867 g/mol',
 '_id': ObjectId('654e5f141ed012c187817f9f')},
{'formula': 'Tm[+2]',
 'molecular_weight': '168.93421 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fa7')},
{'formula': 'UO2[+2]',
 'molecular_weight': '270.02771 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fad')},
{'formula': 'V[+2]',
 'molecular_weight': '50.9415 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fb2')},
```

(continues on next page)

(continued from previous page)

```
{'formula': 'Yb[+2]',
 'molecular_weight': '173.04 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fba')},
{'formula': 'Zn[+2]',
 'molecular_weight': '65.409 g/mol',
 '_id': ObjectId('654e5f141ed012c187817fc2')}]
```

Counting Documents

You can use `count()` to see how many documents the database contains

[14]:
```
IonDB.count()
```

[14]:
```
346
```

Count works with queries, too.

[15]:
```
# number of documents with a charge of -3
IonDB.count({"charge": -3})
```

[15]:
```
7
```

More Advanced Query Syntax

Match multiple items with `$in`

If you want to query documents that match *any one of a set of values*, use `$in` with a `list` of possible values. Note that the `$in` operator and your `list` constitute their own dictionary, e.g. `{"$in":<list>}`. This entire dictionary is the "value" of your query for the associated field. For example:

[16]:
```
# all alkali cations
IonDB.count({"formula":
             {"$in": ["Li[+1]", "Na[+1]", "K[+1]", "Rb[+1]", "Cs[+1]"]}
            }
            )
```

[16]:
```
5
```

Greater than or less than - `$gt` / `$gte` / `$lt` / `$lte`

In a similar manner, you can query fields whose values are greater than / less than or equal to some value

[17]:
```
# all solutes with a charge less than 0
IonDB.count({"charge":
             {"$lt": 0}
            }
            )
```

[17]:
```
76
```

[18]:
```
# all solutes with a charge greater than or equal to 1
IonDB.count({"charge":
             {"$gte": 1}
            }
            )
```

[18]:
```
108
```

Unique Values

It's often useful to understand how many unique values of a field there are. To do so, use `distinct()` with any field name

[19]:
```
# list of all unique `formula`
IonDB.distinct('formula')
```

[19]:
```
['U(ClO5)2(aq)',
 'LiClO4(aq)',
 'Sb(OH)6[-1]',
 'Ba[+2]',
 'RbNO3(aq)',
 'KBrO3(aq)',
 'H3O[+1]',
 'CsNO2(aq)',
 'Re[+1]',
 'KHC2O.1H2O(aq)',
 'Ni[+3]',
 'H8S(NO2)2(aq)',
 'Sm[+2]',
 'B(OH)4[-1]',
 'CoI2(aq)',
 'ZnBr2(aq)',
 'Sn[+2]',
 'USO6(aq)',
 'Ir[+3]',
 'Ag(CN)2[-1]',
 'KNO3(aq)',
 'Ga[+3]',
 'Zn(NO3)2(aq)',
 'NaHC3.2H2O(aq)',
 'Ni(NO3)2(aq)',
 'S[-2]',
 'HS[-1]',
 'Eu[+2]',
 'ZnSO4(aq)',
 'BeSO4(aq)',
 'MnO4[-1]',
 'K2CO3(aq)',
 'Pa[+3]',
 'SrI2(aq)',
 'FeCl2(aq)',
 'Eu(NO3)3(aq)',
```

(continues on next page)

(continued from previous page)

```
'NaClO4(aq)',
'Zn[+2]',
'SeO4[-1]',
'NaCrO4(aq)',
'CsOH(aq)',
'Na3PO4(aq)',
'KCSN(aq)',
'HSO4[-1]',
'Mn[+3]',
'H4NClO4(aq)',
'NiSO4(aq)',
'IO4[-1]',
'Sr(ClO4)2(aq)',
'SeO4[-2]',
'Ag[+1]',
'LiI(aq)',
'SiF6[-2]',
'HF2[-1]',
'CoBr2(aq)',
'Pr[+3]',
'BaBr2(aq)',
'ClO2[-1]',
'MgBr2(aq)',
'Ho[+3]',
'Be[+2]',
'H2O(aq)',
'Po[+2]',
'P2O7[-4]',
'RbCl(aq)',
'K[+1]',
'ClO4[-1]',
'Mg(ClO4)2(aq)',
'NdCl3(aq)',
'Au[+1]',
'Rb2SO4(aq)',
'Na2PHO4(aq)',
'Th[+4]',
'Fe[+3]',
'Ra[+2]',
'Tl(NO3)3(aq)',
'Rb[+1]',
'KF(aq)',
'Gd[+3]',
'NiCl2(aq)',
'Rh[+3]',
'Ag[+3]',
'Cd(ClO4)2(aq)',
'FeCl3(aq)',
'NO3[-1]',
'MoO4[-2]',
'Tl[+3]',
'CuSO4(aq)',
```

(continues on next page)

(continued from previous page)

```
'Tm[+3]',
'OH[-1]',
'Zn(ClO4)2(aq)',
'Th(NO3)4(aq)',
'HNO3(aq)',
'AgNO3(aq)',
'Cr(NO3)3(aq)',
'Tl(NO2)3(aq)',
'CaI2(aq)',
'Pt[+2]',
'U(NO4)2(aq)',
'LiNO2(aq)',
'Na2CO3(aq)',
'Np[+4]',
'LiNO3(aq)',
'VO2[+1]',
'KCrO4(aq)',
'PO4[-3]',
'Ca(NO3)2(aq)',
'Li[+1]',
'SrCl2(aq)',
'KPO3.1H2O(aq)',
'CH3COO[-1]',
'PrCl3(aq)',
'In[+3]',
'ZnI2(aq)',
'SmCl3(aq)',
'KI(aq)',
'K3Fe(CN)6(aq)',
'CdSO4(aq)',
'GdCl3(aq)',
'Os[+3]',
'LiHC2O.1H2O(aq)',
'Sr(NO3)2(aq)',
'H2SO4(aq)',
'Hg[+2]',
'NaNO2(aq)',
'Cd[+2]',
'H5N2[+1]',
'Mg(NO3)2(aq)',
'KClO3(aq)',
'P(OH)2[-1]',
'Sn[+4]',
'Tm[+2]',
'U(ClO)2(aq)',
'HCO2[-1]',
'BO2[-1]',
'KBr(aq)',
'K2SO4(aq)',
'SeO3[-1]',
'Ta[+3]',
'YNO3(aq)',
```

(continues on next page)

(continued from previous page)

```
'Cu[+1]',
'Er[+3]',
'Al[+3]',
'HSO3[-1]',
'Tl[+1]',
'BrO3[-1]',
'Li2SO4(aq)',
'Co[+2]',
'ZnCl2(aq)',
'HO2[-1]',
'I[-1]',
'Au[+3]',
'CSN[-1]',
'NaHCO2(aq)',
'Ca[+2]',
'P(HO2)2[-1]',
'Ba(NO3)2(aq)',
'Dy[+2]',
'Cs2SO4(aq)',
'F[-1]',
'Pb[+2]',
'EuCl3(aq)',
'Ca(ClO4)2(aq)',
'Al2(SO4)3(aq)',
'PH9(NO2)2(aq)',
'RbBr(aq)',
'CuCl2(aq)',
'Co(H3N)6[-3]',
'Pb(NO3)2(aq)',
'CSeN[-1]',
'Ce[+3]',
'RbOH(aq)',
'P3O10[-5]',
'NaOH(aq)',
'Sm[+3]',
'Yb[+3]',
'C2N3[-1]',
'BaC4O.3H2O(aq)',
'Fe(CN)6[-3]',
'RbNO2(aq)',
'Re[+3]',
'LaCl3(aq)',
'CrO4[-2]',
'H[+1]',
'K3PO4(aq)',
'Ce[+4]',
'Cu[+2]',
'H2CO3(aq)',
'BaCl2(aq)',
'NaBrO3(aq)',
'Zr[+4]',
'CsI(aq)',
```

(continues on next page)

(continued from previous page)

```
'CoCl2(aq)',
'Ac[+3]',
'Ti[+2]',
'Nd(NO3)3(aq)',
'NaBr(aq)',
'La(NO3)3(aq)',
'MgC4O.3H2O(aq)',
'PO3[-1]',
'CrCl3(aq)',
'U[+3]',
'HCl(aq)',
'Tc[+2]',
'Au(CN)4[-1]',
'HOsO5[-1]',
'WO4[-1]',
'PHO4[-2]',
'ClO3[-1]',
'K4Fe(CN)6(aq)',
'Br[-0.33333333]',
'Sr[+2]',
'Cr[+3]',
'UO2[+2]',
'Ni[+2]',
'Tl(ClO4)3(aq)',
'Pu[+4]',
'NaHC2O.1H2O(aq)',
'MgI2(aq)',
'TlH(C3O)2.4H2O(aq)',
'Co(CN)6[-3]',
'K2PHO4(aq)',
'Mn[+2]',
'NaNO3(aq)',
'ScCl3(aq)',
'CeCl3(aq)',
'MnSO4(aq)',
'Ru[+3]',
'CsNO3(aq)',
'HCO3[-1]',
'H4NCl(aq)',
'Ag[+2]',
'Nb[+3]',
'CNO[-1]',
'H4IN(aq)',
'In[+1]',
'Ge[+2]',
'HI(aq)',
'ReO4[-1]',
'MgCl2(aq)',
'CsF(aq)',
'N[-0.33333333]',
'HClO4(aq)',
'CsCl(aq)',
```

(continues on next page)

(continued from previous page)

```
'KHCO3(aq)',
'La[+3]',
'Ba(ClO4)2(aq)',
'Bi[+3]',
'CaBr2(aq)',
'Yb[+2]',
'CaCl2(aq)',
'UO2[+1]',
'Mo[+3]',
'KClO4(aq)',
'KOH(aq)',
'LiBr(aq)',
'SO3[-1]',
'NaPO3.1H2O(aq)',
'AsO4[-3]',
'Br[-1]',
'Dy[+3]',
'IO3[-1]',
'Sc[+3]',
'H2SNO3[-1]',
'Np[+3]',
'Fe(CN)6[-4]',
'RbHC2O.1H2O(aq)',
'Nd[+2]',
'Pr[+2]',
'HBr(aq)',
'MgSO4(aq)',
'PO3F[-2]',
'RbI(aq)',
'Cu[+3]',
'Pm[+3]',
'H4BrN(aq)',
'MnCl2(aq)',
'CsHC2O.1H2O(aq)',
'Y[+3]',
'Hf[+4]',
'Na[+1]',
'H5C6O7[-3]',
'Co(NO3)2(aq)',
'NaCSN(aq)',
'NaHCO3(aq)',
'CsBr(aq)',
'W[+3]',
'NO2[-1]',
'V[+2]',
'SO4[-1]',
'B(OH)3(aq)',
'KCl(aq)',
'Cl[-1]',
'SrBr2(aq)',
'PF6[-1]',
'YCl3(aq)',
```

(continues on next page)

(continued from previous page)

```
'Pb(ClO4)2(aq)',
'Co[+3]',
'S2O3[-2]',
'H4N2O3(aq)',
'Cr[+2]',
'NaF(aq)',
'H4N[+1]',
'Au(CN)2[-1]',
'SO4[-2]',
'Tc[+3]',
'NaCl(aq)',
'Eu[+3]',
'Ru[+2]',
'V[+3]',
'Cu(NO3)2(aq)',
'Fe[+2]',
'Nd[+3]',
'SO3[-2]',
'BF4[-1]',
'Sb(HO2)2[-1]',
'B(H5C6)4[-1]',
'Mg[+2]',
'Cd(NO2)2(aq)',
'SO2[-1]',
'NaI(aq)',
'Ti[+3]',
'Cs[+1]',
'HSeO3[-1]',
'LiOH(aq)',
'LiCl(aq)',
'RbF(aq)',
'WO4[-2]',
'Pd[+2]',
'Tb[+3]',
'In[+2]',
'Re[-1]',
'Na2S2O3(aq)',
'KNO2(aq)',
'TcO4[-1]',
'U[+4]',
'BaI2(aq)',
'CO3[-2]',
'H4SNO4(aq)',
'CN[-1]',
'Sc[+2]',
'Cd(NO3)2(aq)',
'Na2SO4(aq)',
'IrO4[-1]',
'Lu[+3]',
'Au[+2]']
```

4.4 Installing

4.4.1 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

4.4.2 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'

Warning: If you are using a Mac with an Apple M1, M2, etc. chip (i.e., Arm64 architecture), some features of `pyEQL` will be unavailable. Specifically, anything which depends on PHREEQC (e.g., the `equilibrate` method in the native engine and the entire *`phreeqc` engine*) will not work. This is because `phreeqpython` is currently not available for this platform. All other functions of `pyEQL` should work as expected.

Feel free to post your experiences or proposed solutions at https://github.com/KingsburyLab/pyEQL/issues/109

4.4.3 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.

4.4.4 Installing the development branch

If you want to use the bleeding edge version before it is released to PyPi instead of the latest stable release, you can substitute the following for the above 'pip install' command:

```
pip install git+https://github.com/KingsburyLab/pyEQL.git@main
```

4.4.5 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/KingsburyLab/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.

4.5 Creating a Solution

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. There are several ways to create a Solution.

4.5.1 Empty solution

With no input arguments, you get an empty Solution at pH 7 and 1 atm pressure.

```
>>> from pyEQL import Solution
>>> s = Solution()
>>> print(s)
Volume: 1.000 l
Pressure: 1.000 atm
Temperature: 298.150 K
Components: ['H2O(aq)', 'H[+1]', 'OH[-1]']
```

4.5.2 Manual Creation

Typically, you will create a solution by specifying a list of solutes. Solutes are passed as a dict with amounts given **as strings** that include units (see *units*). Any unit that can be understood by get_amount is valid.

```
>>> from pyEQL import Solution
>>> s = Solution({"Na+": "0.5 mol/L", "Cl-": "0.5 mol/L"})
```

You can also specify conditions such as temperature, pressure, pH, and pE (redox potential).

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'},
                  pH=8,
                  temperature = '20 degC',
                  volume='8 L',
                  pE = 4,
                  )
```

4.5.3 Using a preset

Alternatively, you can use the Solution.from_preset() classmethod to easily create common solutions like seawater:

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

4.5.4 From a dictionary

If you have *converted a `Solution` to a `dict`*, you can re-instantiate it using the `Solution.from_dict()` class method.

