pyEQL Documentation

Release v0.6.0

Ryan Kingsbury

Aug 16, 2023

CONTENTS

1 Description **3**
1.1 Key Features 3

Bibliography **63**

Python Module Index **65**

Index **67**

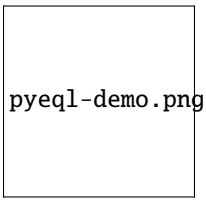

pyEQL

CHAPTER

ONE

DESCRIPTION

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



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

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

1.1 Key Features

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

Contents:

1.1.1 Installation

Use a conda environment

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

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

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

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

pip install

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

```
pip install pyEQL
```

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

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

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

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

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

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

Other dependencies

pyEQL also requires the following packages:

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

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

Installing the development branch

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

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

Manually install via Git

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

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

Then install by executing:

```
pip install -e pyEQL
```

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

1.1.2 Tutorial

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

Creating a Solution Object

Create a Solution object by invoking the Solution class:

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

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

More usefully, you can specify solutes and bulk properties:

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

Retrieving Solution Properties

Bulk Solution Properties

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

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

Individual Solute Properties

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

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

Units-Aware Calculations using pint

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

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

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

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

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

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

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

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

Using pyEQL in your projects

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

```
>>> import pyEQL
```

In order to directly create `Quantity` objects, you need to explicitly import the `unit` module:

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

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

1.1.3 The Solution Class

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

Creating a solution

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

Class reference

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

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*, *solvent: str | list = 'H2O'*, *engine: Literal['native', 'ideal'] = 'native'*, *database: str | Path | Store | None = None*)

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*, *solvent: str | list = 'H2O'*, *engine: Literal['native', 'ideal'] = 'native'*, *database: str | Path | Store | None = None*)

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 unit. 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.
- **solvent** – Formula of the solvent. Solvents other than water are not supported at this time.
- **engine** –

- **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.

Examples

```
>>> s1 = pyEQL.Solution([['Na+','1 mol/L'],['Cl-','1 mol/L']],temperature='20␣
↪degC',volume='500 mL')
>>> print(s1)
Components:
['H2O', 'Cl-', 'H+', 'OH-', 'Na+']
Volume: 0.5 l
Density: 1.0383030844030992 kg/l
```

property mass: Quantity

Return the total mass of the solution.

The mass is calculated each time this method is called. :param None:

Returns
Quantity

Return type
the mass of the solution, in kg

property solvent_mass

Return the mass of the solvent.

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

Returns
Quantity

Return type
the mass of the solvent, in kg

See also:

get_amount()

property volume: Quantity

Return the volume of the solution.

Returns:

Quantity: the volume of the solution, in L

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: Quantity

Return the pH of the solution.

p(*solute*, *activity=True*)

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

Return type
Quantity

property density: Quantity

Return the density of the solution.

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

Returns
Quantity

Return type
The density of the solution.

property dielectric_constant: Quantity

Returns the dielectric constant of the solution.

Parameters
None –

Returns
Quantity

Return type
the dielectric constant of the solution, dimensionless.

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 viscosity_dynamic: Quantity

Return the dynamic (absolute) viscosity of the solution.

Calculated from the kinematic viscosity

See Also:

viscosity_kinematic

property viscosity_kinematic

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

Compute the electrical conductivity of the solution.

Parameters
None –

Returns
The electrical conductivity of the solution in Siemens / meter.

Return type
Quantity

Notes

Conductivity is calculated by summing the molar conductivities of the respective solutes, but they are activity-corrected and adjusted using an empricial exponent. This approach is used in PHREEQC and Aqion models [aq] [hc]

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

Where:

$$\alpha = \begin{cases} \frac{0.6}{\sqrt{|z_i|}} & I < 0.36|z_i|\frac{\sqrt{I}}{|z_i|} \\ otherwise \end{cases}$$

Note: PHREEQC uses the molal rather than molar concentration according to http://wwwbrr.cr.usgs.gov/projects/GWC_coupled/phreeqc/phreeqc3-html/phreeqc3-43.htm

References

See also:

ionic_strength get_molar_conductivity() get_activity_coefficient()

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

- *Quantity* – The ionic strength of the parent solution, mol/kg.
- *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:

$$CB = \sum_i n_i z_i$$

where n_i is the number of moles, and z_i is the charge on species i.

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

Return type
float

property alkalinity

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 [stm]

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

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

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.

Parameters
None –

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

Return type
Quantity

property debye_length: Quantity

Return the Debye length of a solution.

Debye length is calculated as [wk3]

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

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

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.

