pyEQL Documentation

Release 0.6-dev

Ryan S. Kingsbury

Sep 19, 2018

Contents

Release 0.6-dev

Date Sep 19, 2018

Contents:

CHAPTER 1

Installation

1.1 Dependencies

pyEQL requires Python 3.0 or greater. We highly recommend the Anaconda distribution of Python, which bundles many other scientific computing packages and is easier to install (especially on Windows). You can download it at https://www.continuum.io/downloads.

pyEQL also requires the following packages:

- pint
- scipy

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

1.2 Automatically install via pip and PyPI

Once Python is installed, The Python Package Index repository will allow installation to be done easily from the command line as follows:

```
pip install pyEQL
```

This should automatically pull in the required dependencies as well.

Note: You may have to run 'pip3' rather than 'pip' if you intend to use your system's default Python installation rather than Anaconda. For example, on many Linux and Mac systems Python 2.x and Python 3.x are installed side-by-side. You can tell if this is the case on your system by going to a command line and typing 'python' like so:

```
$ python
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

1.3 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 Automatic installation section.

CHAPTER 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.

2.1 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')
```

2.2 Retrieving Solution Properties

2.2.1 Bulk Solution Properties

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

```
>>> s2.get_volume()
8.071524653929277 liter
>>> s2.get_density()
1.0182802742389558 kilogram/liter
>>> s2.get_conductivity()
4.083570230022633 siemens/meter
>>> s2.get_ionic_strength()
0.500000505903012 mole/kilogram
```

2.2.2 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+','diffusion_coefficient')
1.1206048116287536e-05 centimeter2/second
```

2.3 Units-Aware Calculations using pint

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

```
>>> from pyEQL import 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.

2.4 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 = pyEQL.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.

CHAPTER 3

Contributing to pyEQL

3.1 Reporting Issues

You can report any bugs, packaging issues, feature requests, comments, or questions using the issue tracker on github.

3.2 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!

It is **strongly** recommended that you read the following short articles before starting your work, especially if you are new to the open source community.

- Open Source Contribution Etiquette
- Don't "Push" Your Pull Requests
- A Successful Git Branching Model

3.2.1 Hacking pyEQL in Six Easy Steps:

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. Create a branch for your work. We loosely follow the branching guidelines outlined at http://nvie.com/posts/a-successful-git-branching-model.

   If you are adding **documentation** or **bug fixes**, start with the **master** branch and prefix your branch with "fix-" or "doc-" as appropriate:

```
git checkout -b fix-myfix master

git checkout -b doc-mydoc master
```

If you are adding a **new feature**, start with the **develop** branch and prefix your branch with "feature-":

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

4. Hack away until you're satisfied.
5. Push your work back to Github:

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

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

3.3 Generating Test Cases

pyEQL has many capabilities that have not been tested thoroughly. You can help the project simply by using pyEQL and comparing the output to experimental data and/or more established models. Report back your results on the issue tracker.

Even better, write up an automated test case (see the tests/ directory for examples).

3.4 Making a Donation

If you'd like to leave a 'tip' for the project maintainer to support the time and effort required to develop pyEQL, simply send it via Paypal to RyanSKingsbury@alumni.unc.edu

CHAPTER 4

Chemical Formulas

4.1 Representing Chemical Substances in pyEQL

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.

4.1.1 How to Enter Valid Chemical Formulas

Generally speaking, type the chemical formula of your solute the "normal" way and pyEQL should be able to inerpret it. Here are some examples:

- Sodium Chloride - NaCl
- Sodium Sulfate - Na(SO4)2
- Methanol - CH4OH or CH5O
- Magnesium Ion - Mg+2
- Chloride Ion - Cl-

Formula Rules:

1. Are composed of valid atomic symbols that start with capital letters
2. Contain no non-alphanumeric characters other than '(', ')', '+', or '-'
3. If a '+' or '-' is present, the formula must contain ONLY '+' or '-' (e.g. 'Na+-' is invalid) and the formula must end with either a series of charges (e.g. 'Fe+++') or a numeric charge (e.g. 'Fe+3')
4. Formula must contain matching numbers of '(' and ')'
5. Open parentheses must precede closed parentheses

4.1.2 Alternate Formulas and Isomers

Many complex molecules can be written in multiple ways. pyEQL cares only about the number and identity of the elements and the formal charge on the molecule, so you can use any form you choose. The hill_order() method takes a formula and reduces it to its simplest form, like so:

```
>>> pyEQL.chemical_formula.hill_order('CH2(CH3)4COOH')
'C6H15O2'
```

When searching the parameters database, pyEQL uses this method to reduce both user-entered formulas AND keys in the database. So even if you created a solution containing 'ClNa', pyEQL would still match it with parameters for 'NaCl'.

Currently pyEQL **does not distinguish between isomers**.

4.2 API Documentation (chemical_formula.py)

This module contains classes, functions, and methods to facilitate the input, output, and parsing of chemical formulas for pyEQL.

The correct case must be used when specifying elements.

copyright 2013-2018 by Ryan S. Kingsbury

license LGPL, see LICENSE for more details.

pyEQL.chemical_formula.**contains**(*formula*, *element*)

Check whether a formula contains a given element.

Parameters

formula: str String representing a molecular formula. e.g. 'H2O' or 'FeOH+' Valid molecular formulas must meet the following criteria:

1. Are composed of valid atomic symbols that start with capital letters
2. Contain no non-alphanumeric characters other than '(', ')', '+', or '-'
3. If a '+' or '-' is present, the formula must contain ONLY '+' or '-' (e.g. 'Na+-' is invalid) and the formula must end with either a series of charges (e.g. 'Fe+++') or a numeric charge (e.g. 'Fe+3')
4. Formula must contain matching numbers of '(' and ')'
5. Open parentheses must precede closed parentheses

element: str String representing the element to check for. Must be a valid element name.

Returns

bool True if the formula contains the element. False otherwise.