4.5.5 From a file

If you *save a `Solution` to a `.json` file*, you can *recreate it* using monty.serialization.loadfn

```
>>> from monty.serialization import loadfn
>>> s = loadfn('test.json')
print(s)
Volume: 1.000 l
Pressure: 1.000 atm
Temperature: 298.150 K
Components: ['H2O(aq)', 'H[+1]', 'OH[-1]']
```

4.6 Writing 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 *property database* for parameters (e.g. diffusion coefficient) that are used in subsequent calculations.

4.6.1 How to Enter Valid Chemical Formulas

Generally speaking, type the chemical formula of your solute the "normal" way and pyEQL should be able to interpret 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	"Na2(SO4)" or "Na2SO4"	"Na(SO4)(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]`

4.6.2 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>)
...
```

4.6.3 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).

4.7 Converting Units

pyEQL uses pint to automatically interpret and convert units. For this reason, many quantitative arguments are passed to functions **as strings** rather than numbers. For example, to specify temperature, you pass `temperature='298 K'` and NOT `temperature=298`.

4.7.1 Quantity objects

Most `Solution` class methods return `pint Quantity` objects.

```
>>> from pyEQL import Solution
>>> s = Solution()
>>> s.pressure
<Quantity(1, 'standard_atmosphere')>
```

If you want to create a simple `Quantity` not attached to a `Solution`, you can do so as follows:

```
>>> from pyEQL import ureg
>>> q = ureg.Quantity('1 m')
```

`Quantity` objects have three important attributes: `magnitude`, `units`, and `dimensionality`. To get the numerical value, call magnitude

```
>>> from pyEQL import ureg
>>> q = ureg.Quantity('1 m')
>>> q.magnitude
1
```

Similarly, to get the units, call `units`

```
>>> from pyEQL import ureg
>>> q = ureg.Quantity('1 m')
>>> q.units
<Unit('meter')>
```

To convert from one unit to another, use `to()`:

```
>>> from pyEQL import ureg
>>> q = ureg.Quantity('1 m')
>>> q.to('ft')
<Quantity(3.2808399, 'foot')>
```

If you encounter a `DimensionalityError` when working with pyEQL, it probably means you are trying to do an operation on two quantities with incompatible units (or perhaps on a `Quantity` and a regular `float` or `int`). For example, you can't convert `m` into `m**3`:

```
>>> from pyEQL import ureg
>>> q = ureg.Quantity('1 m')
>>> q.to('m^3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/ryan/mambaforge/envs/pbx/lib/python3.10/site-packages/pint/facets/plain/
↪quantity.py", line 517, in to
    magnitude = self._convert_magnitude_not_inplace(other, *contexts, **ctx_kwargs)
  File "/home/ryan/mambaforge/envs/pbx/lib/python3.10/site-packages/pint/facets/plain/
↪quantity.py", line 462, in _convert_magnitude_not_inplace
    return self._REGISTRY.convert(self._magnitude, self._units, other)
  File "/home/ryan/mambaforge/envs/pbx/lib/python3.10/site-packages/pint/facets/plain/
↪registry.py", line 961, in convert
    return self._convert(value, src, dst, inplace)
  File "/home/ryan/mambaforge/envs/pbx/lib/python3.10/site-packages/pint/facets/context/
↪registry.py", line 403, in _convert
    return super()._convert(value, src, dst, inplace)
  File "/home/ryan/mambaforge/envs/pbx/lib/python3.10/site-packages/pint/facets/
↪nonmultiplicative/registry.py", line 254, in _convert
    return super()._convert(value, src, dst, inplace)
  File "/home/ryan/mambaforge/envs/pbx/lib/python3.10/site-packages/pint/facets/plain/
↪registry.py", line 1000, in _convert
    raise DimensionalityError(src, dst, src_dim, dst_dim)
pint.errors.DimensionalityError: Cannot convert from 'meter' ([length]) to 'meter ** 3'␣
↪([length] ** 3)
```

Refer to the pint documentation for more etails about working with `Quantity`.

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 `ureq = UnitRegistry()` does the same thing.

Important: 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.

4.7.2 Custom Units

`pyEQL` extends the `pint` unit library to include some additional units that are commonly encountered in solution chemistry.

4.8 Getting Concentrations

4.8.1 Get the amount of a specific solute

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

```
>>> from pyEQL import Solution
>>> s = Solution({"Mg+2": "0.5 mol/L", "Cl-": "1.0 mol/L"})
>>> s.get_amount('Mg[+2]', 'mol')
<Quantity(0.5, 'mole')>
```

`get_amount` is highly flexible with respect to the types of units it can interpret. You can request amounts in moles, mass, or equivalents (i.e., charge-weighted moles) per unit of mass or volume.

```
>>> s.get_amount('Mg[+2]', 'M')
<Quantity(0.5, 'molar')>
>>> s.get_amount('Mg[+2]', 'm')
<Quantity(0.506124103, 'mole / kilogram')>
>>> s.get_amount('Mg[+2]', 'eq/L')
<Quantity(1.0, 'mole / liter')>
>>> s.get_amount('Mg[+2]', 'ppm')
<Quantity(12152.5, 'milligram / liter')>
>>> s.get_amount('Mg[+2]', 'ppb')
<Quantity(12152500.0, 'microgram / liter')>
>>> s.get_amount('Mg[+2]', 'ppt')
<Quantity(1.21525e+10, 'nanogram / liter')>
```

Important: The unit `'ppt'` is ambiguous in the water community. To most researchers, it means "parts per trillion" or ng/L, while to many engineers and operators it means "parts per THOUSAND" or g/L. `pyEQL` interprets `ppt` as **parts per trillion**.

You can also request dimensionless concentrations as weight percent (`'%'`), mole fraction (`'fraction'`) or the total *number* of particles in the solution (`'count'`, useful for setting up simulation boxes).

```
>>> s.get_amount('Mg[+2]', '%')
<Quantity(1.17358141, 'dimensionless')>
>>> s.get_amount('Mg[+2]', 'fraction')
<Quantity(0.00887519616, 'dimensionless')>
>>> s.get_amount('Mg[+2]', 'count')
<Quantity(3.01107038e+23, 'dimensionless')>
```

4.8.2 See all components in the solution

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. `components` is reverse sorted, with the most predominant component (i.e., the solvent) listed first.

```
>>> from pyEQL import Solution
>>> s = Solution({"Mg+2": "0.5 mol/L", "Cl-": "1.0 mol/L"})
>>> s.components
{'H2O(aq)': 54.83678280993063, 'Cl[-1]': 1.0, 'Mg[+2]': 0.5, 'H[+1]': 1e-07, 'OH[-1]':␣
↪1e-07}
```

Similarly, you can use the properties `anions`, `cations`, `neutrals`, and `solvent` to retrieve subsets of `components`:

```
>>> s.anions
{'Cl[-1]': 1.0, 'OH[-1]': 1e-07}
>>> s.cations
{'Mg[+2]': 0.5, 'H[+1]': 1e-07}
>>> s.neutrals
{'H2O(aq)': 54.83678280993063}
>>> s.solvent
'H2O(aq)'
```

Like `components`, all of the above dicts are sorted in order of decreasing amount.

4.8.3 Salt vs. Solute Concentrations

Sometimes the concentration of a dissolved *salt* (e.g., MgCl2) is of greater interest than the concentrations of the individual solutes (Mg+2 and Cl-). `pyEQL` has the ability to interpret a `Solution` composition and represent it as a mixture of salts.

To retrieve only *the predominant salt* (i.e., the salt with the highest concentration), use `get_salt`. This returns a `Salt` object with several useful attributes.

```
>>> from pyEQL import Solution
>>> s = Solution({"Mg+2": "0.4 mol/L", "Na+": "0.1 mol/L", "Cl-": "1.0 mol/L"})
>>> s.get_salt()
<pyEQL.salt_ion_match.Salt object at 0x7f0ded09fd30>
>>> s.get_salt().formula
'MgCl2'
>>> s.get_salt().anion
'Cl[-1]'
>>> s.get_salt().z_cation
```

(continues on next page)

(continued from previous page)

```
2.0
>>> s.get_salt().nu_anion
2
```

To see a `dict` of all the salts in given solution, use `get_salt_dict()`. This method returns a dict keyed by the salt's formula, where the values are `Salt` objects converted into dictionaries. All the usual attributes like `anion`, `z_cation` etc. are accessible in the corresponding keys. Each value also contains a `mol` key giving the moles of the salt present.

```
>>> from pyEQL import Solution
>>> s = Solution({"Mg+2": "0.4 mol/L", "Na+": "0.1 mol/L", "Cl-": "1.0 mol/L"})
>>> s.get_salt_dict()
{'MgCl2': {'@module': 'pyEQL.salt_ion_match',
           '@class': 'Salt', '@version': '0.5.2',
           'cation': 'Mg[+2]',
           'anion': 'Cl[-1]',
           'mol': 0.4},
 'NaCl': {'@module': 'pyEQL.salt_ion_match',
          '@class': 'Salt', '@version': '0.5.2',
          'cation': 'Na[+1]',
          'anion': 'Cl[-1]',
          'mol': 0.1},
 'NaOH': {'@module': 'pyEQL.salt_ion_match',
          '@class': 'Salt', '@version': '0.5.2',
          'cation': 'Na[+1]',
          'anion': 'OH[-1]',
          'mol': 1e-07}
}
```

Refer to the *Salt Matching module reference* for more details.

4.8.4 Total Element Concentrations

"Total" concentrations (i.e., concentrations of all species containing a particular element) are important for certain types of equilibrium calculations. These can be retrieved via `get_total_amount`. `get_total_amount` takes an element name as the first argument, and a unit as the second.

```
>>> from pyEQL import Solution
>>> s = Solution({"Mg+2": "0.5 mol/L", "Cl-": "1.0 mol/L"})
>>> s.equilibrate()
>>> s.components
{'H2O(aq)': 54.85346847938828, 'Cl[-1]': 0.9186683796593457, 'Mg[+2]': 0.
↪41866839204646417, 'MgCl[+1]': 0.08133160795194606, 'OH[-1]': 1.4679440802358093e-07,
↪'H[+1]': 1.1833989847708719e-07, 'HCl(aq)': 1.2388705241250352e-08, 'MgOH[+1]': 3.
↪9747494391744955e-13, 'O2(aq)': 7.027122927701743e-25, 'HClO(aq)': 1.5544872892067526e-
↪27, 'ClO[-1]': 6.339364938003202e-28, 'H2(aq)': 5.792559717610837e-35, 'ClO2[-1]': 0.0,
↪ 'ClO3[-1]': 0.0, 'ClO4[-1]': 0.0, 'HClO2(aq)': 0.0}
>>> s.get_total_amount('Mg', 'mol')
<Quantity(0.5, 'mole')>
```

4.8.5 Elements present in a `Solution`

If you just want to know the elements present in the `Solution`, use `elements`. This returns a list of elements, sorted alphabetically.

```
>>> from pyEQL import Solution
>>> s = Solution({"Mg+2": "0.5 mol/L", "Cl-": "1.0 mol/L"})
>>> s.elements
['Cl', 'H', 'Mg', 'O']
```

4.9 Arithmetic Operations

4.9.1 Addition and Subtraction

You can use the + operator to mix (combine) two solutions. The moles of each component in the two solutions will be added together, and the volume of the mixed solution will be *approximately* equal to the sum of the two volumes, depending on the electrolyte modeling engine used. The pressure and temperature of the mixed solution are computed as volume-weighted averages.

```
>>> from pyEQL import Solution
>>> s1 = Solution({"Na+": "0.5 mol/L", "Cl-": "0.5 mol/L"})
>>> s2 = Solution({"Na+": "0.1 mol/L", "Cl-": "0.1 mol/L"})
>>> s1+s2
<pyEQL.solution.Solution object at 0x7f171aee3af0>
>>> (s1+s2).get_amount('Na+', 'mol')
<Quantity(0.6, 'mole')>
>>> (s1+s2).volume
<Quantity(1.99989659, 'liter')>
```

Note: Both `Solution` involved in an addition operation must use the same *electrolyte modeling engine*.

Subtraction is not implemented and will raise a `NotImplementedError`.

```
>>> s1-s2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/ryan/mambaforge/envs/pbx/code/pyEQL/src/pyEQL/solution.py", line 2481, in _
↪_sub__
    raise NotImplementedError("Subtraction of solutions is not implemented.")
NotImplementedError: Subtraction of solutions is not implemented.
```

4.9.2 Multiplication and Division

The * and / operators scale the volume and all component amounts by a factor.

```
from pyEQL import Solution
>>> s = Solution({"Na+": "0.2 mol/L", "Cl-": "0.2 mol/L"})
>>> s.volume
<Quantity(1, 'liter')>
>>> s.get_amount('Cl-', 'mol')
<Quantity(0.2, 'mole')>
>>> s*=1.5
<Quantity(1.5, 'liter')>
s.get_amount('Cl-', 'mol')
<Quantity(0.3, 'mole')>
```

The modulo operator // is not implemented.

4.10 Saving and Loading from Files

4.10.1 Serialization to `dict`

Any `Solution` can be converted into a `dict` by calling `as_dict`:

```
>>> from pyEQL import Solution
>>> s = Solution({"Na+": "0.5 mol/L", "Cl-": "0.5 mol/L"})
>>> s.as_dict()
{'@module': 'pyEQL.solution', '@class': 'Solution', '@version': '0.5.2', 'solutes': {
↪'H2O(aq)': '55.34455402076251 mol', 'H[+1]': '1e-07 mol', 'OH[-1]': '1e-07 mol'},
↪'volume': '1 l', 'temperature': '298.15 K', 'pressure': '1 atm', 'pH': 7.0, 'pE': 8.5,
↪'balance_charge': None, 'solvent': 'H2O(aq)', 'engine': 'native', 'database': {'@module
↪': 'maggma.stores.mongolike', '@class': 'JSONStore', '@version': '0.19.1.post1.
↪dev1792+g0517496', 'paths': ['/home/ryan/mambaforge/envs/pbx/code/pyEQL/src/pyEQL/
↪database/pyeql_db.json'], 'read_only': True, 'serialization_option': None,
↪'serialization_default': None, 'key': 'formula'}}
```

This `dict` can be stored in a database or used to *recreate the `Solution`* using `from_dict()`.

4.10.2 Saving to a `.json` file

`Solution` can be serialized (and later recreated from) a `.json` file using `to_file`, which is based on `monty.serializtaion.dumpfn`.