Parameters
None –

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

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 [sata] [wk]

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

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
0.0
```

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

get_amount(*solute*, *units*)

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') 2. substance per mass of solvent (e.g., 'mol/kg') 3. mass of substance (e.g., 'kg') 4. moles of substance ('mol') 5. mole fraction ('fraction') 6. percent by weight (%) 7. number of molecules ('count')

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

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

See also:

add_amount, *set_amount*, *get_total_amount*, *get_osmolarity*, *get_osmolality*, get_mass, *get_total_moles_solute*

get_total_amount(*element*, *units*)

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

Parameters

- **element** (*str*) – String representing the name of the element of interest
- **units** (*str*) – Units desired for the output. Examples of valid units are 'mol/L','mol/kg','mol', 'kg', and 'g/L'

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

Notes

There is currently no way to distinguish between different oxidation states of the same element (e.g. TOTFe(II) vs. TOTFe(III)). This is planned for a future release.

See also:

get_amount

add_solute(*formula*, *amount*)

Primary method for adding substances to a pyEQL solution.

Parameters

- **formula** (*str*) – Chemical formula for the solute. Charged species must contain a + or - and (for polyvalent solutes) a number representing the net charge (e.g. 'SO4-2').
- **amount** (*str*) – The amount of substance in the specified unit system. The string should contain both a quantity and a pint-compatible representation of a unit. e.g. '5 mol/kg' or '0.1 g/L'

add_solvent(*formula*, *amount*)

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

add_amount(*solute*, *amount*)

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

Return type
Nothing. The concentration of solute is modified.

set_amount(*solute*, *amount*)

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.

Return type
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)`

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)`

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()`

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., 1 M MgCl2 yields 1 M Mg+2 and 2 M Cl-).

Parameters
None –

Returns
Salt object containing information about the parent salt.

Return type
Salt

See also:

get_activity(), *get_activity_coefficient()*, *get_water_activity()*, *get_osmotic_coefficient()*, `get_osmotic_pressure()`, `get_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() → dict

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.

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-).

Parameters
None –

Returns

- *dict* – A dictionary of Salt objects, keyed to the salt formula
- *See Also*
- ———
- *get_activity()*
- *get_activity_coefficient()*
- *get_water_activity()*
- *get_osmotic_coefficient()*
- get_osmotic_pressure()
- get_viscosity_kinematic()

get_activity_coefficient(*solute: str*, *scale: Literal['molal', 'molar', 'fugacity', 'rational'] = 'molal'*, *verbose: bool = False*)

Return the activity coefficient of a solute in solution.

The model used to calculte 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

Quantity: the activity coefficient as a dimensionless pint Quantity

get_activity(*solute: str*, *scale: Literal['molal', 'molar', 'rational'] = 'molal'*, *verbose: bool = False*)

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)

Notes

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

References

See also:

get_activity_coefficient() ionic_strength get_salt()

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

Return the osmotic coefficient of an aqueous solution.

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

get_water_activity()

Return the water activity.

Returns

The thermodynamic activity of water in the solution.

Return type

Quantity

See also:

get_activity_coefficient(), *ionic_strength*, *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=True*)

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

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

Return type

Quantity

Notes

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

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

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

_get_property(*solute: str*, *name: str*) → Quantity | 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
Quantity

Return type
The desired parameter or None if not found

get_transport_number(*solute*, *activity_correction=False*)

Calculate the transport number of the solute in the solution.

Parameters

- **solute** – String identifying the solute for which the transport number is to be calculated.
- **activity_correction** – If True, the transport number will be corrected for activity following the same method used for solution conductivity. Defaults to False if omitted.
- **Returns** – The transport number of *solute*
- **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.

If *activity_correction* is True, the contribution of each ion to the transport number is corrected with an activity factor. See the documentation for Solution.conductivity for an explanation of this correction.

- **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.

get_molar_conductivity(*solute*)

Calculate the molar (equivalent) conductivity for a solute.

Parameters
solute – String identifying the solute for which the molar conductivity is to be calculated.

Returns
The molar or equivalent conductivity of the species in the solution. Zero if the solute is not charged.

Notes

Molar conductivity is calculated from the Nernst-Einstein relation [smed]

$$\kappa_i = \frac{z_i^2 D_i F^2}{RT}$$

Note that the diffusion coefficient is strongly variable with temperature.

References

get_mobility(*solute*)

Calculate the ionic mobility of the solute.

Parameters
solute (*str*) – String identifying the solute for which the mobility is to be calculated.

Returns
float

Return type
The ionic mobility. Zero if the solute is not charged.

Notes

This function uses the Einstein relation to convert a diffusion coefficient into an ionic mobility [smed]

$$\mu_i = \frac{F|z_i|D_i}{RT}$$

References

get_lattice_distance(*solute*)

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
Quantity

Return type
The average distance between solute molecules

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:

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

copy()

Return a copy of the solution.

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().

list_solutes()

List all the solutes in the solution.

list_concentrations(*unit='mol/kg'*, *decimals=4*, *type='all'*)

List the concentration of each species in a solution.

Parameters

- **unit** (*str*) – String representing the desired concentration unit.
- **decimals** (*int*) – The number of decimal places to display. Defaults to 4.
- **type** (*str*) – The type of component to be sorted. Defaults to 'all' for all solutes. Other valid arguments are 'cations' and 'anions' which return lists of cations and anions, respectively.

Returns

Dictionary containing a list of the species in solution paired with their amount in the specified units

Return type

dict

list_activities(*decimals=4*)

List the activity of each species in a solution.

Parameters

decimals (*int*) – The number of decimal places to display. Defaults to 4.

Returns

Dictionary containing a list of the species in solution paired with their activity

Return type

dict

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(*v*)

pydantic Validator for MSONable pattern

1.1.4 Chemical Formulas

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

How to Enter Valid Chemical Formulas

Generally speaking, type the chemical formula of your solute the "normal" way and `pyEQL` should be able to inerpret it. Internally, `pyEQL` uses the `pymatgen.core.ion.Ion` class to "translate" chemical formulas into a consistent format. Anything that the `Ion` class can understand will be processed into a valid formula by `pyEQL`.

Here are some examples:

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

Specifically, `pyEQL` 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}'

Manually testing a formula

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