Examples

```
>>> contains('Fe2(SO4)3','Fe')
True
>>> contains('NaCOOH','S')
False
```

pyEQL.chemical_formula.**get_element_mole_ratio**(*formula*, *element*)
compute the moles of a specific element per mole of formula

Parameters

formula: str String representing a molecular formula. e.g. 'H2O' or 'FeOH+' Valid molecular formulas must meet the following criteria:

1. Are composed of valid atomic symbols that start with capital letters
2. Contain no non-alphanumeric characters other than '(', ')', '+', or '-'
3. If a '+' or '-' is present, the formula must contain ONLY '+' or '-' (e.g. 'Na+-' is invalid) and the formula must end with either a series of charges (e.g. 'Fe+++') or a numeric charge (e.g. 'Fe+3')
4. Formula must contain matching numbers of '(' and ')'
5. Open parentheses must precede closed parentheses

element: str String representing the element to check for. Must be a valid element name.

Returns

number The number of moles of element per mole of formula, mol/mol.

>>> get_element_mole_ratio('NaCl','Na')

1

>>> get_element_mole_ratio('H2O','H')

2

>>> get_element_mole_ratio('H2O','Br')

0

>>> get_element_mole_ratio('CH3CH2CH3','C')

3

See also:

contains, consolidate_formula, *get_element_weight*, *get_element_weight_fraction*

pyEQL.chemical_formula.**get_element_names**(*formula*)
Return the names of the elements in a chemical formula

Parameters

formula: str String representing a chemical formula

Examples

```
>>> get_element_names('FeSO4')
['Iron', 'Sulfur', 'Oxygen']
```

pyEQL.chemical_formula.**get_element_numbers**(*formula*)
Return the atomic numbers of the elements in a chemical formula

Parameters

formula: str String representing a chemical formula

Examples

```
>>> get_element_numbers('FeSO4')
[26, 16, 8]
```

pyEQL.chemical_formula.**get_element_weight**(*formula*, *element*)
compute the weight of a specific element in a formula

Parameters

formula: str String representing a molecular formula. e.g. 'H2O' or 'FeOH+' Valid molecular formulas must meet the following criteria:

1. Are composed of valid atomic symbols that start with capital letters
2. Contain no non-alphanumeric characters other than '(', ')', '+', or '-'
3. If a '+' or '-' is present, the formula must contain ONLY '+' or '-' (e.g. 'Na+-' is invalid) and the formula must end with either a series of charges (e.g. 'Fe+++') or a numeric charge (e.g. 'Fe+3')
4. Formula must contain matching numbers of '(' and ')'
5. Open parentheses must precede closed parentheses

element: str String representing the element to check for. Must be a valid element name.

Returns

number The weight of the specified element within the formula, g/mol.

>>> get_element_weight('NaCl','Na')

22.98977

>>> get_element_weight('H2O','H')

2.01588

>>> get_element_weight('H2O','Br')

0.0

>>> get_element_weight('CH3CH2CH3','C')

36.0321

See also:

contains, _consolidate_formula, elements, *get_element_mole_ratio*

pyEQL.chemical_formula.**get_element_weight_fraction**(*formula*, *element*)
compute the weight fraction of a specific element in a formula

Parameters

formula: str String representing a molecular formula. e.g. 'H2O' or 'FeOH+' Valid molecular formulas must meet the following criteria:

1. Are composed of valid atomic symbols that start with capital letters
2. Contain no non-alphanumeric characters other than '(', ')', '+', or '-'
3. If a '+' or '-' is present, the formula must contain ONLY '+' or '-' (e.g. 'Na+-' is invalid) and the formula must end with either a series of charges (e.g. 'Fe+++') or a numeric charge (e.g. 'Fe+3')

4. Formula must contain matching numbers of '(' and ')'
5. Open parentheses must precede closed parentheses

element: str String representing the element to check for. Must be a valid element name.

Returns

number The weight fraction of the specified element within the formula.

>>> get_element_weight_fraction('NaCl','Na')

0.39337...

>>> get_element_weight_fraction('H2O','H')

0.111898...

>>> get_element_weight_fraction('H2O','Br')

0.0

>>> get_element_weight_fraction('CH3CH2CH3','C')

0.8171355...

See also:

get_element_weight, *contains*, _consolidate_formula, elements

pyEQL.chemical_formula.**get_elements**(*formula*)

Return a list of strings representing the elements in a molecular formula, with no duplicates.

See also:

_check_formula

Examples

```
>>> get_elements('FeSO4')
['Fe', 'S', 'O']
>>> get_elements('CH3(CH2)4(CO)3')
['C', 'H', 'O']
```

pyEQL.chemical_formula.**get_formal_charge**(*formula*)

Return the formal charge on a molecule based on its formula

See also:

_check_formula

Examples

```
>>> get_formal_charge('Na+')
1
>>> get_formal_charge('PO4-3')
-3
>>> get_formal_charge('Fe+++')
3
```

pyEQL.chemical_formula.**get_molecular_weight**(*formula*)

compute the molecular weight of a formula

```
>>> get_molecular_weight('Na+')
22.98977
>>> get_molecular_weight('H2O')
18.01528
>>> get_molecular_weight('CH3CH2CH3')
44.09562
```

See also:

`_consolidate_formula`, `elements`

pyEQL.chemical_formula.**hill_order**(*formula*)

Return a string representing the simplest form of 'formula' in the Hill order (Carbon, Hydrgen, then other elements in alphabetical order). If no Carbon is present, then all elements are listed in alphabetical order.

NOTE: this function does NOT (yet) honor exceptions to the Hill Order for acids, hydroxides, oxides, and ionic compounds. It follows the rule above no matter what.

Examples

```
>>> hill_order('CH2(CH3)4COOH')
'C6H15O2'
```

```
>>> hill_order('NaCl')
'ClNa'
```

```
>>> hill_order('NaHCO2') == hill_order('HCOONa')
True
```

```
>>> hill_order('Fe+2') == hill_order('Fe+3')
False
```

pyEQL.chemical_formula.**is_valid_element**(*formula*)

Check whether a string is a valid atomic symbol

Parameters

:formula: str String representing an atomic symbol. First letter must be uppercase, second letter must be lowercase.