```
>>> from pyEQL import Solution
>>> s = Solution({"Na+": "0.5 mol/L", "Cl-": "0.5 mol/L"})
>>> s.to_file('test.json')
```

4.10.3 Loading from a `.json` file

Similarly, `from_file` (based on monty.serialization.loadfn) can be used to create a `Solution` from a compatible `.json` file.

```
>>> from pyEQL import Solution
>>> s = Solution.from_file('test.json')
<pyEQL.solution.Solution object at 0x7febf742d8a0>
>>> print(s)
Volume: 1.000 l
Pressure: 1.000 atm
Temperature: 298.150 K
Components: ['H2O(aq)', 'H[+1]', 'OH[-1]']
```

4.11 Electrolyte Modeling Engines

4.11.1 Overview

Every `Solution` is instantiated with an electrolyte modeling "engine", which is a subclass of *pyEQL.engine.EOS*. The modeling engine performs three functions:

1. Calculate species activity and osmotic coefficients via `get_activity_coefficient` and `get_osmotic_coefficient`, respectively
2. Update the composition of the `Solution` (i.e., speciation) via `equilibrate`
3. Calculate the volume occupied by the solutes via `get_solute_volume`

All other calculations are performed directly by the `Solution` class and are the same, regardless of the engine selected

The purpose of this architecture is to allow `Solution` to provide a **consistent interface** for working with electrolyte solutions, but allow the underlying models to be customized as needed to particular use cases.

`pyEQL` currently supports three modeling engines: `ideal`, `native`, and `phreeqc`, which are selected via the `engine` kwarg to `Solution.__init__()`. Each engine is briefly described below.

> **Warning:** If you are using a Mac with an Apple M1, M2, etc. chip (i.e., Arm64 architecture), some features of `pyEQL` will be unavailable. Specifically, anything which depends on PHREEQC (e.g., the `equilibrate` method in the native engine and the entire *`phreeqc` engine*) will not work. This is because `phreeqpython` is currently not available for this platform. All other functions of `pyEQL` should work as expected.
>
> Feel free to post your experiences or proposed solutions at https://github.com/KingsburyLab/pyEQL/issues/109

4.11.2 The 'native' engine (Default)

The native engine is the default choice and was the only option available prior to version 0.6.0.

Activity and osmotic coefficients

Activity coefficients are calculated using the "effetive Pitzer model" of Mistry et al. when possible. pyEQL selects parameters by identifying the predominant salt in the solution (see *Salt Matching*). The ionic strength is calculated based on all solutes, but only the predominant salt parameters are used in the Pitzer calculation.

If the required parameters are not available in the *property database*, the native engine decays gracefully through several models more appropriate for dilute solutions, including Davies, Guntelberg, and Debye-Huckel. See the *module reference* for full details.

Solute volumes

Solute volumes are also calculated according to the Pitzer model whenever parameters are available. Specifically, the apparent molar volume of the primary salt is calculated via Pitzer. The volumes of all other components (except the solvent, water) are added based on fixed partial molar volumes, if the data are available in the *property database*. If data are not available, the volume for that solute is not accounted for.

Speciation

Speciation calculations are provided by PHREEQC via phreeqpython. We use the llnl.dat PHREEQC database due to its applicability for moderate salinity water and the large number of species included (see Lu et al.). See pyEQL.equilibrium.eqiulibrate_phreeqc in the *module reference* for more details.

Warning: Speciation support was added to the native engine in v0.8.0 and should be considered experimental. Specifically, because the native engine uses a non-Pitzer PHREEQC database for speciation but uses the Pitzer model (when possible) for activity coefficients. As such, there may be subtle thermodynamic inconsistencies between the activities and the equilibrium concentrations returned by equilibrate().

4.11.3 The 'phreeqc' engine

The phreeqc engine uses phreeqpython for speciation, activity, and volume calculations. The PHREEQC engine uses the phreeqc.dat PHREEQC database by default, although it is possible to instantiate the engine with other databases such as llnl.dat, pitzer.dat, etc. See pyEQL.equilibrium.eqiulibrate_phreeqc in the *module reference* for more details.

Activity and osmotic coefficients

Activity coefficients are calculated by dividing the PHREEQC activity by the molal concentration of the solute.

Due to limitations in the phreeqpython interface, the osmotic coefficient is always returned as 1 at present.

Warning: The phreeqc engine currently returns an osmotic coefficient of 1 and solute volume of 0 for all solutions. There appear to be limitations in the phreeqpython interface that make it difficult to access these properties.

Solute volumes

Due to limitations in the `phreeqpython` interface, solute volumes are ignored (as in the `ideal` engine). More research is needed to determine whether this is consistent with intended PHREEQC behavior (when using the default database) or not.

Warning: The `phreeqc` engine currently returns an osmotic coefficient of 1 and solute volume of 0 for all solutions. There appear to be limitations in the `phreeqpython` interface that make it difficult to access these properties.

Speciation

Speciation calculations are provided by PHREEQC via `phreeqpython`.

4.11.4 The `'ideal'` engine

The `'ideal'` engine applies ideal solution behavior. Activity and osmotic coefficients are always equal to 1, solute volumes are always equal to zero, and there is no support for speciation.

4.11.5 Custom engines

The modeling engine system is designed to be extensible and customizable. To define a custom engine, you simply need to inherit from `pyEQL.engines.EOS` (or a pre-existing engine class) and then populate the abstract methods `get_activity_coefficient)`, `get_osmotic_coefficient`, `get_solute_volume`, and `equilibrate`.

Equations that implement commonly used models or the above properties (such as the Debye-Huckel and Pitzer activity models, among others) are available in `pyEQL.activity_correction` and `pyEQL.equilibrium`, respectively. The idea is that end users can "compose" custom engine classes by mixing and matching the desired functions from these modules, adding custom logic as necessary.

4.12 Property Database

pyEQL is distributed with a database of solute properties and model parameters needed to perform its 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*

4.12.1 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)
```

4.12.2 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<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}}}
```

4.12.3 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.

4.12.4 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)
```

(continues on next page)

(continued from previous page)

```
- 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]
```

(continues on next page)

(continued from previous page)

```
- 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)
```

(continues on next page)

(continued from previous page)

```
- 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)
```

(continues on next page)

(continued from previous page)

```
- 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)
```

(continues on next page)

(continued from previous page)

```
- 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]
```

(continues on next page)

(continued from previous page)

```
- 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]
```

4.13 Mixing Functions

pyEQL contains several mixing and equilibration functions that take `Solution` as arguments. pyEQL functions that take Solution objects as inputs or return Solution objects.

copyright
2013-2024 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

pyEQL.functions.**gibbs_mix**(*solution1: Solution*, *solution2: Solution*)

Return the Gibbs energy change associated with mixing two solutions.

Parameters

- **solution1** – a solution to be mixed.
- **solution2** – a solution to be mixed.

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

Notes

The Gibbs energy of mixing is calculated as follows

$$\Delta_{mix}G = \sum_i (n_c + n_d)RT \ln a_b - \sum_i n_c RT \ln a_c - \sum_i n_d RT \ln a_d$$

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.**entropy_mix**(*solution1: Solution*, *solution2: Solution*)

Return the ideal mixing entropy associated with mixing two solutions.

Parameters

- **solution1** – The two solutions to be mixed.
- **solution2** – The two solutions to be mixed.

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

Notes

The ideal entropy of mixing is calculated as follows

$$\Delta_{mix}S = \sum_i (n_c + n_d)RT \ln x_b - \sum_i n_c RT \ln x_c - \sum_i n_d RT \ln x_d$$

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.**donnan_eql**(*solution: Solution*, *fixed_charge: str*)

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

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()

4.14 `Solution` Class Reference

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

4.14.1 Solution

class pyEQL.**Solution**(*solutes: list[list[str]] | dict[str, str] | None = None*, *volume: str | None = None*, *temperature: str = '298.15 K'*, *pressure: str = '1 atm'*, *pH: float = 7*, *pE: float = 8.5*, *balance_charge: str | None = None*, *solvent: str | list = 'H2O'*, *engine: EOS | Literal['native', 'ideal', 'phreeqc'] = 'native'*, *database: str | Path | Store | None = None*, *default_diffusion_coeff: float = 1.6106e-09*)

Class representing the properties of a solution. Instances of this class contain information about the solutes, solvent, and bulk properties.

__init__(*solutes: list[list[str]] | dict[str, str] | None = None*, *volume: str | None = None*, *temperature: str = '298.15 K'*, *pressure: str = '1 atm'*, *pH: float = 7*, *pE: float = 8.5*, *balance_charge: str | None = None*, *solvent: str | list = 'H2O'*, *engine: EOS | Literal['native', 'ideal', 'phreeqc'] = 'native'*, *database: str | Path | Store | None = None*, *default_diffusion_coeff: float = 1.6106e-09*)

Instantiate a Solution from a composition.

Parameters

- **solutes** – dict, optional. Keys must be the chemical formula, while values must be str Quantity representing the amount. For example:

  {"Na+": "0.1 mol/L", "Cl-": "0.1 mol/L"}

  Note that an older "list of lists" syntax is also supported; however this will be deprecated in the future and is no longer recommended. The equivalent list syntax for the above example is

  [["Na+", "0.1 mol/L"], ["Cl-", "0.1 mol/L"]]

  Defaults to empty (pure solvent) if omitted

- **volume** – str, optional Volume of the solvent, including the unit. Defaults to '1 L' if omitted. Note that the total solution volume will be computed using partial molar volumes of the respective solutes as they are added to the solution.

- **temperature** – str, optional The solution temperature, including the ureg. Defaults to '25 degC' if omitted.

- **pressure** – Quantity, optional The ambient pressure of the solution, including the unit. Defaults to '1 atm' if omitted.

- **pH** – number, optional Negative log of H+ activity. If omitted, the solution will be initialized to pH 7 (neutral) with appropriate quantities of H+ and OH- ions

- **pE** – the pE value (redox potential) of the solution. Lower values = more reducing, higher values = more oxidizing. At pH 7, water is stable between approximately -7 to +14. The default value corresponds to a pE value typical of natural waters in equilibrium with the atmosphere.
- **balance_charge** – The strategy for balancing charge during init and equilibrium calculations. Valid options are 'pH', which will adjust the solution pH to balance charge, 'pE' which will adjust the redox equilibrium to balance charge, or the name of a dissolved species e.g. 'Ca+2' or 'Cl-' that will be added/subtracted to balance charge. If set to None, no charge balancing will be performed either on init or when equilibrate() is called. Note that in this case, equilibrate() can distort the charge balance!
- **solvent** – Formula of the solvent. Solvents other than water are not supported at this time.
- **engine** – Electrolyte modeling engine to use. See documentation for details on the available engines.
- **database** – path to a .json file (str or Path) or maggma Store instance that contains serialized SoluteDocs. *None* (default) will use the built-in pyEQL database.
- **default_diffusion_coeff** – Diffusion coefficient value in m^2/s to use in calculations when there is no diffusion coefficient for a species in the database. This affects several important property calculations including conductivity and transport number, which are related to the weighted sums of diffusion coefficients of all species. Setting this argument to zero will exclude any species that does not have a tabulated diffusion coefficient from these calculations, possibly resulting in underestimation of the conductivity and/or inaccurate transport numbers.

  Missing diffusion coefficients are especially likely in complex electrolytes containing, for example, complexes or paired species such as NaSO4[-1]. In such cases, setting default_diffusion_coeff to zero is likely to result in the above errors.

  By default, this argument is set to the diffusion coefficient of NaCl salt, 1.61x10^-9 m2/s.

Examples

```
>>> s1 = pyEQL.Solution({'Na+': '1 mol/L','Cl-': '1 mol/L'},temperature='20 degC
↪',volume='500 mL')
>>> print(s1)
Components:
Volume: 0.500 l
Pressure: 1.000 atm
Temperature: 293.150 K
Components: ['H2O(aq)', 'H[+1]', 'OH[-1]', 'Na[+1]', 'Cl[-1]']
```

balance_charge

Standardized formula of the species used for charge balancing.

water_substance

IAPWS instance describing water properties.

components

Special dictionary where keys are standardized formula and values are the moles present in Solution.

database

Store instance containing the solute property database.

solvent

Formula of the component that is set as the solvent (currently only H2O(aq) is supported).

property mass: Quantity

Return the total mass of the solution.

The mass is calculated each time this method is called.

Returns: The mass of the solution, in kg

property solvent_mass: Quantity

Return the mass of the solvent.

This property is used whenever mol/kg (or similar) concentrations are requested by get_amount()

Returns
The mass of the solvent, in kg

See also:

get_amount()

property volume: Quantity

Return the volume of the solution.

Returns
the volume of the solution, in L

Return type
Quantity

property temperature: Quantity

Return the temperature of the solution in Kelvin.

property pressure: Quantity

Return the hydrostatic pressure of the solution in atm.

property pH: float | None

Return the pH of the solution.

p(*solute: str*, *activity=True*) → float | None

Return the negative log of the activity of solute.

Generally used for expressing concentration of hydrogen ions (pH)

Parameters

- **solute** – str String representing the formula of the solute
- **activity** – bool, optional If False, the function will use the molar concentration rather than the activity to calculate p. Defaults to True.

Returns

Quantity
The negative log10 of the activity (or molar concentration if activity = False) of the solute.

property density: Quantity

Return the density of the solution.

Density is calculated from the mass and volume each time this method is called.

Returns
The density of the solution.

Return type
Quantity

property dielectric_constant: Quantity

Returns the dielectric constant of the solution.

Parameters
None –

Returns
the dielectric constant of the solution, dimensionless.

Return type
Quantity

Notes

Implements the following equation as given by Zuber et al.

$$\frac{\epsilon = \epsilon_{solvent}}{1 + \sum_i \alpha_i x_i}$$

where α_i is a coefficient specific to the solvent and ion, and x_i is the mole fraction of the ion in solution.

References

A. Zuber, L. Cardozo-Filho, V.F. Cabral, R.F. Checoni, M. Castier, An empirical equation for the dielectric constant in aqueous and nonaqueous electrolyte mixtures, Fluid Phase Equilib. 376 (2014) 116-123. doi:10.1016/j.fluid.2014.05.037.

property chemical_system: str

Return the chemical system of the Solution as a "-" separated list of elements, sorted alphabetically. For example, a solution containing CaCO3 would have a chemical system of "C-Ca-H-O".

property elements: list

Return a list of elements that are present in the solution.

For example, a solution containing CaCO3 would return ["C", "Ca", "H", "O"]

property cations: dict[str, float]

Returns the subset of *components* that are cations.