```
>>> from pymatgen.core.ion import Ion
>>> Ion.from_formula(<your_formula>).reduced_formula
...
```

Formulas you will see when using `Solution`

When using the `Solution` class,

- When creating a `Solution`, you can enter chemical formulas in any format you prefer, as long as `pymatgen` can understand it (see *manual testing*).
- The keys (solute formulas) in `Solution.components` are preserved in the same format the user enters them. So if you entered `Na+` for sodium ion, it will stay that way.
- Arguments to `Solution.get_property` can be entered in any format you prefer. When `pyEQL` queries the database, it will automatically convert the formula to the canonical one from `pymatgen`
- Property data in the database is uniquely identified by the canonical ion formula (output of `Ion.from_formula(<formula>).reduced_formla`, e.g. "Na[+1]" for sodium ion).

1.1.5 Property Database

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

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

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

Format

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

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

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

or

```
from maggma.core import JSONStore
```

(continues on next page)

(continued from previous page)

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

The Solute class

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

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

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

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

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

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

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

‘”

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

Searching the database

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

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

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

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

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

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

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

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

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

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

Species included

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

- Ac[+3]
- Ag(CN)2[-1]
- AgNO3(aq)
- Ag[+1]
- Ag[+2]
- Ag[+3]
- Al2(SO4)3(aq)
- Al[+3]
- AsO4[-3]
- Au(CN)2[-1]
- Au(CN)4[-1]

- Au[+1]
- Au[+2]
- Au[+3]
- B(H5C6)4[-1]
- B(OH)3(aq)
- B(OH)4[-1]
- BF4[-1]
- BO2[-1]
- Ba(ClO4)2(aq)
- Ba(NO3)2(aq)
- BaBr2(aq)
- BaC4O.3H2O(aq)
- BaCl2(aq)
- BaI2(aq)
- Ba[+2]
- BeSO4(aq)
- Be[+2]
- Bi[+3]
- BrO3[-1]
- Br[-0.33333333]
- Br[-1]
- C2N3[-1]
- CH3COO[-1]
- CNO[-1]
- CN[-1]
- CO3[-2]
- CSN[-1]
- CSeN[-1]
- Ca(ClO4)2(aq)
- Ca(NO3)2(aq)
- CaBr2(aq)
- CaCl2(aq)
- CaI2(aq)
- Ca[+2]
- Cd(ClO4)2(aq)
- Cd(NO2)2(aq)

- Cd(NO3)2(aq)
- CdSO4(aq)
- Cd[+2]
- CeCl3(aq)
- Ce[+3]
- Ce[+4]
- ClO2[-1]
- ClO3[-1]
- ClO4[-1]
- Cl[-1]
- Co(CN)6[-3]
- Co(H3N)6[-3]
- Co(NO3)2(aq)
- CoBr2(aq)
- CoCl2(aq)
- CoI2(aq)
- Co[+2]
- Co[+3]
- Cr(NO3)3(aq)
- CrCl3(aq)
- CrO4[-2]
- Cr[+2]
- Cr[+3]
- Cs2SO4(aq)
- CsBr(aq)
- CsCl(aq)
- CsF(aq)
- CsHC2O.1H2O(aq)
- CsI(aq)
- CsNO2(aq)
- CsNO3(aq)
- CsOH(aq)
- Cs[+1]
- Cu(NO3)2(aq)
- CuCl2(aq)