Returns

bool True if the string is a valid atomic symbol. False otherwise.

Examples

```
>>> is_valid_element('Cu')
True
>>> is_valid_element('Na+')
False
```

pyEQL.chemical_formula.**is_valid_formula**(*formula*)

Check that a molecular formula is formatted correctly

Parameters

formula: str String representing a molecular formula. e.g. 'H2O' or 'FeOH+' Valid molecular formulas must meet the following criteria:

1. Are composed of valid atomic symbols that start with capital letters
2. Contain no non-alphanumeric characters other than '(', ')', '+', or '-'
3. If a '+' or '-' is present, the formula must contain ONLY '+' or '-' (e.g. 'Na+-' is invalid) and the formula must end with either a series of charges (e.g. 'Fe+++') or a numeric charge (e.g. 'Fe+3')
4. Formula must contain matching numbers of '(' and ')'
5. Open parentheses must precede closed parentheses

Returns

bool True if the formula is valid. False otherwise.

Examples

```
>>> is_valid_formula('Fe2(SO4)3')
True
>>> is_valid_formula('2Na+')
False
>>> is_valid_formula('HCO3-')
True
>>> is_valid_formula('Na+-')
False
>>> is_valid_formula('C10h12')
False
```

pyEQL.chemical_formula.**print_latex**(*formula*)

Print a LaTeX - formatted version of the formula

Examples

```
>>> print_latex('Fe2SO4')
Fe_2SO_4
>>> print_latex('CH3CH2CH3')
CH_3CH_2CH_3
>>> print_latex('Fe2(OH)2+2')
Fe_2(OH)_2^+^2
```

CHAPTER 5

Database System

pyEQL creates a database to collect various parameters needed to perform it's calculations. pyEQL's default database includes a collection of the following parameters for some common electrolytes:

- 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 10 ions

5.1 Basics

The Paramsdb class creates a container for parameters. Each paramter is an object which contains not only the value, but also information about the units, the reference, and the conditions of measurement. paramsdb() also defines several methods that are helpful for retrieving parameters.

pyEQL automatically initializes an instance of Paramsdb under the name 'db'. You can access database methods like this:

```
>>> import pyEQL
>>> pyEQL.db
<pyEQL.database.Paramsdb at 0x7fead183f240>
>>> pyEQL.db.has_species('H+')
True
```

Anytime a new solute is added to a solution, the search_parameters() method is called. This method searches every database file within the search path (by default, only pyEQL's built-in databases) for any parameters associated with that solute, and adds them to the database.

5.2 Adding your own Database Files

5.2.1 Custom Search Paths

The database system is meant to be easily extensible. To include your own parameters, first you need to add a directory of your choosing to the search path.

```
>>> pyEQL.db.add_path('/home/user')
```

You can always check to see which paths pyEQL is searching by using list_path():

```
>>> pyEQL.db.list_path()
<default installation directory>/database
/home/user
```

Then, place your custom database file inside that directory. **NOTE: custom database files are searched IN ADDITION TO the default databases.** You don't need to re-create the information from the built-in files. Custom databases only need to contain extra parameters that are not included already.

5.2.2 File Format

Databases are formatted as TAB-SEPARATED text files and carry the .tsv extension. The intent of this format is to make database files easy to edit with common spreadsheet software.

> **Warning:** If you open an existing or template database file for editing, some spreadsheet software will try to replace the tabs with commas when you save it again. pyEQL does NOT read comma-separated files.

Since pyEQL compiles the database from multiple files, the intent is for each file to contain values for one type of parameter (such as a diffusion coefficient) from one source. The file can then list values of that parameter for a number of different solutes.

The upper section of each file contains information about the source of the data, the units, the name of the parameter, and the conditions of measurement. The top of each database file must, at a minimum, contain rows for 'Name' and 'Units'. Preferably, other information such as conditions, notes and a reference are also supplied. See *template.tsv* in the database subdirectory for an example.

The remainder of the file contains solute formulas in the first column (see *Chemical Formulas*) and corresponding values of the parameter in the following columns. Sets of parameters (such as activity correction coefficients) can be specified by using more than one column.

> **Warning:** Currently there is no way to handle duplicated parameters. So if you supply a parameter with the same name as a built-in one, unexpected behavior may result.

5.2.3 Special Names

The name of a parameter is used as a kind of index within pyEQL. Certain methods expect certain parameter names. The following are the currently-used internal names:

- 'diffusion_coefficient' - diffusion coefficient
- 'pitzer_parameters_activity' - coefficients for the Pitzer model for activity correction

- 'pitzer_parameters_volume'- coefficients for the Pitzer model for partial molar volume
- 'erying_viscosity_coefficients' - coefficients for an Erying-type viscosity correction model
- 'partial_molar_volume'- the partial molar volume (used if Pitzer parameters are not available)
- 'hydrated_radius' - hydrated radius
- 'ionic_radius' - ionic radius
- 'jones_dole_B' - Jones-Dole "B" coefficient

If you wish to supply these parameters for a custom solute not included in the built-in database, make sure to format the name exactly the same way.