The returned dict is sorted by amount in descending order.

property anions: dict[str, float]

Returns the subset of *components* that are anions.

The returned dict is sorted by amount in descending order.

property neutrals: dict[str, float]

Returns the subset of *components* that are neutral (not charged).

The returned dict is sorted by amount in descending order.

property viscosity_dynamic: Quantity

Return the dynamic (absolute) viscosity of the solution.

Calculated from the kinematic viscosity

See also:

viscosity_kinematic

property viscosity_kinematic: Quantity

Return the kinematic viscosity of the solution.

Notes

The calculation is based on a model derived from the Eyring equation and presented in

$$\ln \nu = \ln \frac{\nu_w MW_w}{\sum_i x_i MW_i} + 15x_+^2 + x_+^3 \delta G_{123}^* + 3x_+ \delta G_{23}^* (1 - 0.05x_+)$$

Where:

$$\delta G_{123}^* = a_o + a_1 (T)^{0.75}$$

$$\delta G_{23}^* = b_o + b_1 (T)^{0.5}$$

In which ν is the kinematic viscosity, MW is the molecular weight, x_+ is the mole fraction of cations, and T is the temperature in degrees C.

The a and b fitting parameters for a variety of common salts are included in the database.

References

Vásquez-Castillo, G.; Iglesias-Silva, G. a.; Hall, K. R. An extension of the McAllister model to correlate kinematic viscosity of electrolyte solutions. Fluid Phase Equilib. 2013, 358, 44-49.

See also:

viscosity_dynamic()

property conductivity: Quantity

Compute the electrical conductivity of the solution.

Returns

The electrical conductivity of the solution in Siemens / meter.

Notes

Conductivity is calculated by summing the molar conductivities of the respective solutes.

$$EC = \frac{F^2}{RT} \sum_i D_i z_i^2 m_i = \sum_i \lambda_i m_i$$

Where D_i is the diffusion coefficient, m_i is the molal concentration, z_i is the charge, and the summation extends over all species in the solution. Alternatively, λ_i is the molar conductivity of solute i.

Diffusion coefficients D_i (and molar conductivities λ_i) are adjusted for the effects of temperature and ionic strength using the method implemented in PHREEQC >= 3.4. [**?**] [**?**] See *get_diffusion_coefficient for* further details.

References

See also:

ionic_strength get_diffusion_coefficient() get_molar_conductivity()

property ionic_strength: Quantity

Return the ionic strength of the solution.

Return the ionic strength of the solution, calculated as 1/2 * sum (molality * charge ^2) over all the ions.

Molal (mol/kg) scale concentrations are used for compatibility with the activity correction formulas.

Returns
The ionic strength of the parent solution, mol/kg.

Return type
Quantity

See also:

get_activity() get_water_activity()

Notes

The ionic strength is calculated according to:

$$I = \sum_i m_i z_i^2$$

Where m_i is the molal concentration and z_i is the charge on species i.

Examples

```
>>> s1 = pyEQL.Solution([['Na+','0.2 mol/kg'],['Cl-','0.2 mol/kg']])
>>> s1.ionic_strength
<Quantity(0.20000010029672785, 'mole / kilogram')>
```

```
>>> s1 = pyEQL.Solution([['Mg+2','0.3 mol/kg'],['Na+','0.1 mol/kg'],['Cl-','0.7␣
↪mol/kg']],temperature='30 degC')
>>> s1.ionic_strength
<Quantity(1.0000001004383303, 'mole / kilogram')>
```

property charge_balance: float

Return the charge balance of the solution.

Return the charge balance of the solution. The charge balance represents the net electric charge on the solution and SHOULD equal zero at all times, but due to numerical errors will usually have a small nonzero value. It is calculated according to:

$$CB = \sum_i C_i z_i$$

where C_i is the molar concentration, and z_i is the charge on species i.

Returns
The charge balance of the solution, in equivalents (mol of charge) per L.

Return type
float

property alkalinity: Quantity

Return the alkalinity or acid neutralizing capacity of a solution.

Returns
The alkalinity of the solution in mg/L as CaCO3

Return type
Quantity

Notes

The alkalinity is calculated according to [**?**]

$$Alk = \sum_i z_i C_B + \sum_i z_i C_A$$

Where C_B and C_A are conservative cations and anions, respectively (i.e. ions that do not participate in acid-base reactions), and z_i is their signed charge. In this method, the set of conservative cations is all Group I and Group II cations, and the conservative anions are all the anions of strong acids.

References

property hardness: Quantity

Return the hardness of a solution.

Hardness is defined as the sum of the equivalent concentrations of multivalent cations as calcium carbonate.

NOTE: at present pyEQL cannot distinguish between mg/L as CaCO3 and mg/L units. Use with caution.

Returns
The hardness of the solution in mg/L as CaCO3

Return type
Quantity

property total_dissolved_solids: Quantity

Total dissolved solids in mg/L (equivalent to ppm) including both charged and uncharged species.

The TDS is defined as the sum of the concentrations of all aqueous solutes (not including the solvent), except for H[+1] and OH[-1]].

property TDS: Quantity

Alias of *total_dissolved_solids()*.

property debye_length: Quantity

Return the Debye length of a solution.

Debye length is calculated as [**?**]

$$\kappa^{-1} = \sqrt{(\frac{\epsilon_r \epsilon_o k_B T}{(2 N_A e^2 I)})}$$

where I is the ionic strength, ϵ_r and ϵ_r are the relative permittivity and vacuum permittivity, k_B is the Boltzmann constant, and T is the temperature, e is the elementary charge, and N_A is Avogadro's number.

Returns The Debye length, in nanometers.

References

See also:

ionic_strength dielectric_constant

property bjerrum_length: Quantity

Return the Bjerrum length of a solution.

Bjerrum length represents the distance at which electrostatic interactions between particles become comparable in magnitude to the thermal energy.:math:*lambda_B* is calculated as

$$\lambda_B = \frac{e^2}{(4\pi\epsilon_r\epsilon_o k_B T)}$$

where e is the fundamental charge, ϵ_r and ϵ_r are the relative permittivity and vacuum permittivity, k_B is the Boltzmann constant, and T is the temperature.

Returns
The Bjerrum length, in nanometers.

Return type
Quantity

References

https://en.wikipedia.org/wiki/Bjerrum_length

Examples

```
>>> s1 = pyEQL.Solution()
>>> s1.bjerrum_length
<Quantity(0.7152793009386953, 'nanometer')>
```

See also:

dielectric_constant

property osmotic_pressure: Quantity

Return the osmotic pressure of the solution relative to pure water.

Returns
The osmotic pressure of the solution relative to pure water in Pa

See also:

get_water_activity get_osmotic_coefficient get_salt

Notes

Osmotic pressure is calculated based on the water activity [**?**] [**?**]

$$\Pi = -\frac{RT}{V_w} \ln a_w$$

Where Π is the osmotic pressure, V_w is the partial molar volume of water (18.2 cm**3/mol), and a_w is the water activity.

References

Examples

```
>>> s1=pyEQL.Solution()
>>> s1.osmotic_pressure
<Quantity(0.495791416, 'pascal')>
```

```
>>> s1 = pyEQL.Solution([['Na+','0.2 mol/kg'],['Cl-','0.2 mol/kg']])
>>> soln.osmotic_pressure
<Quantity(906516.7318131207, 'pascal')>
```

get_amount(*solute: str*, *units: str = 'mol/L'*) → Quantity

Return the amount of 'solute' in the parent solution.

The amount of a solute can be given in a variety of unit types. 1. substance per volume (e.g., 'mol/L', 'M') 2. equivalents (i.e., moles of charge) per volume (e.g., 'eq/L', 'meq/L') 3. substance per mass of solvent (e.g., 'mol/kg', 'm') 4. mass of substance (e.g., 'kg') 5. moles of substance ('mol') 6. mole fraction ('fraction') 7. percent by weight (%) 8. number of molecules ('count') 9. "parts-per-x" units, where ppm = mg/L, ppb = ug/L ppt = ng/L

Parameters

- **solute** – str String representing the name of the solute of interest
- **units** – str Units desired for the output. Examples of valid units are 'mol/L','mol/kg','mol', 'kg', and 'g/L' Use 'fraction' to return the mole fraction. Use '%' to return the mass percent

Returns

The amount of the solute in question, in the specified units

See also:

mass *add_amount()* *set_amount()* *get_total_amount()* *get_osmolarity()* *get_osmolality()* *get_total_moles_solute()* pyEQL.utils.interpret_units()

get_components_by_element() → dict[str, list]

Return a list of all species associated with a given element.

Elements (keys) are suffixed with their oxidation state in parentheses, e.g.,

{"Na(1.0)":["Na[+1]", "NaOH(aq)"]}

Species associated with each element are sorted in descending order of the amount present (i.e., the first species listed is the most abundant).

get_el_amt_dict()

Return a dict of Element: amount in mol.

Elements (keys) are suffixed with their oxidation state in parentheses, e.g. "Fe(2.0)", "Cl(-1.0)".

get_total_amount(*element: str*, *units: str*) → Quantity

Return the total amount of 'element' (across all solutes) in the solution.

Parameters

- **element** – The symbol of the element of interest. The symbol can optionally be followed by the oxidation state in parentheses, e.g., "Na(1.0)", "Fe(2.0)", or "O(0.0)". If no oxidation state is given, the total concentration of the element (over all oxidation states) is returned.
- **units** – str Units desired for the output. Any unit understood by *get_amount* can be used. Examples of valid units are 'mol/L','mol/kg','mol', 'kg', and 'g/L'.

Returns

The total amount of the element in the solution, in the specified units

See also:

get_amount() pyEQL.utils.interpret_units()

add_solute(*formula: str*, *amount: str*)

Primary method for adding substances to a pyEQL solution.

Parameters

- **formula** (*str*) – Chemical formula for the solute. Charged species must contain a + or - and
- **charge** (*(for polyvalent solutes) a number representing the net*) –
- **amount** (*str*) – The amount of substance in the specified unit system. The string should contain
- **g/L'.** (*both a quantity and a pint-compatible representation of a ureg. e.g. '5 mol/kg' or '0.1*) –

add_solvent(*formula: str*, *amount: str*)

Same as add_solute but omits the need to pass solvent mass to pint.

add_amount(*solute: str*, *amount: str*)

Add the amount of 'solute' to the parent solution.

Parameters

- **solute** – str String representing the name of the solute of interest
- **amount** – str quantity String representing the concentration desired, e.g. '1 mol/kg' If the units are given on a per-volume basis, the solution volume is not recalculated If the units are given on a mass, substance, per-mass, or per-substance basis, then the solution volume is recalculated based on the new composition

Returns

Nothing. The concentration of solute is modified.

set_amount(*solute: str*, *amount: str*)

Set the amount of 'solute' in the parent solution.

Parameters

- **solute** – str String representing the name of the solute of interest

- **amount** – str Quantity String representing the concentration desired, e.g. '1 mol/kg' If the units are given on a per-volume basis, the solution volume is not recalculated and the molar concentrations of other components in the solution are not altered, while the molal concentrations are modified.

  If the units are given on a mass, substance, per-mass, or per-substance basis, then the solution volume is recalculated based on the new composition and the molal concentrations of other components are not altered, while the molar concentrations are modified.

Returns
Nothing. The concentration of solute is modified.

get_total_moles_solute() → Quantity

Return the total moles of all solute in the solution.

get_moles_solvent() → Quantity

Return the moles of solvent present in the solution.

Returns
The moles of solvent in the solution.

get_osmolarity(*activity_correction=False*) → Quantity

Return the osmolarity of the solution in Osm/L.

Parameters
activity_correction – bool If TRUE, the osmotic coefficient is used to calculate the osmolarity. This correction is appropriate when trying to predict the osmolarity that would be measured from e.g. freezing point depression. Defaults to FALSE if omitted.

get_osmolality(*activity_correction=False*) → Quantity

Return the osmolality of the solution in Osm/kg.

Parameters
activity_correction – bool If TRUE, the osmotic coefficient is used to calculate the osmolarity. This correction is appropriate when trying to predict the osmolarity that would be measured from e.g. freezing point depression. Defaults to FALSE if omitted.

get_salt() → Salt

Determine the predominant salt in a solution of ions.

Many empirical equations for solution properties such as activity coefficient, partial molar volume, or viscosity are based on the concentration of single salts (e.g., NaCl). When multiple ions are present (e.g., a solution containing Na+, Cl-, and Mg+2), it is generally not possible to directly model these quantities. pyEQL works around this problem by treating such solutions as single salt solutions.

The get_salt() method examines the ionic composition of a solution and returns an object that identifies the single most predominant salt in the solution, defined by the cation and anion with the highest mole fraction. The Salt object contains information about the stoichiometry of the salt to enable its effective concentration to be calculated (e.g., if a solution contains 0.5 mol/kg of Na+ and Cl-, plus traces of H+ and OH-, the matched salt is 0.5 mol/kg NaCl).

Returns
Salt object containing information about the parent salt.

See also:

get_activity() *get_activity_coefficient()* *get_water_activity()* *get_osmotic_coefficient()* *osmotic_pressure* *viscosity_kinematic*

Examples

```
>>> s1 = Solution([['Na+','0.5 mol/kg'],['Cl-','0.5 mol/kg']])
>>> s1.get_salt()
<pyEQL.salt_ion_match.Salt object at 0x7fe6d3542048>
>>> s1.get_salt().formula
'NaCl'
>>> s1.get_salt().nu_cation
1
>>> s1.get_salt().z_anion
-1
```

```
>>> s2 = pyEQL.Solution([['Na+','0.1 mol/kg'],['Mg+2','0.2 mol/kg'],['Cl-','0.5␣
↪mol/kg']])
>>> s2.get_salt().formula
'MgCl2'
>>> s2.get_salt().nu_anion
2
>>> s2.get_salt().z_cation
2
```

get_salt_dict(*cutoff: float = 0.01*, *use_totals: bool = True*) → dict[str, dict]

Returns a dict of salts that approximates the composition of the Solution. Like *components*, the dict is keyed by formula and the values are the total moles present in the solution, e.g., {"NaCl(aq)": 1}. If the Solution is pure water, the returned dict contains only 'HOH'.

Parameters

- **cutoff** – Lowest salt concentration to consider. Analysis will stop once the concentrations of Salts being analyzed goes below this value. Useful for excluding analysis of trace anions.
- **use_totals** – Whether to base the analysis on total element concentrations or individual species concentrations.