- CuSO4(aq)

- Cu[+1]
- Cu[+2]
- Cu[+3]
- Dy[+2]
- Dy[+3]
- Er[+2]
- Er[+3]
- Eu(NO3)3(aq)
- EuCl3(aq)
- Eu[+2]
- Eu[+3]
- F[-1]
- Fe(CN)6[-3]
- Fe(CN)6[-4]
- FeCl2(aq)
- FeCl3(aq)
- Fe[+2]
- Fe[+3]
- Ga[+3]
- GdCl3(aq)
- Gd[+3]
- Ge[+2]
- H2CO3(aq)
- H2O(aq)
- H2SNO3[-1]
- H2SO4(aq)
- H3O[+1]
- H4BrN(aq)
- H4IN(aq)
- H4N2O3(aq)
- H4NCl(aq)
- H4NClO4(aq)
- H4N[+1]
- H4SNO4(aq)
- H5C6O7[-3]
- H5N2[+1]

- H8S(NO2)2(aq)
- HBr(aq)
- HCO2[-1]
- HCO3[-1]
- HCl(aq)
- HClO4(aq)
- HF2[-1]
- HI(aq)
- HNO3(aq)
- HO2[-1]
- HOsO5[-1]
- HSO3[-1]
- HSO4[-1]
- HS[-1]
- HSeO3[-1]
- H[+1]
- Hf[+4]
- Hg[+2]
- Ho[+2]
- Ho[+3]
- IO3[-1]
- IO4[-1]
- I[-1]
- In[+1]
- In[+2]
- In[+3]
- IrO4[-1]
- Ir[+3]
- K2CO3(aq)
- K2PHO4(aq)
- K2SO4(aq)
- K3Fe(CN)6(aq)
- K3PO4(aq)
- K4Fe(CN)6(aq)
- KBr(aq)
- KBrO3(aq)

- KCSN(aq)
- KCl(aq)
- KClO3(aq)
- KClO4(aq)
- KCrO4(aq)
- KF(aq)
- KHC2O.1H2O(aq)
- KHCO3(aq)
- KI(aq)
- KNO2(aq)
- KNO3(aq)
- KOH(aq)
- KPO3.1H2O(aq)
- K[+1]
- La(NO3)3(aq)
- LaCl3(aq)
- La[+3]
- Li2SO4(aq)
- LiBr(aq)
- LiCl(aq)
- LiClO4(aq)
- LiHC2O.1H2O(aq)
- LiI(aq)
- LiNO2(aq)
- LiNO3(aq)
- LiOH(aq)
- Li[+1]
- Lu[+3]
- Mg(ClO4)2(aq)
- Mg(NO3)2(aq)
- MgBr2(aq)
- MgC4O.3H2O(aq)
- MgCl2(aq)
- MgI2(aq)
- MgSO4(aq)
- Mg[+2]

- MnCl2(aq)
- MnO4[-1]
- MnSO4(aq)
- Mn[+2]
- Mn[+3]
- MoO4[-2]
- Mo[+3]
- NO2[-1]
- NO3[-1]
- N[-0.33333333]
- Na2CO3(aq)
- Na2PHO4(aq)
- Na2S2O3(aq)
- Na2SO4(aq)
- Na3PO4(aq)
- NaBr(aq)
- NaBrO3(aq)
- NaCSN(aq)
- NaCl(aq)
- NaClO4(aq)
- NaCrO4(aq)
- NaF(aq)
- NaHC2O.1H2O(aq)
- NaHC3.2H2O(aq)
- NaHCO2(aq)
- NaHCO3(aq)
- NaI(aq)
- NaNO2(aq)
- NaNO3(aq)
- NaOH(aq)
- NaPO3.1H2O(aq)
- Na[+1]
- Nb[+3]
- Nd(NO3)3(aq)
- NdCl3(aq)
- Nd[+2]

- Nd[+3]
- Ni(NO3)2(aq)
- NiCl2(aq)
- NiSO4(aq)
- Ni[+2]
- Ni[+3]
- Np[+3]
- Np[+4]
- OH[-1]
- Os[+3]
- P(HO2)2[-1]
- P(OH)2[-1]
- P2O7[-4]
- P3O10[-5]
- PF6[-1]
- PH9(NO2)2(aq)
- PHO4[-2]
- PO3F[-2]
- PO3[-1]
- PO4[-3]
- Pa[+3]
- Pb(ClO4)2(aq)
- Pb(NO3)2(aq)
- Pb[+2]
- Pd[+2]
- Pm[+2]
- Pm[+3]
- Po[+2]
- PrCl3(aq)
- Pr[+2]
- Pr[+3]
- Pt[+2]
- Pu[+2]
- Pu[+4]
- Ra[+2]
- Rb2SO4(aq)

- RbBr(aq)
- RbCl(aq)
- RbF(aq)
- RbHC2O.1H2O(aq)
- RbI(aq)
- RbNO2(aq)
- RbNO3(aq)
- RbOH(aq)
- Rb[+1]
- ReO4[-1]
- Re[+1]
- Re[+3]
- Re[-1]
- Rh[+3]
- Ru[+2]
- Ru[+3]
- S2O3[-2]
- SO2[-1]
- SO3[-1]
- SO3[-2]
- SO4[-1]
- SO4[-2]
- S[-2]
- Sb(HO2)2[-1]
- Sb(OH)6[-1]
- ScCl3(aq)
- Sc[+2]
- Sc[+3]
- SeO3[-1]
- SeO4[-1]
- SeO4[-2]
- SiF6[-2]
- SmCl3(aq)
- Sm[+2]
- Sm[+3]
- Sn[+2]

- Sn[+4]
- Sr(ClO4)2(aq)
- Sr(NO3)2(aq)
- SrBr2(aq)
- SrCl2(aq)
- SrI2(aq)
- Sr[+2]
- Ta[+3]
- Tb[+3]
- TcO4[-1]
- Tc[+2]
- Tc[+3]
- Th(NO3)4(aq)
- Th[+4]
- Ti[+2]
- Ti[+3]
- Tl(ClO4)3(aq)
- Tl(NO2)3(aq)
- Tl(NO3)3(aq)
- TlH(C3O)2.4H2O(aq)
- Tl[+1]
- Tl[+3]
- Tm[+2]
- Tm[+3]
- U(ClO)2(aq)
- U(ClO5)2(aq)
- U(NO4)2(aq)
- UO2[+1]
- UO2[+2]
- USO6(aq)
- U[+3]
- U[+4]
- VO2[+1]
- V[+2]
- V[+3]
- WO4[-1]

- WO4[-2]
- W[+3]
- YCl3(aq)
- YNO3(aq)
- Y[+3]
- Yb[+2]
- Yb[+3]
- Zn(ClO4)2(aq)
- Zn(NO3)2(aq)
- ZnBr2(aq)
- ZnCl2(aq)
- ZnI2(aq)
- ZnSO4(aq)
- Zn[+2]
- Zr[+4]

1.1.6 Contributing to pyEQL

Reporting Issues

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

Contributing Code

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

Hacking pyEQL, step by step

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

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

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

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

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