You can also specify a custom parameter name, and retrieve it using the get_parameter() method. If the solute is 'Na+'

```
>>> pyEQL.db.get_parameter('Na+','my_parameter_name')
```

5.3 Viewing the Database

You can view the entire contents of the database using the print_database() method. Since pyEQL searches for parameters as they are added, the database will only contain parameters for solutes that have actually been used during the execution of your script. The output is organized by solute.

```
>>> pyEQL.db.print_database()

>>> s1 = pyEQL.Solution([['Na+','0.5 mol/kg'],['Cl-','0.5 mol/kg']])
>>> pyEQL.db.print_database()
Parameters for species Cl-:
--------------------------
Parameter diffusion_coefficient
Diffusion Coefficient
-----------------------------------------
Value: 2.032e-05 cm^2/s
Conditions (T,P,Ionic Strength): 25 celsius, 1 atm, 0
Notes: For most ions, increases 2-3% per degree above 25C
Reference: CRC Handbook of Chemistry and Physics, 92nd Ed., pp. 5-77 to 5-79

Parameter partial_molar_volume
Partial molar volume
---------------------------------------
Value: 21.6 cm^3/mol
Conditions (T,P,Ionic Strength): 25 celsius, 1 atm, 0
Notes: correction factor 5e-4 cm3/g-K
Reference: Durchschlag, H., Zipper, P., 1994. "Calculation of the Partial Molal␣
↪Volume of Organic Compounds and Polymers." Progress in Colloid & Polymer Science␣
↪(94), 20-39.
...
```

5.4 API Documentation (database.py)

This module contains classes, functions, and methods for reading input files and assembling database entries for use by pyEQL.

By default, pyEQL searches all files in the /database subdirectory for parameters.

copyright 2013-2018 by Ryan S. Kingsbury

license LGPL, see LICENSE for more details.

class pyEQL.database.**Paramsdb**

create a global dictionary to contain a dynamically-generated list of Parameters for solute species. The dictionary keys are the individual chemical species formulas. The dictionary's values are a python set object containing all parameters that apply to the species.

Methods

add_parameter(formula, parameter)	Add a parameter to the database
add_path(path)	Add a user-defined directory to the database search path
get_parameter(formula, name)	Retrieve a parameter from the database
has_parameter(formula, name)	Boolean test to determine whether a parameter exists in the database for a given species
has_species(formula)	Boolean test to determine whether a species is present in the database
list_path()	List all search paths for database files
print_database([solute])	Function to generate a human-friendly summary of all the database parameters that are actually used in the simulation
search_parameters(formula)	Each time a new solute species is created in a solution, this function:

add_parameter(*formula*, *parameter*)

Add a parameter to the database

add_path(*path*)

Add a user-defined directory to the database search path

get_parameter(*formula*, *name*)

Retrieve a parameter from the database

has_parameter(*formula*, *name*)

Boolean test to determine whether a parameter exists in the database for a given species

has_species(*formula*)

Boolean test to determine whether a species is present in the database

list_path()

List all search paths for database files

print_database(*solute=None*)

Function to generate a human-friendly summary of all the database parameters that are actually used in the simulation

Parameters

solute [str, optional] The chemical formula for a species. If this argument of supplied, the output will contain only the database entries for this species. Otherwise, all database entries will be printed.

search_parameters(*formula*)

Each time a new solute species is created in a solution, this function:

1) searches to see whether a list of parameters for the species has already been compiled from the database 2) searches all files in the specified database directory(ies) for the species 3) creates a Parameter object for each value found 4) compiles these objects into a set 5) adds the set to a dictionary indexed by species name (formula) 6) points the new solute object to the dictionary

formula [str] String representing the chemical formula of the species.

5.5 API Documentation (parameter.py)

This module implements the Parameter() class, which is used to store values, units, uncertainties, and reference data for various quantities used throughout pyEQL.

copyright 2013-2018 by Ryan S. Kingsbury

license LGPL, see LICENSE for more details.

class pyEQL.parameter.**Parameter**(*name*, *magnitude*, *units=''*, ***kwargs*)

Class for storing and retrieving measured parameter values together with their units, context, and reference information.

Some pyEQL functions search for specific parameter names, such as: diffusion_coefficient

Methods

get_dimensions()	Return the dimensions of the parameter.
get_magnitude([temperature, pressure, ...])	Return the magnitude of a parameter at the specified conditions.
get_name()	Return the name of the parameter.
get_units()	Return the units of a parameter
get_value([temperature, pressure, ...])	Return the value of a parameter at the specified conditions.

get_dimensions()

Return the dimensions of the parameter.

get_magnitude(*temperature=None*, *pressure=None*, *ionic_strength=None*)

Return the magnitude of a parameter at the specified conditions.

Parameters

temperature [str, optional] The temperature at which 'magnitude' was measured in degrees Celsius. Specify the temperature as a string containing the magnitude and a unit, e.g. '25 degC', '32 degF', '298 kelvin', and '500 degR'

pressure [str, optional] The pressure at which 'magnitude' was measured in Pascals Specify the pressure as a string containing the magnitude and a unit. e.g. '101 kPa'. Typical valid units are 'Pa', 'atm', or 'torr'.

ionic_strength [str, optional] The ionic strength of the solution in which 'magnitude' was measured. Specify the ionic strength as a string containing the magnitude and a unit. e.g. '2 mol/kg'

Returns

Number The magnitude of the parameter at the specified conditions.

get_name()
Return the name of the parameter.

Parameters

None

Returns

str The name of the parameter

get_units()
Return the units of a parameter

get_value(*temperature=None*, *pressure=None*, *ionic_strength=None*)
Return the value of a parameter at the specified conditions.

Parameters

temperature [str, optional] The temperature at which 'magnitude' was measured in degrees Celsius. Specify the temperature as a string containing the magnitude and a unit, e.g. '25 degC', '32 degF', '298 kelvin', and '500 degR'

pressure [str, optional] The pressure at which 'magnitude' was measured in Pascals Specify the pressure as a string containing the magnitude and a unit. e.g. '101 kPa'. Typical valid units are 'Pa', 'atm', or 'torr'.

ionic_strength [str, optional] The ionic strength of the solution in which 'magnitude' was measured. Specify the ionic strength as a string containing the magnitude and a unit. e.g. '2 mol/kg'

Returns

Quantity The value of the parameter at the specified conditions.

CHAPTER 6

The Solution Class

pyEQL Solution Class

copyright 2013-2018 by Ryan S. Kingsbury

license LGPL, see LICENSE for more details.

class pyEQL.solution.**Solution**(*solutes=[]*, ***kwargs*)

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

Parameters

solutes [list of lists, optional] See add_solute() documentation for formatting of this list. 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

Returns

Solution A Solution object.