Notes

Salts are identified by pairing the predominant cations and anions in the solution, in descending order of their respective equivalent amounts.

Many empirical equations for solution properties such as activity coefficient, partial molar volume, or viscosity are based on the concentration of single salts (e.g., NaCl). When multiple ions are present (e.g., a solution containing Na+, Cl-, and Mg+2), it is generally not possible to directly model these quantities.

The get_salt_dict() method examines the ionic composition of a solution and simplifies it into a list of salts. The method returns a dictionary of Salt objects where the keys are the salt formulas (e.g., 'NaCl'). The Salt object contains information about the stoichiometry of the salt to enable its effective concentration to be calculated (e.g., 1 M MgCl2 yields 1 M Mg+2 and 2 M Cl-).

Returns

dict
A dictionary of Salt objects, keyed to the salt formula

See also:

osmotic_pressure *viscosity_kinematic* *get_activity()* *get_activity_coefficient()* *get_water_activity()* *get_osmotic_coefficient()*

equilibrate(***kwargs*) → None

Update the composition of the Solution using the thermodynamic engine.

Any kwargs specified are passed through to self.engine.equilibrate()

Returns
Nothing. The .components attribute of the Solution is updated.

get_activity_coefficient(*solute: str*, *scale: Literal['molal', 'molar', 'fugacity', 'rational'] = 'molal'*) → Quantity

Return the activity coefficient of a solute in solution.

The model used to calculate the activity coefficient is determined by the Solution's equation of state engine.

Parameters

- **solute** – The solute for which to retrieve the activity coefficient
- **scale** – The activity coefficient concentration scale
- **verbose** – If True, pyEQL will print a message indicating the parent salt that is being used for activity calculations. This option is useful when modeling multicomponent solutions. False by default.

Returns
the activity coefficient as a dimensionless pint Quantity

Return type
Quantity

get_activity(*solute: str*, *scale: Literal['molal', 'molar', 'rational'] = 'molal'*) → Quantity

Return the thermodynamic activity of the solute in solution on the chosen concentration scale.

Parameters

- **solute** – String representing the name of the solute of interest
- **scale** – The concentration scale for the returned activity. Valid options are "molal", "molar", and "rational" (i.e., mole fraction). By default, the molal scale activity is returned.
- **verbose** – If True, pyEQL will print a message indicating the parent salt that is being used for activity calculations. This option is useful when modeling multicomponent solutions. False by default.

Returns
The thermodynamic activity of the solute in question (dimensionless Quantity)

Notes

The thermodynamic activity depends on the concentration scale used [**?**] . By default, the ionic strength, activity coefficients, and activities are all calculated based on the molal (mol/kg) concentration scale.

References

See also:

ionic_strength get_activity_coefficient() get_salt()

get_osmotic_coefficient(*scale: Literal['molal', 'molar', 'rational'] = 'molal'*) → Quantity

Return the osmotic coefficient of an aqueous solution.

The method used depends on the Solution object's equation of state engine.

get_water_activity() → Quantity

Return the water activity.

Returns
The thermodynamic activity of water in the solution.

Return type
Quantity

See also:

ionic_strength get_activity_coefficient() get_salt()

Notes

Water activity is related to the osmotic coefficient in a solution containing i solutes by:

$$\ln a_w = -\Phi M_w \sum_i m_i$$

Where M_w is the molar mass of water (0.018015 kg/mol) and m_i is the molal concentration of each species.

If appropriate Pitzer model parameters are not available, the water activity is assumed equal to the mole fraction of water.

References

Blandamer, Mike J., Engberts, Jan B. F. N., Gleeson, Peter T., Reis, Joao Carlos R., 2005. "Activity of water in aqueous systems: A frequently neglected property." *Chemical Society Review* 34, 440-458.

Examples

```
>>> s1 = pyEQL.Solution([['Na+','0.3 mol/kg'],['Cl-','0.3 mol/kg']])
>>> s1.get_water_activity()
<Quantity(0.9900944932888518, 'dimensionless')>
```

get_chemical_potential_energy(*activity_correction: bool = True*) → Quantity

Return the total chemical potential energy of a solution (not including pressure or electric effects).

Parameters
activity_correction – bool, optional If True, activities will be used to calculate the true chemical potential. If False, mole fraction will be used, resulting in a calculation of the ideal chemical potential.

Returns

Quantity
The actual or ideal chemical potential energy of the solution, in Joules.

Notes

The chemical potential energy (related to the Gibbs mixing energy) is calculated as follows: [**?**]

$$E = RT \sum_i n_i \ln a_i$$

or

$$E = RT \sum_i n_i \ln x_i$$

Where n is the number of moles of substance, T is the temperature in kelvin, R the ideal gas constant, x the mole fraction, and a the activity of each component.

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

A differential approach.* Elsevier, 2007, pp. 23-37.

_get_property(*solute: str*, *name: str*) → Any | None

Retrieve a thermodynamic property (such as diffusion coefficient) for solute, and adjust it from the reference conditions to the conditions of the solution.

Parameters

- **solute** – str String representing the chemical formula of the solute species
- **name** – str The name of the property needed, e.g. 'diffusion coefficient'

Returns
The desired parameter or None if not found

Return type
Quantity

get_transport_number(*solute: str*) → Quantity

Calculate the transport number of the solute in the solution.

Parameters
solute – Formula of the solute for which the transport number is to be calculated.

Returns
The transport number of *solute*, as a dimensionless Quantity.

Notes

Transport number is calculated according to :

$$t_i = \frac{D_i z_i^2 C_i}{\sum D_i z_i^2 C_i}$$

Where C_i is the concentration in mol/L, D_i is the diffusion coefficient, and z_i is the charge, and the summation extends over all species in the solution.

Diffusion coefficients D_i are adjusted for the effects of temperature and ionic strength using the method implemented in PHREEQC >= 3.4. See *get_diffusion_coefficient for* further details.

References

Geise, G. M.; Cassady, H. J.; Paul, D. R.; Logan, E.; Hickner, M. A. "Specific ion effects on membrane potential and the permselectivity of ion exchange membranes."" *Phys. Chem. Chem. Phys.* 2014, 16, 21673-21681.

See also:

`get_diffusion_coefficient()` `get_molar_conductivity()`

get_lattice_distance(*solute: str*) → Quantity

Calculate the average distance between molecules.

Calculate the average distance between molecules of the given solute, assuming that the molecules are uniformly distributed throughout the solution.

Parameters
solute – str String representing the name of the solute of interest

Returns
The average distance between solute molecules

Return type
Quantity

Examples

```
>>> soln = Solution([['Na+','0.5 mol/kg'],['Cl-','0.5 mol/kg']])
>>> soln.get_lattice_distance('Na+')
1.492964.... nanometer
```

Notes

The lattice distance is related to the molar concentration as follows:

$$d = (C_i N_A)^{-\frac{1}{3}}$$

as_dict() → dict

Convert the Solution into a dict representation that can be serialized to .json or other format.

classmethod from_dict(*d: dict*) → Solution

Instantiate a Solution from a dictionary generated by as_dict().

print(*mode: Literal['all', 'ions', 'cations', 'anions', 'neutrals'] = 'all'*, *units: Literal['ppm', 'mol', 'mol/kg', 'mol/L', '%', 'activity'] = 'mol'*, *places=4*)

Print details about the Solution.

Parameters

- **mode** – Whether to list the amounts of all solutes, or only anions, cations, any ion, or any neutral solute.
- **units** – The units to list solute amounts in. "activity" will list dimensionless activities instead of concentrations.
- **places** – The number of decimal places to round the solute amounts.

to_json() → str

Returns a json string representation of the MSONable object.

unsafe_hash()

Returns an hash of the current object. This uses a generic but low performance method of converting the object to a dictionary, flattening any nested keys, and then performing a hash on the resulting object

classmethod validate_monty_v1(*_MSONable__input_value*)

Pydantic validator with correct signature for pydantic v1.x

classmethod validate_monty_v2(*_MSONable__input_value*, *_*)

Pydantic validator with correct signature for pydantic v2.x

classmethod from_preset(*preset: Literal['seawater', 'rainwater', 'wastewater', 'urine', 'normal saline', 'Ringers lactate']*) → Solution

Instantiate a solution from a preset composition.

Parameters

preset (*str*) – String representing the desired solution. Valid entries are 'seawater', 'rainwater', 'wastewater', 'urine', 'normal saline' and 'Ringers lactate'.

Returns

A pyEQL Solution object.

Raises

FileNotFoundError – If the given preset file doesn't exist on the file system.

Notes

The following sections explain the different solution options:

- '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

to_file(*filename: str | Path*) → None

Saving to a .yaml or .json file.

Parameters
filename (*str | Path*) – The path to the file to save Solution. Valid extensions are .json or .yaml.

classmethod from_file(*filename: str | Path*) → Solution

Loading from a .yaml or .json file.

Parameters
filename (*str | Path*) – Path to the .json or .yaml file (including extension) to load the Solution from. Valid extensions are .json or .yaml.

Returns
A pyEQL Solution object.

Raises
FileNotFoundError – If the given filename doesn't exist on the file system.

4.15 Module reference

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

4.15.1 Salt Matching module

pyEQL salt matching library.

This file contains functions that allow a pyEQL Solution object composed of individual species (usually ions) to be mapped to a solution of one or more salts. This mapping is necessary because some parameters (such as activity coefficient data) can only be determined for salts (e.g. NaCl) and not individual species (e.g. Na+)

copyright
2013-2024 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

class pyEQL.salt_ion_match.**Salt**(*cation*, *anion*)

Class to represent a salt.

get_effective_molality(*ionic_strength*)

Calculate the effective molality according to [**?**].

$$\frac{2I}{(\nu_+ z_+^2 + \nu_- z_-^2)}$$

Parameters
ionic_strength – Quantity The ionic strength of the parent solution, mol/kg

Returns
the effective molality of the salt in the parent solution

Return type
Quantity

References

4.15.2 Modeling Engines module

pyEQL engines for computing aqueous equilibria (e.g., speciation, redox, etc.).

copyright
2013-2024 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

class pyEQL.engines.**EOS**

Abstract base class for pyEQL equation of state classes.

The intent is that concrete implementations of this class make use of the standalone functions available in pyEQL.activity_correction and pyEQL.equilibrium as much as possible. This facilitates robust unit testing while allowing users to "mix and match" or customize the various models as needed.

abstract equilibrate(*solution*)

Adjust the speciation and pH of a Solution object to achieve chemical equilibrium.

The Solution should be modified in-place, likely using add_moles / set_moles, etc.

Parameters
solution – pyEQL Solution object

Returns
Nothing. The speciation of the Solution is modified in-place.

Raises
ValueError if the calculation cannot be completed, e.g. due to insufficient number of parameters or lack of convergence. –

abstract get_activity_coefficient(*solution*, *solute*)

Return the *molal scale* activity coefficient of solute, given a Solution object.

Parameters

- **solution** – pyEQL Solution object
- **solute** – str identifying the solute of interest

Returns
Quantity: dimensionless quantity object

Raises
ValueError if the calculation cannot be completed, e.g. due to insufficient number of parameters. –

abstract get_osmotic_coefficient(*solution*)

Return the *molal scale* osmotic coefficient of a Solution.

Parameters
solution – pyEQL Solution object

Returns
Quantity: dimensionless molal scale osmotic coefficient

Raises
ValueError if the calculation cannot be completed, e.g. due to insufficient number of parameters. –

abstract get_solute_volume()

Return the volume of only the solutes.

Parameters
solution – pyEQL Solution object

Returns
Quantity: solute volume in L

Raises
ValueError if the calculation cannot be completed, e.g. due to insufficient number of parameters. –

class pyEQL.engines.**IdealEOS**

Ideal solution equation of state engine.

equilibrate(*solution*)

Adjust the speciation of a Solution object to achieve chemical equilibrium.

get_activity_coefficient(*solution*, *solute*)

Return the *molal scale* activity coefficient of solute, given a Solution object.

get_osmotic_coefficient(*solution*)

Return the *molal scale* osmotic coefficient of solute, given a Solution object.

get_solute_volume(*solution*)

Return the volume of the solutes.

class pyEQL.engines.**NativeEOS**(*phreeqc_db: Literal['vitens.dat', 'wateq4f_PWN.dat', 'pitzer.dat', 'llnl.dat', 'geothermal.dat'] = 'llnl.dat'*)

pyEQL's native EOS. Uses the Pitzer model when possible, falls back to other models (e.g. Debye-Huckel) based on ionic strength if sufficient parameters are not available.

equilibrate(*solution*)

Adjust the speciation of a Solution object to achieve chemical equilibrium.

get_activity_coefficient(*solution*, *solute*)

Whenever the appropriate parameters are available, the Pitzer model [**?**] is used. If no Pitzer parameters are available, then the appropriate equations are selected according to the following logic: [**?**].

I <= 0.0005: Debye-Huckel equation 0.005 < I <= 0.1: Guntelberg approximation 0.1 < I <= 0.5: Davies equation I > 0.5: Raises a warning and returns activity coefficient = 1

The ionic strength, activity coefficients, and activities are all calculated based on the molal (mol/kg) concentration scale. If a different scale is given as input, then the molal-scale activity coefficient $\gamma_{\pm}$ is converted according to [**?**]

$$f_{\pm} = \gamma_{\pm} * (1 + M_w \sum_i \nu_i m_i)$$

$$y_{\pm} = \frac{m \rho_w}{C \gamma_{\pm}}$$

where $f_{\pm}$ is the rational activity coefficient, M_w is the molecular weight of water, the summation represents the total molality of all solute species, $y_{\pm}$ is the molar activity coefficient, ρ_w is the density of pure water, m and C are the molal and molar concentrations of the chosen salt (not individual solute), respectively.

Parameters

- **solute** – String representing the name of the solute of interest
- **scale** – The concentration scale for the returned activity coefficient. Valid options are "molal", "molar", and "rational" (i.e., mole fraction). By default, the molal scale activity coefficient is returned.

Returns

The mean ion activity coefficient of the solute in question on the selected scale.