```
git checkout -b mybranch
```

or

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

or

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

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

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

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

Guidelines

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

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

Documentation

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

Changelog

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

1.1.7 Functions Module

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

copyright
2013-2023 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

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

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

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

Returns
A pyEQL Solution object.

Return type
Solution

Notes

The following sections explain the different solution options:

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

References:

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

Return a solution object in equilibrium with fixed_charge.

Parameters

- **`solution`** (*`Solution object`*) – The external solution to be brought into equilibrium with the fixed charges

- **fixed_charge** (*str quantity*) – String representing the concentration of fixed charges, including sign. May be specified in mol/L or mol/kg units. e.g. '1 mol/kg'

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

Return type
Solution

Notes

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

In addition, electroneutrality must prevail within the membrane phase:

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

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

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

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

References

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

See Also:

get_salt()

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

Return the ideal mixing entropy associated with mixing two solutions.

Parameters

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

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

Return type
Quantity

Notes

The ideal entropy of mixing is calculated as follows

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

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

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

References

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

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

Return the Gibbs energy change associated with mixing two solutions.

Parameters

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

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

Return type
Quantity

Notes

The Gibbs energy of mixing is calculated as follows

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

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

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

References

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

pyEQL.functions.**mix**(*Solution1*, *Solution2*)

Mix two solutions together.

Returns a new Solution object that results from the mixing of Solution1 and Solution2

Parameters

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

Returns
A Solution object representing the mixed solution.

Return type
Solution

1.1.8 Internal module reference

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

Salt analysis module

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
2013-2023 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 [mistry].

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

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

Returns
Quantity

Return type
the effective molality of the salt in the parent solution

References

pyEQL.salt_ion_match.**_sort_components**(*Solution*, *type='all'*)

Sort the components of a solution in descending order (by mol).

Parameters

- **Solution** (*Solution object*) –
- **type** (*The type of component to be sorted. Defaults to 'all' for all*) – solutes. Other valid arguments are 'cations' and 'anions' which return sorted lists of cations and anions, respectively.

Returns

- *A list whose keys are the component names (formulas) and whose*
- *values are the component objects themselves*

pyEQL.salt_ion_match.**generate_salt_list**(*sol*, *unit='mol/kg'*)

Generate a list of salts that represents the ionic composition of a solution.

Returns
A dictionary of Salt objects, where Salt objects are the keys and the amounts are the values.

Return type
dict

pyEQL.salt_ion_match.**identify_salt**(*sol*)

Analyze the components of a solution and identify the salt that most closely approximates it. (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).

Create a Salt object for this salt.

Return type
A Salt object.

Activity Correction module

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
2013-2023 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

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

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

Parameters
temperature (*str Quantity, optional*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Notes

The parameter B is equal to:

$$B = (\frac{8\pi N_A e^2}{1000 \epsilon k T})^{\frac{1}{2}}$$

References

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

Examples:

```
>>> _debye_parameter_B()
0.3291...
```

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

Return the constant A for use in the Debye-Huckel limiting law (base 10).

Parameters
temperature (*str Quantity, optional*) – String representing the temperature of the solution. Defaults to '25 degC' if not specified.

Return type
Quantity The parameter A for use in the Debye-Huckel limiting law (base e)

Notes

The parameter A is equal to:

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

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. The value is often given in base 10 terms (0.509 at 25 degC) in older textbooks.

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.

Examples:

```
>>> _debye_parameter_activity()
1.17499...
```

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 (*str Quantity, optional*) – 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

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 (*str Quantity, optional*) – 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]_

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

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

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.

See Also:

_pitzer_f1

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

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

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

or

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

Parameters

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

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

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

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*, *formal_charge=1*, *temperature='25 degC'*)

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

Parameters

- **formal_charge** (*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

- *Quantity* – The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless.
- *See Also*
- ——
- *_debye_parameter_activity*

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

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*, *formal_charge=1*, *temperature='25 degC'*)

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

Parameters

- **formal_charge** (*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

- *Quantity* – The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless.
- *See Also*
- ——
- *_debye_parameter_activity*

Notes

Activity coefficient is calculated according to:

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

Valid only for I < 0.005

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*, *formal_charge=1*, *temperature='25 degC'*)

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

Parameters

- **formal_charge** (*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

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

- *See Also*
- ——
- *_debye_parameter_activity*

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

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** (*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*) – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta1** (*number*) – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **beta2** (*number*) – Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.
- **C_phi** (*number*) – 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

- *Quantity* – The mean molal (mol/kg) scale ionic activity coefficient of solute, dimensionless
- *Examples*
- ——
- *>>> get_activity_coefficient_pitzer(0.5*unit.Quantity('mol/kg'),0.5*unit.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*unit.Quantity('mol/kg'),5.6153*unit.Quantity('mol/kg'),3,0.5,0.036999 0.00171868,1,-1,1,1,b=1.2)*
- *0.76331…*
- **NOTE** (*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*unit.Quantity('mol/kg'),10*unit.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*unit.Quantity('mol/kg'),5*unit.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*unit.Quantity('mol/kg'),18*unit.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_gamma _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

- *Quantity* – The apparent molar volume of the solute, cm ** 3 / mol
- *Examples*
- ——
- **NOTE** (*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*unit.Quantity('mol/kg'),0.25*unit.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*unit.Quantity('mol/kg'),1.0*unit.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*unit.Quantity('mol/kg'),10.0*unit.Quantity('mol/kg'),2,0,0.000001742,0.0(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*unit.Quantity('mol/kg'),20.0*unit.Quantity('mol/kg'),2,0,0.000001742,0.0(1,1,1,b=1.2)*