See also:

add_solute

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
```

Methods

add_amount(solute, amount)	Add the amount of 'solute' to the parent solution.
add_solute(formula, amount[, parameters])	Primary method for adding substances to a pyEQL solution
add_solvent(formula, amount)	Same as add_solute but omits the need to pass solvent mass to pint
copy()	Return a copy of the solution
get_activity(solute[, scale, verbose])	Return the thermodynamic activity of the solute in solution on the molal scale.
get_activity_coefficient(solute[, scale, ...])	Return the activity coefficient of a solute in solution.
get_alkalinity()	Return the alkalinity or acid neutralizing capacity of a solution
get_amount(solute, units)	Return the amount of 'solute' in the parent solution.
get_bjerrum_length()	Return the Bjerrum length of a solution
get_charge_balance()	Return the charge balance of the solution.
get_chemical_potential_energy([...])	Return the total chemical potential energy of a solution (not including pressure or electric effects)
get_conductivity()	Compute the electrical conductivity of the solution.
get_debye_length()	Return the Debye length of a solution
get_density()	Return the density of the solution.
get_dielectric_constant()	Returns the dielectric constant of the solution.
get_hardness()	Return the hardness of a solution.
get_ionic_strength()	Return the ionic strength of the solution.
get_lattice_distance(solute)	Calculate the average distance between molecules
get_mass()	Return the total mass of the solution.
get_mobility(solute)	Calculate the ionic mobility of the solute
get_molar_conductivity(solute)	Calculate the molar (equivalent) conductivity for a solute
get_mole_fraction(solute)	Return the mole fraction of 'solute' in the solution
get_moles_solvent()	Return the moles of solvent present in the solution
get_osmolality([activity_correction])	Return the osmolality of the solution in Osm/kg
get_osmolarity([activity_correction])	Return the osmolarity of the solution in Osm/L
get_osmotic_coefficient([scale])	Return the osmotic coefficient of an aqueous solution.
get_osmotic_pressure()	Return the osmotic pressure of the solution relative to pure water.
get_pressure()	Return the hydrostatic pressure of the solution.

Continued on next page

Table 1 – continued from previous page

get_property(solute, name)	Retrieve a thermodynamic property (such as diffusion coefficient) for solute, and adjust it from the reference conditions to the conditions of the solution
get_salt()	Determine the predominant salt in a solution of ions.
get_salt_list()	Determine the predominant salt in a solution of ions.
get_solute(i)	Return the specified solute object.
get_solvent()	Return the solvent object.
get_solvent_mass()	Return the mass of the solvent.
get_temperature()	Return the temperature of the solution.
get_total_amount(element, units)	Return the total amount of 'element' (across all solutes) in the solution.
get_total_moles_solute()	Return the total moles of all solute in the solution
get_transport_number(solute[, ...])	Calculate the transport number of the solute in the solution
get_viscosity_dynamic()	Return the dynamic (absolute) viscosity of the solution.
get_viscosity_kinematic()	Return the kinematic viscosity of the solution.
get_viscosity_relative()	Return the viscosity of the solution relative to that of water
get_volume()	Return the volume of the solution.
get_water_activity()	Return the water activity.
list_activities([decimals])	List the activity of each species in a solution.
list_concentrations([unit, decimals, type])	List the concentration of each species in a solution.
list_solutes()	List all the solutes in the solution.
p(solute[, activity])	Return the negative log of the activity of solute.
set_amount(solute, amount)	Set the amount of 'solute' in the parent solution.
set_pressure(pressure)	Set the hydrostatic pressure of the solution.
set_temperature(temperature)	Set the solution temperature.
set_volume(volume)	Change the total solution volume to volume, while preserving all component concentrations

list_salts

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

Returns

Nothing. The concentration of solute is modified.

See also:

`Solute.add_moles`

add_solute (*formula*, *amount*, *parameters={}*)

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'

parameters [dictionary, optional] Dictionary of custom parameters, such as diffusion coefficients, transport numbers, etc. Specify parameters as key:value pairs separated by commas within curly braces, e.g. {diffusion_coeff:5e-10,transport_number:0.8}. The 'key' is the name that will be used to access the parameter, the value is its value.

add_solvent(*formula*, *amount*)
Same as add_solute but omits the need to pass solvent mass to pint

copy()
Return a copy of the solution

TODO - clarify whether this is a deep or shallow copy

get_activity(*solute*, *scale='molal'*, *verbose=False*)
Return the thermodynamic activity of the solute in solution on the molal scale.

Parameters

solute [str] String representing the name of the solute of interest

scale [str, optional] 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 [bool, optional] 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)

See also:

get_activity_coefficient, *get_ionic_strength*, *get_salt*

Notes

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

References

get_activity_coefficient(*solute*, *scale='molal'*, *verbose=False*)
Return the activity coefficient of a solute in solution.

Whenever the appropriate parameters are available, the Pitzer model[1] is used. If no Pitzer parameters are available, then the appropriate equations are selected according to the following logic:[2]

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

[2] 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

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[3]

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

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

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 [str] String representing the name of the solute of interest

scale [str, optional] 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 [bool, optional] 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.[4]. 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.

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

[3] Stumm, Werner and Morgan, James J. *Aquatic Chemistry*, 3rd ed, pp 165. Wiley Interscience, 1996.
[4] Robinson, R. A.; Stokes, R. H. Electrolyte Solutions: Second Revised Edition; Butterworths: London, 1968, p.32.

References

get_alkalinity()
Return the alkalinity or acid neutralizing capacity of a solution

Returns

Quantity : The alkalinity of the solution in mg/L as CaCO3

Notes

The alkalinity is calculated according to:[5]

$$Alk = F\sum_i z_i C_B - \sum_i z_i C_A$$

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 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

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') 1. substance per mass of solvent (e.g., 'mol/kg') 1. mass of substance (e.g., 'kg') 1. moles of substance ('mol') 1. mole fraction ('fraction') 1. percent by weight (%)

Parameters

solute [str] String representing the name of the solute of interest

units [str] Units desired for the output. Examples of valid units are 'mol/L','mol/kg','mol', 'kg', and 'g/L' Use 'fraction' to return the mole fraction. Use '%' to return the mass percent

Returns

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

See also:

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

get_bjerrum_length()
Return the Bjerrum length of a solution

Bjerrum length representes the distance at which electrostatic interactions between particles become comparable in magnitude to the thermal energy.:math:*lambda_B* is calculated as[6]

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

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

[5] Stumm, Werner and Morgan, James J. Aquatic Chemistry, 3rd ed, pp 165. Wiley Interscience, 1996.
[6] https://en.wikipedia.org/wiki/Bjerrum_length

Parameters

None

Returns

Quantity The Bjerrum length, in nanometers.

See also:

get_dielectric_constant

References

Examples

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

get_charge_balance()

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.

Returns

float : The charge balance of the solution, in equivalents.

Notes

The charge balance is calculated according to:

$$CB = F \sum_i n_i z_i$$

Where n_i is the number of moles, z_i is the charge on species i, and F is the Faraday constant.

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

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

Notes

The chemical potential energy (related to the Gibbs mixing energy) is calculated as follows:[7]

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

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_conductivity()

Compute the electrical conductivity of the solution.

Parameters

None

Returns

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

See also:

get_ionic_strength, *get_molar_conductivity*, *get_activity_coefficient*

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[8][9]

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

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

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

[8] http://www.aqion.de/site/77

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

References

get_debye_length()
Return the Debye length of a solution

Debye length is calculated as[10]

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

where I is the ionic strength, $epsilon_r$ and $epsilon_r$ 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.

Parameters

None

Returns

Quantity The Debye length, in nanometers.

See also:

get_ionic_strength, *get_dielectric_constant*

References

get_density()
Return the density of the solution.

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

Returns

Quantity: The density of the solution.

get_dielectric_constant()
Returns the dielectric constant of the solution.

Parameters

None

Returns

Quantity: the dielectric constant of the solution, dimensionless.

Notes

Implements the following equation as given by[11]

$$\frac{\epsilon = \epsilon_s olvent}{1 + \sum_i \alpha_i x_i}$$

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

[10] https://en.wikipedia.org/wiki/Debye_length#Debye_length_in_an_electrolyte
[11] [1] A. Zuber, L. Cardozo-Filho, V.F. Cabral, R.F. Checoni, M. Castier,

References

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.

get_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

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

get_ionic_strength()
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.get_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.get_ionic_strength()
<Quantity(1.0000001004383303, 'mole / kilogram')>
```