Notes

For multicomponent mixtures, pyEQL implements the "effective Pitzer model" presented by Mistry et al. [**?**]. In this model, the activity coefficient of a salt in a multicomponent mixture is calculated using an "effective molality," which is the molality that would result in a single-salt mixture with the same total ionic strength as the multicomponent solution.

$$m_{effective} = \frac{2I}{(\nu_+ z_+^2 + \nu_- - z_-^2)}$$

References

See also:

`pyEQL.solution.Solution.ionic_strength`
`pyEQL.activity_correction.get_activity_coefficient_debyehuckel()`
`pyEQL.activity_correction.get_activity_coefficient_guntelberg()`
`pyEQL.activity_correction.get_activity_coefficient_davies()`
`pyEQL.activity_correction.get_activity_coefficient_pitzer()`

`get_osmotic_coefficient`(*solution*)

Return the *molal scale* osmotic coefficient of solute, given a Solution object.

Osmotic coefficient is calculated using the Pitzer model. [**?**] If appropriate parameters for the model are not available, then pyEQL raises a WARNING and returns an osmotic coefficient of 1.

If the 'rational' scale is given as input, then the molal-scale osmotic coefficient ϕ is converted according to [**?**]

$$g = -\phi M_w \frac{\sum_i \nu_i m_i}{\ln x_w}$$

where g is the rational osmotic coefficient, M_w is the molecular weight of water, the summation represents the total molality of all solute species, and x_w is the mole fraction of water.

Parameters

scale – The concentration scale for the returned osmotic coefficient. Valid options are "molal", "rational" (i.e., mole fraction), and "fugacity". By default, the molal scale osmotic coefficient is returned.

Returns

The osmotic coefficient

Return type
Quantity

See also:

`pyEQL.solution.Solution.get_water_activity()` `pyEQL.solution.Solution.get_salt()` `pyEQL.solution.Solution.ionic_strength`

Notes

For multicomponent mixtures, pyEQL adopts the "effective Pitzer model" presented by Mistry et al. [**?**]. In this approach, the osmotic coefficient of each individual salt is calculated using the normal Pitzer model based on its respective concentration. Then, an effective osmotic coefficient is calculated as the concentration-weighted average of the individual osmotic coefficients.

For example, in a mixture of 0.5 M NaCl and 0.5 M KBr, one would calculate the osmotic coefficient for each salt using a concentration of 0.5 M and an ionic strength of 1 M. Then, one would average the two resulting osmotic coefficients to obtain an effective osmotic coefficient for the mixture.

(Note: in the paper referenced below, the effective osmotic coefficient is determined by weighting using the "effective molality" rather than the true molality. Subsequent checking and correspondence with the author confirmed that the weight factor should be the true molality, and that is what is implemented in pyEQL.)

References

Examples

```
>>> s1 = pyEQL.Solution([['Na+','0.2 mol/kg'],['Cl-','0.2 mol/kg']])
>>> s1.get_osmotic_coefficient()
<Quantity(0.923715281, 'dimensionless')>
```

```
>>> s1 = pyEQL.Solution([['Mg+2','0.3 mol/kg'],['Cl-','0.6 mol/kg']],
↪temperature='30 degC')
>>> s1.get_osmotic_coefficient()
<Quantity(0.891409618, 'dimensionless')>
```

get_solute_volume(*solution*)

Return the volume of the solutes.

class pyEQL.engines.**PhreeqcEOS**(*phreeqc_db: Literal['vitens.dat', 'wateq4f_PWN.dat', 'pitzer.dat', 'llnl.dat', 'geothermal.dat'] = 'phreeqc.dat'*)

Engine based on the PhreeqC model, as implemented via the phreeqpython package.

get_activity_coefficient(*solution*, *solute*)

Return the *molal scale* activity coefficient of solute, given a Solution object.

get_osmotic_coefficient(*solution*)

Return the *molal scale* osmotic coefficient of solute, given a Solution object.

PHREEQC appears to assume a unit osmotic coefficient unless the pitzer database is used. Unfortunately, there is no easy way to access the osmotic coefficient via phreeqcpython

get_solute_volume(*solution*)

Return the volume of the solutes.

4.15.3 Activity Correction functions

pyEQL activity correction library.

This file contains functions for computing molal-scale activity coefficients of ions and salts in aqueous solution.

Individual functions for activity coefficients are defined here so that they can be used independently of a pyEQL solution object. Normally, these functions are called from within the get_activity_coefficient method of the Solution class.

copyright
2013-2024 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

`pyEQL.activity_correction._debye_parameter_B`(*temperature: str = '25 degC'*) → Quantity

Return the constant B used in the extended Debye-Huckel equation.

Parameters
temperature – The temperature of the solution at which to calculate the constant. Defaults to '25 degC'.

Returns
The parameter B for use in extended Debye-Huckel equation (base e). For base 10, divide the resulting value by 2.303. Note that A is often given in base 10 terms in older textbooks and reference material (0.3281 at 25 degC).

Notes

The parameter B is equal to:

$$B = \left(\frac{2N_A\rho_w e^2}{\epsilon_o \epsilon_r kT}\right)^{\frac{1}{2}}$$

References

Bockris and Reddy. /Modern Electrochemistry/, vol 1. Plenum/Rosetta, 1977, p.210.

Archer, Donald G. and Wang, Peiming. "The Dielectric Constant of Water and Debye-Huckel Limiting Law Slopes." /J. Phys. Chem. Ref. Data/ 19(2), 1990.

https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Physical_Chemistry_(LibreTexts)/25%3A_Solutions_II_-_Nonvolatile_Solutes/25.06%3A_The_Debye-Huckel_Theory

https://en.wikipedia.org/wiki/Debye%E2%80%93H%C3%BCckel_equation

`pyEQL.activity_correction._debye_parameter_activity`(*temperature: str = '25 degC'*) → Quantity

Return the constant A for use in the Debye-Huckel limiting law (base e).

Parameters
temperature – The temperature of the solution at which to calculate the constant. Defaults to '25 degC'.

Returns
The parameter A for use in the Debye-Huckel limiting law (base e). For base 10, divide the resulting value by 2.303. Note that A is often given in base 10 terms in older textbooks and reference material (0.509 at 25 degC).

Notes

The parameter A is equal to:

$$A^{\gamma} = \frac{e^3 \left(2\pi N_A \rho\right)^{0.5}}{\left(4\pi\epsilon_o \epsilon_r kT\right)^{1.5}}$$

Note that this equation returns the parameter value that can be used to calculate the natural logarithm of the activity coefficient. For base 10, divide the value returned by 2.303.

References

Archer, Donald G. and Wang, Peiming. "The Dielectric Constant of Water and Debye-Huckel Limiting Law Slopes." /J. Phys. Chem. Ref. Data/ 19(2), 1990.

https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Physical_Chemistry_(LibreTexts)/25%3A_Solutions_II_-_Nonvolatile_Solutes/25.06%3A_The_Debye-Huckel_Theory

https://en.wikipedia.org/wiki/Debye%E2%80%93H%C3%BCckel_equation

See also:

_debye_parameter_osmotic()

pyEQL.activity_correction.**_debye_parameter_osmotic**(*temperature='25 degC'*)

Return the constant A_phi for use in calculating the osmotic coefficient according to Debye-Huckel theory.

Parameters

temperature – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Notes

Not to be confused with the Debye-Huckel constant used for activity coefficients in the limiting law. Takes the value 0.392 at 25 C. This constant is calculated according to: [kim] [arch]

$$A^{\phi} = \frac{1}{3} A^{\gamma}$$

References

J. Chemical Engineering Data 33, pp.177-184.

and Debye-Huckel Limiting Law Slopes." /J. Phys. Chem. Ref. Data/ 19(2), 1990.

Examples

```
>>> _debye_parameter_osmotic()
0.3916...
```

See also:

_debye_parameter_activity()

pyEQL.activity_correction.**_debye_parameter_volume**(*temperature='25 degC'*)

Return the constant A_V, the Debye-Huckel limiting slope for apparent molar volume.

Parameters

temperature – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Notes

Takes the value 1.8305 cm ** 3 * kg ** 0.5 / mol ** 1.5 at 25 C. This constant is calculated according to: [1]_

$$A_V = -2A_\phi RT\left[\frac{3}{\epsilon}\frac{\partial \epsilon}{\partial p} - \frac{1}{\rho}\frac{\partial \rho}{\partial p}\right]$$

Notes: at this time, the term in brackets (containing the partial derivatives) is approximate. These approximations give the correct value of the slope at 25 degC and produce estimates with less than 10% error between 0 and 60 degC.

The derivative of epsilon with respect to pressure is assumed constant (for atmospheric pressure) at -0.01275 1/MPa. Note that the negative sign does not make sense in light of real data, but is required to give the correct result.

The second term is equivalent to the inverse of the bulk modulus of water, which is taken to be 2.2 GPa. [2]_

References

See also:

_debye_parameter_osmotic()

pyEQL.activity_correction.**_pitzer_B_MX**(*ionic_strength*, *alpha1*, *alpha2*, *beta0*, *beta1*, *beta2*)

Return the B_MX coefficient for the Pitzer ion interaction model.

$$B_M X = \beta_0 + \beta_1 f1(\alpha_1 I^{0.5}) + \beta_2 f2(\alpha_2 I^{0.5})$$

Parameters

- **ionic_strength** – The ionic strength of the parent solution, mol/kg
- **alpha1** – Coefficients for the Pitzer model, kg ** 0.5 / mol ** 0.5
- **alpha2** – Coefficients for the Pitzer model, kg ** 0.5 / mol ** 0.5
- **beta0** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta1** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta2** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.

Returns

The B_MX parameter for the Pitzer ion interaction model.

References

Scharge, T., Munoz, A.G., and Moog, H.C. (2012). Activity Coefficients of Fission Products in Highly Salinary Solutions of Na+, K+, Mg2+, Ca2+, Cl-, and SO42- : Cs+. /Journal of Chemical& Engineering Data (57), p. 1637-1647.

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

See also:

_pitzer_f1()

pyEQL.activity_correction.**_pitzer_B_phi**(*ionic_strength*, *alpha1*, *alpha2*, *beta0*, *beta1*, *beta2*)

Returns the B^Phi coefficient for the Pitzer ion interaction model.

This function calculates the B^Phi coefficient using the formula:

$$B^{\Phi} = \beta_0 + \beta 1 \exp(-\alpha_1 I^{0.5}) + \beta_2 \exp(-\alpha_2 I^{0.5})$$

or

$$B^{\Phi} = B^{\gamma} - B_{MX}$$

Parameters

- **ionic_strength** – The ionic strength of the parent solution, mol/kg.
- **alpha1** – Coefficients for the Pitzer model, kg ** 0.5 / mol ** 0.5.
- **alpha2** – Coefficients for the Pitzer model, kg ** 0.5 / mol ** 0.5.
- **beta0** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta1** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta2** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.

Returns

The B^Phi parameter for the Pitzer ion interaction model.

Return type

float

References

Scharge, T., Munoz, A.G., and Moog, H.C. (2012). Activity Coefficients of Fission Products in Highly Salinary Solutions of Na+, K+, Mg2+, Ca2+, Cl-, and SO42- : Cs+. /Journal of Chemical& Engineering Data (57), p. 1637-1647.

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

Beyer, R., & Steiger, M. (2010). Vapor Pressure Measurements of NaHCOO + H 2 O and KHCOO + H 2 O from 278 to 308 K and Representation with an Ion Interaction (Pitzer) Model. Journal of Chemical & Engineering Data, 55(2), 830-838. doi:10.1021/je900487a

pyEQL.activity_correction.**_pitzer_f1**(x)

The function of ionic strength used to calculate beta_MX in the Pitzer ion interaction model.

$$f(x) = 2[1 - (1 + x)\exp(-x)]/x^2$$

References

Scharge, T., Munoz, A.G., and Moog, H.C. (2012). Activity Coefficients of Fission Products in Highly Salinary Solutions of Na+, K+, Mg2+, Ca2+, Cl-, and SO42- : Cs+. /Journal of Chemical& Engineering Data (57), p. 1637-1647.

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

pyEQL.activity_correction.**_pitzer_f2**(x)

The function of ionic strength used to calculate beta_gamma in the Pitzer ion interaction model.

$$f(x) = -\frac{2}{x^2}[1 - (\frac{1 + x + x^2}{2})\exp(-x)]$$

References

Scharge, T., Munoz, A.G., and Moog, H.C. (2012). Activity Coefficients of Fission Products in Highly Salinary Solutions of Na+, K+, Mg2+, Ca2+, Cl-, and SO42- : Cs+. /Journal of Chemical& Engineering Data (57), p. 1637-1647.

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

pyEQL.activity_correction.**_pitzer_log_gamma**(*ionic_strength*, *molality*, *B_MX*, *B_phi*, *C_phi*, *z_cation*, *z_anion*, *nu_cation*, *nu_anion*, *temperature='25 degC'*, *b=<Quantity(1.2, 'kilogram ** 0.5 / mole ** 0.5')>*)

Returns the natural logarithm of the binary activity coefficient calculated by the Pitzer ion interaction model.

$$\ln \gamma_{MX} = -|z_+ z_-|A^{Phi}(\frac{I^{0.5}}{(1 + bI^{0.5})} + \frac{2}{b}\ln(1 + bI^{0.5})) + \frac{m(2\nu_+\nu_-)}{(\nu_+ + \nu_-)}(B_{MX} + B^{\Phi}_{MX}) + \frac{m^2(3(\nu_+\nu_-)^{1.5}}{(\nu_+ + \nu_-))}C^{\Phi}_{MX}$$

Parameters

- **ionic_strength** (*Quantity*) – The ionic strength of the parent solution, mol/kg.
- **molality** (*Quantity*) – The concentration of the salt, mol/kg.
- **B_MX** (*Quantity*) – Calculated parameters for the Pitzer ion interaction model.
- **B_phi** (*Quantity*) – Calculated parameters for the Pitzer ion interaction model.
- **C_phi** (*Quantity*) – Calculated parameters for the Pitzer ion interaction model.
- **z_cation** (*int*) – The formal charge on the cation and anion, respectively.
- **z_anion** (*int*) – The formal charge on the cation and anion, respectively.
- **nu_cation** (*int*) – The stoichiometric coefficient of the cation and anion in the salt.
- **nu_anion** (*int*) – The stoichiometric coefficient of the cation and anion in the salt.
- **temperature** (*str, Quantity*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

- **b** (*number, optional*) – Coefficient. Usually set equal to 1.2 kg ** 0.5 / mol ** 0.5 and considered independent of temperature and pressure.

Returns
The natural logarithm of the binary activity coefficient calculated by the Pitzer ion interaction model.