- *51.145…*
- **NOTE** (*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*unit.Quantity('mol/kg'),0.8*unit.Quantity('mol/kg'),2,0,0.000024693,0.000(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

- *Quantity* – The osmotic coefficient of water, dimensionless
- *Examples*
- ——
- *Experimental value according to Beyer and Stieger reference is 1.3550*
- *>>> get_osmotic_coefficient_pitzer(10.175*unit.Quantity('mol/kg'),10.175*unit.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*unit.Quantity('mol/kg'),5.6153*unit.Quantity('mol/kg'),3,0.5,0.03699 0.00171868,1,-1,1,1,b=1.2)*
- *1.0850 . . .*
- **NOTE** (*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*unit.Quantity('mol/kg'),10*unit.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*unit.Quantity('mol/kg'),5*unit.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*unit.Quantity('mol/kg'),18*unit.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_gamma _pitzer_B_phi _pitzer_log_gamma

Speciation Engines module

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

copyright
2013-2023 by Ryan S. Kingsbury

license
LGPL, see LICENSE for more details.

class pyEQL.engines.**EOS**

Abstract base class for pyEQL equation of state classes.

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**

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 [may] is used. If no Pitzer parameters are available, then the appropriate equations are selected according to the following logic: [stumm].

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 [rbs]

$$f_{\pm} = \gamma_{\pm} * (1 + M_w \sum_i \nu_{ii})$$

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

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

See also:

get_ionic_strength get_salt activity_correction.get_activity_coefficient_debyehuckel activity_correction.get_activity_coefficient_guntelberg activity_correction.get_activity_coefficient_davies activity_correction.get_activity_coefficient_pitzer

Notes

For multicomponent mixtures, pyEQL implements the "effective Pitzer model" presented by Mistry et al. [mistry]. 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.

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

References

get_osmotic_coefficient(*solution*)

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

Osmotic coefficient is calculated using the Pitzer model. [may] 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 [rbs]

$$g = -\phi * M_w \sum_i \nu_{ii}) / \ln x_w$$

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** –
- **"molal"** (*The concentration scale for the returned osmotic coefficient. Valid options are*) –

:param : :param "rational": :type "rational": i.e., mole fraction :param coefficient is returned.:

Returns

Quantity:
The osmotic coefficient

See also:

get_water_activity get_ionic_strength get_salt

Notes

For multicomponent mixtures, pyEQL adopts the "effective Pitzer model" presented by Mistry et al. [mstry]. 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.

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.

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

[smed] Smedley, Stuart. The Interpretation of Ionic Conductivity in Liquids, pp 1-9. Plenum Press, 1980.

[smed] Smedley, Stuart I. The Interpretation of Ionic Conductivity in Liquids. Plenum Press, 1980.

[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.

[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," *J. Chemical Engineering Data* 33, pp.177-184.

[arch] 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.

[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.

PYTHON MODULE INDEX

p

pyEQL.activity_correction, 44
pyEQL.engines, 57
pyEQL.functions, 40
pyEQL.salt_ion_match, 43

INDEX

Symbols

__init__() (*pyEQL.Solution method*), 9
_debye_parameter_B() (*in module pyEQL.activity_correction*), 44
_debye_parameter_activity() (*in module pyEQL.activity_correction*), 45
_debye_parameter_osmotic() (*in module pyEQL.activity_correction*), 46
_debye_parameter_volume() (*in module pyEQL.activity_correction*), 46
_get_property() (*pyEQL.Solution method*), 22
_pitzer_B_MX() (*in module pyEQL.activity_correction*), 47
_pitzer_B_phi() (*in module pyEQL.activity_correction*), 48
_pitzer_f1() (*in module pyEQL.activity_correction*), 48
_pitzer_f2() (*in module pyEQL.activity_correction*), 49