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 [The average distance between solute molecules]

Notes

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

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

Examples

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

get_mass()

Return the total mass of the solution.

The mass is calculated each time this method is called. Parameters ———- None

Returns

Quantity: the mass of the solution, in kg

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 [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[12]

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

References

get_molar_conductivity(*solute*)

Calculate the molar (equivalent) conductivity for a solute

Parameters

solute [str] String identifying the solute for which the molar conductivity is to be calculated.

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

Returns

float 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[13]

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

Note that the diffusion coefficient is strongly variable with temperature.

References

Examples

TODO

get_mole_fraction(*solute*)
Return the mole fraction of 'solute' in the solution

Notes

This function is DEPRECATED and will raise a warning when called. Use get_amount() instead and specify 'fraction' as the unit type.

get_moles_solvent()
Return the moles of solvent present in the solution

Parameters

None

Returns

Quantity The moles of solvent in the solution.

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_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.

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

get_osmotic_coefficient(*scale='molal'*)

Return the osmotic coefficient of an aqueous solution.

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

If the 'rational' scale is given as input, then the molal-scale osmotic coefficient ϕ is converted according to[14]

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

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

Returns

——-

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.[15]. 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(0.9235996615888572, 'dimensionless')>
```

[14] 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

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

```
>>> s1 = pyEQL.Solution([['Mg+2','0.3 mol/kg'],['Cl-','0.6 mol/kg']],
↪temperature='30 degC')
>>> s1.get_osmotic_coefficient()
<Quantity(0.891154788474231, 'dimensionless')>
```

get_osmotic_pressure()

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

Parameters

None

Returns

Quantity 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[16][17] :

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

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.get_osmotic_pressure()
0.0
```

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

get_pressure()

Return the hydrostatic pressure of the solution.

Returns

Quantity: The hydrostatic pressure of the solution, in atm.

get_property(*solute*, *name*)

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

Parameters

[16] 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.

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

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: The desired parameter

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 direclty 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 Salt object containing information about the parent 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_list()

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 direclty model these quantities.

The get_salt_list() method examines the ionic composition of a solution and simplifies it into a list of salts. The method retuns 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_solute(*i*)
Return the specified solute object.

get_solvent()
Return the solvent object.

get_solvent_mass()
Return the mass of the solvent.

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

Parameters

None

Returns

Quantity: the mass of the solvent, in kg

See also:

get_amount

get_temperature()
Return the temperature of the solution.

Parameters

None

Returns

Quantity: The temperature of the solution, in Kelvin.

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'

Returns

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

See also:

get_amount

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. (TODO)

get_total_moles_solute()
Return the total moles of all solute in the solution

get_transport_number(*solute*, *activity_correction=False*)
Calculate the transport number of the solute in the solution

Parameters

solute [str] String identifying the solute for which the transport number is to be calculated.

activity_correction: bool If True, the transport number will be corrected for activity following the same method used for solution conductivity. Defaults to False if omitted.

Returns

float The transport number of *solute*

Notes

Transport number is calculated according to[18] :

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

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 get_conductivity() for an explanation of this correction.

References

ion effects on membrane potential and the permselectivity of ion exchange membranes."" *Phys. Chem. Chem. Phys.* 2014, 16, 21673–21681.

get_viscosity_dynamic()
Return the dynamic (absolute) viscosity of the solution.