Return type
float

References

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

May, P. M., Rowland, D., Hefter, G., & Königsberger, E. (2011). A Generic and Updatable Pitzer Characterization of Aqueous Binary Electrolyte Solutions at 1 bar and 25 °C. Journal of Chemical & Engineering Data, 56(12), 5066-5077. doi:10.1021/je2009329

pyEQL.activity_correction.**get_activity_coefficient_davies**(*ionic_strength*, *z=1*, *temperature='25 degC'*)

Return the activity coefficient of solute in the parent solution according to the Davies equation.

Parameters

- **ionic_strength** (*Quantity*) – The ionic strength of the parent solution, mol/kg.
- **z** (*int, optional*) – The charge on the solute, including sign. Defaults to +1 if not specified.
- **temperature** (*str, Quantity, optional*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Returns
The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless.

Return type
Quantity

Notes

Activity coefficient is calculated according to:

$$\ln \gamma = A^{\gamma} z_i^2 (\frac{\sqrt{I}}{(1+\sqrt{I})} + 0.2I)$$

Valid for 0.1 < I < 0.5

See also:

_debye_parameter_activity() get_activity_coefficient_debyehuckel()
get_activity_coefficient_guntelberg()

References

Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed,
pp 103. Wiley Interscience, 1996.

pyEQL.activity_correction.**get_activity_coefficient_debyehuckel**(*ionic_strength*, *z=1*, *temperature='25 degC'*)

Return the activity coefficient of solute in the parent solution according to the Debye-Huckel limiting law.

Parameters

- **z** (*int, optional*) – The charge on the solute, including sign. Defaults to +1 if not specified.
- **ionic_strength** (*Quantity*) – The ionic strength of the parent solution, mol/kg.
- **temperature** (*str, Quantity, optional*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Returns
The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless.

Return type
Quantity

Notes

Activity coefficient is calculated according to:

$$\ln \gamma = A^{\gamma} z_i^2 \sqrt{I}$$

Valid only for I < 0.005

See also:

_debye_parameter_activity() get_activity_coefficient_davies()
get_activity_coefficient_guntelberg()

References

Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed,
pp 103. Wiley Interscience, 1996.

pyEQL.activity_correction.**get_activity_coefficient_guntelberg**(*ionic_strength*, *z=1*, *temperature='25 degC'*)

Return the activity coefficient of solute in the parent solution according to the Guntelberg approximation.

Parameters

- **z** (*int, optional*) – The charge on the solute, including sign. Defaults to +1 if not specified.
- **ionic_strength** (*Quantity*) – The ionic strength of the parent solution, mol/kg.
- **temperature** (*str, Quantity, optional*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Returns
The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless.

Return type
Quantity

Notes

Activity coefficient is calculated according to:

$$\ln \gamma = A^{\gamma} z_i^2 \frac{\sqrt{I}}{(1+\sqrt{I})}$$

Valid for I < 0.1

See also:

_debye_parameter_activity() get_activity_coefficient_davies()
get_activity_coefficient_debyehuckel()

References

Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed,
pp 103. Wiley Interscience, 1996.

pyEQL.activity_correction.**get_activity_coefficient_pitzer**(*ionic_strength*, *molality*, *alpha1*, *alpha2*, *beta0*, *beta1*, *beta2*, *C_phi*, *z_cation*, *z_anion*, *nu_cation*, *nu_anion*, *temperature='25 degC'*, *b=1.2*)

Return the activity coefficient of solute in the parent solution according to the Pitzer model.

Parameters

- **ionic_strength** – The ionic strength of the parent solution, mol/kg
- **molality** – The molal concentration of the parent salt, mol/kg
- **alpha1** – Coefficients for the Pitzer model. This function assigns the coefficients proper units of kg ** 0.5 / mol ** 0.5 after they are entered.
- **alpha2** – Coefficients for the Pitzer model. This function assigns the coefficients proper units of kg ** 0.5 / mol ** 0.5 after they are entered.
- **beta0** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta1** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta2** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **C_phi** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **z_cation** – The charge on the cation and anion, respectively
- **z_anion** – The charge on the cation and anion, respectively
- **nu_cation** – The stoichiometric coefficient of the cation and anion in the salt
- **nu_anion** – The stoichiometric coefficient of the cation and anion in the salt

- **temperature** – String representing the temperature of the solution. Defaults to '25 degC' if not specified.
- **b** – Coefficient. Usually set equal to 1.2 and considered independent of temperature and pressure. If provided, this coefficient is assigned proper units of kg ** 0.5 / mol ** 0.5 after entry.

Returns

Quantity
The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless

Examples

```
>>> get_activity_coefficient_pitzer(0.5*ureg.Quantity('mol/kg'),0.5*ureg.Quantity(
↪'mol/kg'),1,0.5,-.0181191983,-.4625822071,.4682,.000246063,1,-1,1,1,b=1.2)
0.61915...
```

```
>>> get_activity_coefficient_pitzer(5.6153*ureg.Quantity('mol/kg'),5.6153*ureg.
↪Quantity('mol/kg'),3,0.5,0.0369993,0.354664,0.0997513,-0.00171868,1,-1,1,1,b=1.2)
0.76331...
```

Notes: the examples below are for comparison with experimental and modeling data presented in the May et al reference below.

10 mol/kg ammonium nitrate. Estimated result (from graph) = 0.2725

```
>>> get_activity_coefficient_pitzer(10*ureg.Quantity('mol/kg'),10*ureg.Quantity(
↪'mol/kg'),2,0,-0.01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.22595 ...
```

5 mol/kg ammonium nitrate. Estimated result (from graph) = 0.3011

```
>>> get_activity_coefficient_pitzer(5*ureg.Quantity('mol/kg'),5*ureg.Quantity('mol/
↪kg'),2,0,-0.01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.30249 ...
```

18 mol/kg ammonium nitrate. Estimated result (from graph) = 0.1653

```
>>> get_activity_coefficient_pitzer(18*ureg.Quantity('mol/kg'),18*ureg.Quantity(
↪'mol/kg'),2,0,-0.01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.16241 ...
```

References

Scharge, T., Munoz, A.G., and Moog, H.C. (2012). Activity Coefficients of Fission Products in Highly Salinary Solutions of Na+, K+, Mg2+, Ca2+, Cl-, and SO42- : Cs+. /Journal of Chemical& Engineering Data (57), p. 1637-1647.

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

May, P. M., Rowland, D., Hefter, G., & Königsberger, E. (2011). A Generic and Updatable Pitzer Characterization of Aqueous Binary Electrolyte Solutions at 1 bar and 25 °C. Journal of Chemical & Engineering Data, 56(12), 5066-5077. doi:10.1021/je2009329

Beyer, R., & Steiger, M. (2010). Vapor Pressure Measurements of NaHCOO + H 2 O and KHCOO + H 2 O from 278 to 308 K and Representation with an Ion Interaction (Pitzer) Model. Journal of Chemical & Engineering Data, 55(2), 830-838. doi:10.1021/je900487a

See also:

_debye_parameter_activity() _pitzer_B_MX() _pitzer_B_phi() _pitzer_log_gamma()

pyEQL.activity_correction.**get_apparent_volume_pitzer**(*ionic_strength*, *molality*, *alpha1*, *alpha2*, *beta0*, *beta1*, *beta2*, *C_phi*, *V_o*, *z_cation*, *z_anion*, *nu_cation*, *nu_anion*, *temperature='25 degC'*, *b=1.2*)

Return the apparent molar volume of solute in the parent solution according to the Pitzer model.

Parameters

- **ionic_strength** (*Quantity*) – The ionic strength of the parent solution, mol/kg.
- **molality** (*Quantity*) – The molal concentration of the parent salt, mol/kg.
- **alpha1** (*number*) – Coefficients for the Pitzer model. This function assigns the coefficients proper units of kg ** 0.5 / mol ** 0.5 after they are entered.
- **alpha2** (*number*) – Coefficients for the Pitzer model. This function assigns the coefficients proper units of kg ** 0.5 / mol ** 0.5 after they are entered.
- **beta0** (*number*) – Pitzer coefficients for the apparent molar volume. These ion-interaction parameters are specific to each salt system.
- **beta1** (*number*) – Pitzer coefficients for the apparent molar volume. These ion-interaction parameters are specific to each salt system.
- **beta2** (*number*) – Pitzer coefficients for the apparent molar volume. These ion-interaction parameters are specific to each salt system.
- **C_phi** (*number*) – Pitzer coefficients for the apparent molar volume. These ion-interaction parameters are specific to each salt system.
- **V_o** (*number*) – The V^o Pitzer coefficient for the apparent molar volume.
- **z_cation** (*int*) – The formal charge on the cation and anion, respectively.
- **z_anion** (*int*) – The formal charge on the cation and anion, respectively.
- **nu_cation** (*int*) – The stoichiometric coefficient of the cation and anion in the salt.
- **nu_anion** (*int*) – The stoichiometric coefficient of the cation and anion in the salt.
- **temperature** (*str, Quantity*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.
- **b** (*number, optional*) – Coefficient. Usually set equal to 1.2 and considered independent of temperature and pressure. If provided, this coefficient is assigned proper units of kg ** 0.5 / mol ** 0.5 after entry.

Returns

The apparent molar volume of the solute, cm ** 3 / mol.

Return type

Quantity

Examples

Notes: the example below is for comparison with experimental and modeling data presented in the Krumgalz et al reference below.

0.25 mol/kg CuSO4. Expected result (from graph) = 0.5 cm ** 3 / mol

```
>>> get_apparent_volume_pitzer(1.0*ureg.Quantity('mol/kg'),0.25*ureg.Quantity('mol/
↪kg'),1.4,12,0.001499,-0.008124,0.2203,-0.0002589,-6,2,-2,1,1,b=1.2)
0.404...
```

1.0 mol/kg CuSO4. Expected result (from graph) = 4 cm ** 3 / mol

```
>>> get_apparent_volume_pitzer(4.0*ureg.Quantity('mol/kg'),1.0*ureg.Quantity('mol/kg
↪'),1.4,12,0.001499,-0.008124,0.2203,-0.0002589,-6,2,-2,1,1,b=1.2)
4.424...
```

10.0 mol/kg ammonium nitrate. Expected result (from graph) = 50.3 cm ** 3 / mol

```
>>> get_apparent_volume_pitzer(10.0*ureg.Quantity('mol/kg'),10.0*ureg.Quantity('mol/
↪kg'),2,0,0.000001742,0.0002926,0,0.000000424,46.9,1,-1,1,1,b=1.2)
50.286...
```

20.0 mol/kg ammonium nitrate. Expected result (from graph) = 51.2 cm ** 3 / mol

```
>>> get_apparent_volume_pitzer(20.0*ureg.Quantity('mol/kg'),20.0*ureg.Quantity('mol/
↪kg'),2,0,0.000001742,0.0002926,0,0.000000424,46.9,1,-1,1,1,b=1.2)
51.145...
```

Notes: the examples below are for comparison with experimental and modeling data presented in the Krumgalz et al reference below.

0.8 mol/kg NaF. Expected result = 0.03

```
>>> get_apparent_volume_pitzer(0.8*ureg.Quantity('mol/kg'),0.8*ureg.Quantity('mol/kg
↪'),2,0,0.000024693,0.00003169,0,-0.000004068,-2.426,1,-1,1,1,b=1.2)
0.22595 ...
```

References

May, P. M., Rowland, D., Hefter, G., & Königsberger, E. (2011). A Generic and Updatable Pitzer Characterization of Aqueous Binary Electrolyte Solutions at 1 bar and 25 °C. Journal of Chemical & Engineering Data, 56(12), 5066-5077. doi:10.1021/je2009329

Krumgalz, Boris S., Pogorelsky, Rita (1996). Volumetric Properties of Single Aqueous Electrolytes from Zero to Saturation Concentration at 298.15 K Represented by Pitzer's Ion-Interaction Equations. Journal of Physical Chemical Reference Data, 25(2), 663-689.

See also:

_debye_parameter_volume() *_pitzer_B_MX()*

pyEQL.activity_correction.**get_osmotic_coefficient_pitzer**(*ionic_strength*, *molality*, *alpha1*, *alpha2*, *beta0*, *beta1*, *beta2*, *C_phi*, *z_cation*, *z_anion*, *nu_cation*, *nu_anion*, *temperature='25 degC'*, *b=1.2*)

Return the osmotic coefficient of water in an electrolyte solution according to the Pitzer model.

Parameters

- **ionic_strength** (*Quantity*) – The ionic strength of the parent solution, mol/kg.
- **molality** (*Quantity*) – The molal concentration of the parent salt, mol/kg.
- **alpha1** (*number*) – Coefficients for the Pitzer model. This function assigns the coefficients proper units of kg ** 0.5 / mol ** 0.5 after they are entered.
- **alpha2** (*number*) – Coefficients for the Pitzer model. This function assigns the coefficients proper units of kg ** 0.5 / mol ** 0.5 after they are entered.
- **beta0** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta1** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta2** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **C_phi** – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **z_cation** (*int*) – The formal charge on the cation and anion, respectively.
- **z_anion** (*int*) – The formal charge on the cation and anion, respectively.
- **nu_cation** (*int*) – The stoichiometric coefficient of the cation and anion in the salt.
- **nu_anion** (*int*) – The stoichiometric coefficient of the cation and anion in the salt.
- **temperature** (*str, Quantity*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.
- **b** (*number, optional*) – Coefficient. Usually set equal to 1.2 and considered independent of temperature and pressure. If provided, this coefficient is assigned proper units of kg ** 0.5 / mol ** 0.5 after entry.

Returns

The osmotic coefficient of water, dimensionless.

Return type

Quantity

Examples

Experimental value according to Beyer and Stieger reference is 1.3550

```
>>> get_osmotic_coefficient_pitzer(10.175*ureg.Quantity('mol/kg'),10.175*ureg.
↪Quantity('mol/kg'),1,0.5,-.0181191983,-.4625822071,.4682,.000246063,1,-1,1,1,b=1.
↪2)
1.3552 ...
```

Experimental value according to Beyer and Stieger reference is 1.084

```
>>> get_osmotic_coefficient_pitzer(5.6153*ureg.Quantity('mol/kg'),5.6153*ureg.
↪Quantity('mol/kg'),3,0.5,0.0369993,0.354664,0.0997513,-0.00171868,1,-1,1,1,b=1.2)
1.0850 ...
```

Notes: the examples below are for comparison with experimental and modeling data presented in the May et al reference below.