_pitzer_log_gamma() (*in module pyEQL.activity_correction*), 49
_sort_components() (*in module pyEQL.salt_ion_match*), 44

A

add_amount() (*pyEQL.Solution method*), 17
add_solute() (*pyEQL.Solution method*), 17
add_solvent() (*pyEQL.Solution method*), 17
alkalinity (*pyEQL.Solution property*), 14
as_dict() (*pyEQL.Solution method*), 24
autogenerate() (*in module pyEQL.functions*), 40

B

bjerrum_length (*pyEQL.Solution property*), 15

C

charge_balance (*pyEQL.Solution property*), 13
conductivity (*pyEQL.Solution property*), 12
copy() (*pyEQL.Solution method*), 24

D

debye_length (*pyEQL.Solution property*), 14
density (*pyEQL.Solution property*), 11
dielectric_constant (*pyEQL.Solution property*), 11
donnan_eql() (*in module pyEQL.functions*), 40

E

entropy_mix() (*in module pyEQL.functions*), 41
EOS (*class in pyEQL.engines*), 57
equilibrate() (*pyEQL.engines.EOS method*), 57
equilibrate() (*pyEQL.engines.IdealEOS method*), 59
equilibrate() (*pyEQL.engines.NativeEOS method*), 59

F

from_dict() (*pyEQL.Solution class method*), 24

G

generate_salt_list() (*in module pyEQL.salt_ion_match*), 44
get_activity() (*pyEQL.Solution method*), 20
get_activity_coefficient() (*pyEQL.engines.EOS method*), 58
get_activity_coefficient() (*pyEQL.engines.IdealEOS method*), 59
get_activity_coefficient() (*pyEQL.engines.NativeEOS method*), 59
get_activity_coefficient() (*pyEQL.Solution method*), 19
get_activity_coefficient_davies() (*in module pyEQL.activity_correction*), 50
get_activity_coefficient_debyehuckel() (*in module pyEQL.activity_correction*), 51
get_activity_coefficient_guntelberg() (*in module pyEQL.activity_correction*), 51
get_activity_coefficient_pitzer() (*in module pyEQL.activity_correction*), 52
get_amount() (*pyEQL.Solution method*), 16
get_apparent_volume_pitzer() (*in module pyEQL.activity_correction*), 54
get_chemical_potential_energy() (*pyEQL.Solution method*), 21
get_effective_molality() (*pyEQL.salt_ion_match.Salt method*), 43

get_lattice_distance() (*pyEQL.Solution method*), 23
get_mobility() (*pyEQL.Solution method*), 23
get_molar_conductivity() (*pyEQL.Solution method*), 22
get_moles_solvent() (*pyEQL.Solution method*), 18
get_osmolality() (*pyEQL.Solution method*), 18
get_osmolarity() (*pyEQL.Solution method*), 18
get_osmotic_coefficient() (*pyEQL.engines.EOS method*), 58
get_osmotic_coefficient() (*pyEQL.engines.IdealEOS method*), 59
get_osmotic_coefficient() (*pyEQL.engines.NativeEOS method*), 60
get_osmotic_coefficient() (*pyEQL.Solution method*), 20
get_osmotic_coefficient_pitzer() (*in module pyEQL.activity_correction*), 55
get_salt() (*pyEQL.Solution method*), 18
get_salt_dict() (*pyEQL.Solution method*), 19
get_solute_volume() (*pyEQL.engines.EOS method*), 58
get_solute_volume() (*pyEQL.engines.IdealEOS method*), 59
get_solute_volume() (*pyEQL.engines.NativeEOS method*), 61
get_total_amount() (*pyEQL.Solution method*), 16
get_total_moles_solute() (*pyEQL.Solution method*), 17
get_transport_number() (*pyEQL.Solution method*), 22
get_water_activity() (*pyEQL.Solution method*), 20
gibbs_mix() (*in module pyEQL.functions*), 42

H

hardness (*pyEQL.Solution property*), 14

I

IdealEOS (*class in pyEQL.engines*), 59
identify_salt() (*in module pyEQL.salt_ion_match*), 44
ionic_strength (*pyEQL.Solution property*), 13

L

list_activities() (*pyEQL.Solution method*), 24
list_concentrations() (*pyEQL.Solution method*), 24
list_solutes() (*pyEQL.Solution method*), 24

M

mass (*pyEQL.Solution property*), 10
mix() (*in module pyEQL.functions*), 43
module
    pyEQL.activity_correction, 44
    pyEQL.engines, 57
    pyEQL.functions, 40
    pyEQL.salt_ion_match, 43

N

NativeEOS (*class in pyEQL.engines*), 59

O

osmotic_pressure (*pyEQL.Solution property*), 15

P

p() (*pyEQL.Solution method*), 10
pH (*pyEQL.Solution property*), 10
pressure (*pyEQL.Solution property*), 10
pyEQL.activity_correction
    module, 44
pyEQL.engines
    module, 57
pyEQL.functions
    module, 40
pyEQL.salt_ion_match
    module, 43

S

Salt (*class in pyEQL.salt_ion_match*), 43
set_amount() (*pyEQL.Solution method*), 17
Solution (*class in pyEQL*), 9
solvent_mass (*pyEQL.Solution property*), 10

T

temperature (*pyEQL.Solution property*), 10
to_json() (*pyEQL.Solution method*), 24

U

unsafe_hash() (*pyEQL.Solution method*), 24

V

validate_monty() (*pyEQL.Solution class method*), 25
viscosity_dynamic (*pyEQL.Solution property*), 11
viscosity_kinematic (*pyEQL.Solution property*), 12
volume (*pyEQL.Solution property*), 10