Calculated from the kinematic viscosity

See also:

get_viscosity_kinematic, *get_viscosity_relative*

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

get_viscosity_kinematic()
Return the kinematic viscosity of the solution.

See also:

get_viscosity_dynamic, *get_viscosity_relative*

Notes

The calculation is based on a model derived from the Eyring equation and presented in[19]

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

Where:

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

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

In which :math: *nu* 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

get_viscosity_relative()
Return the viscosity of the solution relative to that of water

This is calculated using a simplified form of the Jones-Dole equation:

$$\eta_{rel} = 1 + \sum_i B_i m_i$$

Where m is the molal concentration and B is an empirical parameter.

See <http://downloads.olisystems.com/ResourceCD/TransportProperties/Viscosity-Aqueous.pdf> <http://www.nrcresearchpress.com/doi/pdf/10.1139/v77-148> <http://apple.csgi.unifi.it/~fratini/chen/pdf/14.pdf>

get_volume()
Return the volume of the solution.

Parameters

None

Returns

Quantity: the volume of the solution, in L

get_water_activity()
Return the water activity.

Returns

Quantity : The thermodynamic activity of water in the solution.

See also:

get_osmotic_coefficient, *get_ionic_strength*, *get_salt*

[19] Geise, G. M.; Cassady, H. J.; Paul, D. R.; Logan, E.; Hickner, M. A. "Specific

Notes

Water activity is related to the osmotic coefficient in a solution containing i solutes by:[20]

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

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

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')>
```

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

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

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

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

list_solutes()
List all the solutes in the solution.

p(*solute*, *activity=True*)
Return the negative log of the activity of solute.

Generally used for expressing concentration of hydrogen ions (pH)

[20] 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.

Parameters

solute [str] String representing the formula of the solute

activity: bool, optional If False, the function will use the molar concentration rather than the activity to calculate p. Defaults to True.

Returns

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

Examples

TODO

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.

Returns

Nothing. The concentration of solute is modified.

See also:

`Solute.set_moles`

set_pressure(*pressure*)
Set the hydrostatic pressure of the solution.

Parameters

pressure [str] String representing the temperature, e.g. '25 degC'

set_temperature(*temperature*)
Set the solution temperature.

Parameters

temperature [str] String representing the temperature, e.g. '25 degC'

set_volume(*volume*)
Change the total solution volume to volume, while preserving all component concentrations

Parameters

volume [str quantity] Total volume of the solution, including the unit, e.g. '1 L'

Examples

```
>>> mysol = Solution([['Na+','2 mol/L'],['Cl-','0.01 mol/L']],volume='500 mL')
>>> print(mysol.get_volume())
0.5000883925072983 l
>>> mysol.list_concentrations()
{'H2O': '55.508435061791985 mol/kg', 'Cl-': '0.00992937605907076 mol/kg', 'Na+
↪': '2.0059345573880325 mol/kg'}
>>> mysol.set_volume('200 mL')
>>> print(mysol.get_volume())
0.2 l
>>> mysol.list_concentrations()
{'H2O': '55.50843506179199 mol/kg', 'Cl-': '0.00992937605907076 mol/kg', 'Na+
↪': '2.0059345573880325 mol/kg'}
```

CHAPTER 7

The Solute Class

pyEQL Solute class

This file contains functions and methods for managing properties of individual solutes. The Solute class contains methods for accessing ONLY those properties that DO NOT depend on solution composition. Solute properties such as activity coefficient or concentration that do depend on compsition are accessed via Solution class methods.

copyright 2013-2018 by Ryan S. Kingsbury

license LGPL, see LICENSE for more details.

class pyEQL.solute.**Solute**(*formula*, *amount*, *volume*, *solvent_mass*, *parameters={}*)

represent each chemical species as an object containing its formal charge, transport numbers, concentration, activity, etc.

Methods

add_moles(amount, volume, solvent_mass)	Increase or decrease the amount of a substance present in the solution
add_parameter(name, magnitude[, units])	Add a parameter to the parameters database for a solute
get_formal_charge()	Return the formal charge of the solute
get_molecular_weight()	Return the molecular weight of the solute
get_moles()	Return the moles of solute in the solution
get_name()	Return the name (formula) of the solute
get_parameter(parameter[, temperature, . . .])	Return the value of the parameter named 'parameter'
set_moles(amount, volume, solvent_mass)	Set the amount of a substance present in the solution

add_moles(*amount*, *volume*, *solvent_mass*)

Increase or decrease the amount of a substance present in the solution

Parameters

amount: str quantity Amount of substance to add. Must be in mass or substance units.

Negative values indicate subtraction of material.

add_parameter(*name*, *magnitude*, *units=''*, ***kwargs*)
Add a parameter to the parameters database for a solute

See pyEQL.parameters documentation for a description of the arguments

get_formal_charge()
Return the formal charge of the solute

Parameters

None

Returns

int The formal charge of the solute

get_molecular_weight()
Return the molecular weight of the solute

Parameters

None

Returns

Quantity The molecular weight of the solute, in g/mol

get_moles()
Return the moles of solute in the solution

Parameters

None

Returns

Quantity The number of moles of solute

get_name()
Return the name (formula) of the solute

Parameters

None

Returns

str The chemical formula of the solute

get_parameter(*parameter*, *temperature=None*, *pressure=None*, *ionic_strength=None*)
Return the value of the parameter named 'parameter'

set_moles(*amount*, *volume*, *solvent_mass*)
Set the amount of a substance present in the solution

Parameters

amount: str quantity Desired amount of substance. Must be greater than or equal to zero and given in mass or substance units.

CHAPTER 8

Internal Reference Documentation

8.1 Activity Correction API

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 2013-2018 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:[1]

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

[1] 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.

Returns

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

See also:

_debye_parameter_osmotic

Notes

The parameter A is equal to:[2]

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

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

Examples

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

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.

See also:

_debye_parameter_activity

[2] 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.

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:[3][4]

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

References

Examples

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

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.

See also:

_debye_parameter_osmotic

Notes

Takes the value 1.8305 cm ** 3 * kg ** 0.5 / mol ** 1.5 at 25 C. This constant is calculated according to:[5]

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

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.[6]

[3] 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.

[4] 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.

[5] 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.

[6] http://hyperphysics.phy-astr.gsu.edu/hbase/permot3.html

References

Examples

TODO

pyEQL.activity_correction.**_pitzer_B_MX**(*ionic_strength*, *alpha1*, *alpha2*, *beta0*, *beta1*, *beta2*)
Return the B_MX coefficient for the Pitzer ion interaction model.

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

Parameters

ionic_strength: number The ionic strength of the parent solution, mol/kg

alpha1, alpha2: number Coefficients for the Pitzer model, kg ** 0.5 / mol ** 0.5

beta0, beta1, beta2: number Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.

Returns

float The B_MX parameter for the Pitzer ion interaction model.

See also:

_pitzer_f1

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_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, alpha2: number Coefficients for the Pitzer model, kg ** 0.5 / mol ** 0.5

beta0, beta1, beta2: number Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.

Returns

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

References

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

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

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 intercation 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 eta_gamma in the Pitzer ion intercation 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,B_phi,C_phi: Quantity Calculated paramters for the Pitzer ion interaction model.

z_cation, z_anion: int The formal charge on the cation and anion, respectively

nu_cation, 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

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

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:[7]

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

Valid for 0.1 < I < 0.5

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

References

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:[8]

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

Valid only for I < 0.005

References

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.

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

See also:

_debye_parameter_activity

Notes

Activity coefficient is calculated according to:[9]

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

Valid for I < 0.1

References

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, 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, beta1, beta2, C_phi: number Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.

z_cation, z_anion: int The formal charge on the cation and anion, respectively

nu_cation, 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

See also:

_debye_parameter_activity, *_pitzer_B_MX*, _pitzer_B_gamma, *_pitzer_B_phi*, *_pitzer_log_gamma*

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

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

Examples