10 mol/kg ammonium nitrate. Estimated result (from graph) = 0.62

```
>>> get_osmotic_coefficient_pitzer(10*ureg.Quantity('mol/kg'),10*ureg.Quantity('mol/
↪kg'),2,0,-0.01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.6143 ...
```

5 mol/kg ammonium nitrate. Estimated result (from graph) = 0.7

```
>>> get_osmotic_coefficient_pitzer(5*ureg.Quantity('mol/kg'),5*ureg.Quantity('mol/kg
↪'),2,0,-0.01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.6925 ...
```

18 mol/kg ammonium nitrate. Estimated result (from graph) = 0.555

```
>>> get_osmotic_coefficient_pitzer(18*ureg.Quantity('mol/kg'),18*ureg.Quantity('mol/
↪kg'),2,0,-0.01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.5556 ...
```

References

Scharge, T., Munoz, A.G., and Moog, H.C. (2012). Activity Coefficients of Fission Products in Highly Salinary Solutions of Na+, K+, Mg2+, Ca2+, Cl-, and SO42- : Cs+. /Journal of Chemical& Engineering Data (57), p. 1637-1647.

Kim, H., & Jr, W. F. (1988). Evaluation of Pitzer ion interaction parameters of aqueous electrolytes at 25 degree C. 1. Single salt parameters. Journal of Chemical and Engineering Data, (2), 177-184.

May, P. M., Rowland, D., Hefter, G., & Königsberger, E. (2011). A Generic and Updatable Pitzer Characterization of Aqueous Binary Electrolyte Solutions at 1 bar and 25 °C. Journal of Chemical & Engineering Data, 56(12), 5066-5077. doi:10.1021/je2009329

Beyer, R., & Steiger, M. (2010). Vapor Pressure Measurements of NaHCOO + H 2 O and KHCOO + H 2 O from 278 to 308 K and Representation with an Ion Interaction (Pitzer) Model. Journal of Chemical & Engineering Data, 55(2), 830-838. doi:10.1021/je900487a

See also:

_debye_parameter_activity() _pitzer_B_MX() _pitzer_B_phi() _pitzer_log_gamma()

4.15.4 Speciation functions

pyEQL methods for chemical equilibrium calculations (e.g. acid/base, reactions, redox, complexation, etc.).

NOTE: these methods are not currently used but are here for the future.

pyEQL.equilibrium.**adjust_temp_arrhenius**(*rate_constant*, *activation_energy*, *temperature*, *reference_temperature=<Quantity(25, 'degree_Celsius')>*)

Adjust a reaction equilibrium constant from one temperature to another.

Parameters

- **rate_constant** (*Quantity*) – The parameter value (usually a rate constant) being adjusted.
- **activation_energy** (*Quantity*) – The activation energy of the process, in kJ/mol.
- **temperature** (*Quantity*) – The desired reaction temperature.
- **reference_temperature** (*Quantity, optional*) – The temperature at which equilibrium_constant is valid. Defaults to 25 degrees C if omitted.

Returns
The adjusted reaction equilibrium constant.

Return type
Quantity

Notes

This function implements the Arrhenius equation to adjust measured rate constants to other temperatures. TODO - add better reference

$$ln(\frac{K2}{K1} = \frac{E_a}{R}(\frac{1}{T_1} - \frac{1}{T_2})$$

References

http://chemwiki.ucdavis.edu/Physical_Chemistry/Kinetics/Reaction_Rates/Temperature_Dependence_of_Reaction_Rates/Arrhenius_Equation

Examples

```
>>> adjust_temp_arrhenius(7,900*ureg.Quantity('kJ/mol'),37*ureg.Quantity('degC'),
↪97*ureg.Quantity('degC'))
1.8867225...e-24
```

pyEQL.equilibrium.**adjust_temp_pitzer**(*c1*, *c2*, *c3*, *c4*, *c5*, *temp*, *temp_ref=<Quantity(298.15, 'kelvin')>*)

Calculate a parameter for the Pitzer model based on temperature-dependent coefficients c1,c2,c3,c4,and c5.

Parameters

- **c1** (*float*) – Temperature-dependent coefficients for the pitzer parameter of interest.
- **c2** (*float*) – Temperature-dependent coefficients for the pitzer parameter of interest.
- **c3** (*float*) – Temperature-dependent coefficients for the pitzer parameter of interest.
- **c4** (*float*) – Temperature-dependent coefficients for the pitzer parameter of interest.
- **c5** (*float*) – Temperature-dependent coefficients for the pitzer parameter of interest.
- **temp** (*Quantity*) – The temperature at which the Pitzer parameter is to be calculated.
- **temp_ref** (*Quantity, optional*) – The reference temperature on which the parameters are based. Defaults to 298.15 K if omitted.

Note: As described in the PHREEQC documentation.

pyEQL.equilibrium.**adjust_temp_vanthoff**(*equilibrium_constant*, *enthalpy*, *temperature*, *reference_temperature=<Quantity(25, 'degree_Celsius')>*)

Adjust a reaction equilibrium constant from one temperature to another.

Parameters

- **equilibrium_constant** (*float*) – The reaction equilibrium constant for the reaction.
- **enthalpy** (*Quantity*) – The enthalpy change (delta H) for the reaction in kJ/mol. Assumed independent of temperature (see Notes).
- **temperature** (*Quantity*) – The desired reaction temperature in degrees Celsius.
- **reference_temperature** (*Quantity, optional*) – The temperature at which equilibrium_constant is valid. Defaults to 25 degrees C if omitted.

Returns

The adjusted reaction equilibrium constant.

Return type

float

Note: This function implements the Van't Hoff equation to adjust measured equilibrium constants to other temperatures.

$$ln(K2/K1) = \frac{\delta H}{R}(\frac{1}{T_1} - \frac{1}{T_2})$$

This implementation assumes that the enthalpy is independent of temperature over the range of interest.

References

Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed, pp 53. Wiley Interscience, 1996.

Examples

```
>>> adjust_temp_vanthoff(0.15,ureg.Quantity('-197.6 kJ/mol'),ureg.Quantity('42 degC
↪'),ureg.Quantity(' 25degC'))
0.00203566...
```

If the 'ref_temperature' parameter is omitted, a default of 25 C is used.

```
>>> adjust_temp_vanthoff(0.15,ureg.Quantity('-197.6 kJ/mol'),ureg.Quantity('42 degC
↪'))
0.00203566...
```

pyEQL.equilibrium.**alpha**(*n*, *pH*, *pKa_list*)

Returns the acid-base distribution coefficient (alpha) of an acid in the n-deprotonated form at a given pH.

Parameters

- **n** (*int*) – The number of protons that have been lost by the desired form of the acid. Also the subscript on the alpha value. E.g. for bicarbonate (HCO3-), n=1 because 1 proton has been lost from the fully-protonated carbonic acid (H2CO3) form.
- **pH** (*float or int*) – The pH of the solution.

- **pKa_list** (*list of floats or ints*) – The pKa values (negative log of equilibrium constants) for the acid of interest. There must be a minimum of n pKa values in the list.

Returns
The fraction of total acid present in the specified form.

Return type
float

Notes

The acid-base coefficient is calculated as follows: [**?**]

$$\alpha_n = \frac{term_n}{[H+]^n + k_{a1}[H+]^{n-1} + k_{a1}k_{a2}[H+]^{n-2}...k_{a1}k_{a2}...k_{an}}$$

Where :math: 'term_n' refers to the nth term in the denominator, starting from 0

References

Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed, pp 127-130. Wiley Interscience, 1996.

Examples

```
>>> alpha(1,8,[4.7])
0.999...
```

The sum of all alpha values should equal 1

```
>>> alpha(0,8,[6.35,10.33])
0.021...
>>> alpha(1,8,[6.35,10.33])
0.979...
>>> alpha(2,8,[6.35,10.33])
2.043...e-09
```

If pH is equal to one of the pKa values the function should return 0.5.

```
>>> alpha(1,6.35,[6.35,10.33])
0.5
```

4.15.5 Utilities

pyEQL utilities

copyright
2013-2024 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

class pyEQL.utils.**FormulaDict**(*dict=None*, /, ***kwargs*)

Automatically converts keys on get/set using pymatgen.core.Ion.from_formula(key).reduced_formula.

This allows getting/setting/updating of Solution.components using flexible formula notation (e.g., "Na+", "Na+1", "Na[+]" all have the same effect)

pyEQL.utils.**create_water_substance**(*temperature: float*, *pressure: float*)

Instantiate a water substance model from IAPWS.

Parameters

- **temperature** – the desired temperature in K
- **pressure** – the desired pressure in MPa

Notes

The IAPWS97 model is much faster than IAPWS95, but the latter can do temp below zero. See https://github.com/jjgomera/iapws/issues/14. Hence, IAPWS97 will be used except when *temperature* is less than 0 degC.

Returns
A IAPWS97 or IAPWS95 instance

pyEQL.utils.**format_solutes_dict**(*solute_dict: dict*, *units: str*)

Formats a dictionary of solutes by converting the amount to a string with the provided units suitable for passing to use with the Solution class. Note that all solutes must be given in the same units.

Parameters

- **solute_dict** – The dictionary to format. This must be of the form dict{str: Number} e.g. {"Na+": 0.5, "Cl-": 0.9}
- **units** – The units to use for the solute. e.g. "mol/kg"

Returns
A formatted solute dictionary.

Raises
TypeError if solute_dict is not a dictionary. –

pyEQL.utils.**interpret_units**(*unit: str*) → str

Translate commonly used environmental units such as 'ppm' into strings that *pint* can understand.

Parameters
unit – string representing the unit to translate

Returns: a unit that pint can understand

pyEQL.utils.**standardize_formula**(*formula: str*)

Convert a chemical formula into standard form.

Parameters
formula – the chemical formula to standardize.

Returns
A standardized chemical formula

Raises
ValueError if formula cannot be processed or is invalid. –

Notes

Currently this method standardizes formulae by passing them through *pymatgen.core.ion.Ion.reduced_formula()*. For ions, this means that 1) the charge number will always be listed explicitly and 2) the charge number will be enclosed in square brackets to remove any ambiguity in the meaning of the formula. For example, 'Na+', 'Na+1', and 'Na[+]' will all standardize to "Na[+1]"

4.16 Contributing to pyEQL

4.16.1 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.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered solved.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

4.16.2 Documentation Improvements

You can help improve `pyEQL` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`pyEQL` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

Tip: Please notice that the [GitHub web interface] provides a quick way of propose changes in `pyEQL`'s files. While this mechanism can be tricky for normal code contributions, it works perfectly fine for contributing to the docs, and can be quite handy.

If you are interested in trying this method out, please navigate to the `docs` folder in the source [repository], find which file you would like to propose changes and click in the little pencil icon at the top, to open [GitHub's code editor]. Once you finish editing the file, please write a message in the form at the bottom of the page describing which changes have you made and what are the motivations behind them and submit your proposal.

When working on documentation changes in your local machine, you can compile them using [tox] :

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

4.16.3 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.

The Scientific Python Guide is also an excellent technical reference for new and longtime developers.

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

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.

4.16.4 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!

4.16.5 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).

4.16.6 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`.

4.17 Contributors

pyEQL was originally written by Prof. Ryan Kingsbury (@rkingsbury) and is primarily developed and maintained by the Kingsbury Lab at Princeton University.

Other contributors, listed alphabetically, are:

- Kirill Pushkarev (@kirill-push)
- Dhruv Duseja (@DhruvDuseja)
- Andrew Rosen (@arosen93)
- Hernan Grecco (@hgrecco)

(If you think that your name belongs here, please let the maintainer know)

4.18 License

BIBLIOGRAPHY

[aq] https://www.aqion.de/site/electrical-conductivity

[hc] http://www.hydrochemistry.eu/exmpls/sc.html

[stm] Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed, pp 165. Wiley Interscience, 1996.

[wk3] https://en.wikipedia.org/wiki/Debye_length#Debye_length_in_an_electrolyte

[sata] Sata, Toshikatsu. Ion Exchange Membranes: Preparation, Characterization, and Modification. Royal Society of Chemistry, 2004, p. 10.

[wk] http://en.wikipedia.org/wiki/Osmotic_pressure#Derivation_of_osmotic_pressure

[rs] Robinson, R. A.; Stokes, R. H. Electrolyte Solutions: Second Revised Edition; Butterworths: London, 1968, p.32.

[koga] Koga, Yoshikata, 2007. *Solution Thermodynamics and its Application to Aqueous Solutions:

[mistry] Mistry, K. H.; Hunter, H. a.; Lienhard V, J. H. Effect of composition and nonideal solution behavior on desalination calculations for mixed electrolyte solutions with comparison to seawater. Desalination 2013, 318, 34-47.

[may] **May, P. M., Rowland, D., Hefter, G., & Königsberger, E. (2011).**
A Generic and Updatable Pitzer Characterization of Aqueous Binary Electrolyte Solutions at 1 bar and 25 °C.

Journal of Chemical & Engineering Data, 56(12), 5066-5077. doi:10.1021/je2009329

[stumm] Stumm, Werner and Morgan, James J. *Aquatic Chemistry*, 3rd ed, pp 165. Wiley Interscience, 1996.

[rbs] Robinson, R. A.; Stokes, R. H. Electrolyte Solutions: Second Revised Edition; Butterworths: London, 1968, p.32.

[mistry] Mistry, K. H.; Hunter, H. a.; Lienhard V, J. H. Effect of composition and nonideal solution behavior on desalination calculations for mixed electrolyte solutions with comparison to seawater. Desalination 2013, 318, 34-47.

[may] May, P. M., Rowland, D., Hefter, G., & Königsberger, E. (2011). A Generic and Updatable Pitzer Characterization of Aqueous Binary Electrolyte Solutions at 1 bar and 25 °C. Journal of Chemical & Engineering Data, 56(12), 5066-5077. doi:10.1021/je2009329

[rbs] Robinson, R. A.; Stokes, R. H. Electrolyte Solutions: Second Revised Edition; Butterworths: London, 1968, p.32.

[mstry] Mistry, K. H.; Hunter, H. a.; Lienhard V, J. H. Effect of composition and nonideal solution behavior on desalination calculations for mixed electrolyte solutions with comparison to seawater. Desalination 2013, 318, 34-47.

[kim] Kim, Hee-Talk and Frederick, William Jr, 1988. “Evaluation of Pitzer Ion Interaction Parameters of Aqueous Electrolytes at 25 C. 1. Single Salt Parameters,”

[arch] Archer, Donald G. and Wang, Peiming. “The Dielectric Constant of Water

INDEX

Symbols

A

B

C

D

E

F

G

H

I

M

N

O

P

S

T

U

V

W