```
>>> get_activity_coefficient_pitzer(0.5*unit('mol/kg'),0.5*unit('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('mol/kg'),5.6153*unit('mol/kg'),3,
↪0.5,0.0369993,0.354664,0.0997513,-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('mol/kg'),10*unit('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('mol/kg'),5*unit('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('mol/kg'),18*unit('mol/kg'),2,0,-0.
↪01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.16241 ...
```

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, 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, beta1, beta2, 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, z_anion: int The formal charge on the cation and anion, respectively

nu_cation, 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

See also:

_debye_parameter_volume, *_pitzer_B_MX*

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.

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('mol/kg'),0.25*unit('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('mol/kg'),1.0*unit('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('mol/kg'),10.0*unit('mol/kg'),2,0,0.
↪000001742,0.0002926,0,0.000000424,46.9,1,-1,1,1,b=1.2)
50.286...
```

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

```
>>> get_apparent_volume_pitzer(20.0*unit('mol/kg'),20.0*unit('mol/kg'),2,0,0.
↪000001742,0.0002926,0,0.000000424,46.9,1,-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('mol/kg'),0.8*unit('mol/kg'),2,0,0.
↪000024693,0.00003169,0,-0.000004068,-2.426,1,-1,1,1,b=1.2)
0.22595 ...
```

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, 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, beta1, beta2, C_phi Coefficients for the Pitzer model. These ion-interaction parameters are specific to each salt system.

z_cation, z_anion: int The formal charge on the cation and anion, respectively

nu_cation, 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

See also:

_debye_parameter_activity, *_pitzer_B_MX*, _pitzer_B_gamma, *_pitzer_B_phi*, *_pitzer_log_gamma*

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

Examples

Experimental value according to Beyer and Stieger reference is 1.3550

```
>>> get_osmotic_coefficient_pitzer(10.175*unit('mol/kg'),10.175*unit('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('mol/kg'),5.6153*unit('mol/kg'),3,
↪0.5,0.0369993,0.354664,0.0997513,-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('mol/kg'),10*unit('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('mol/kg'),5*unit('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('mol/kg'),18*unit('mol/kg'),2,0,-0.
↪01709,0.09198,0,0.000419,1,-1,1,1,b=1.2)
0.5556 ...
```

8.2 Water Properties API

pyEQL water properties library

This file contains functions for retrieving various physical properties of water substance

copyright 2013-2018 by Ryan S. Kingsbury

license LGPL, see LICENSE for more details.

pyEQL.water_properties.**water_density**(*temperature=<Quantity(25, 'degC')>*, *pressure=<Quantity(1, 'atmosphere')>*)

Return the density of water in kg/m3 at the specified temperature and pressure.

Parameters

temperature [float or int, optional] The temperature in Celsius. Defaults to 25 degrees if not specified.

pressure [float or int, optional] The ambient pressure of the solution in Pascals (N/m2). Defaults to atmospheric pressure (101325 Pa) if not specified.

Returns

float The density of water in kg/m3.

Notes

Based on the following empirical equation reported in[10]

$$\rho_W = 999.65 + 0.20438T - 6.1744e - 2T^{1.5}$$

Where T is the temperature in Celsius.

Examples

```
>>> water_density(25*unit('degC'))
<Quantity(997.0415, 'kilogram / meter ** 3')>
```

pyEQL.water_properties.**water_dielectric_constant**(*temperature=<Quantity(25, 'degC')>*)

Return the dielectric constant of water at the specified temperature.

Parameters

temperature [Quantity, optional] The temperature. Defaults to 25 degC if omitted.

Returns

float The dielectric constant (or permittivity) of water relative to the permittivity of a vacuum. Dimensionless.

Notes

This function implements a quadratic fit of measured permittivity data as reported in the CRC Handbook[11]. The parameters given are valid over the range 273 K to 372 K. Permittivity should not be extrapolated beyond this range.

$$\epsilon(T) = a + bT + cT^2$$

[10] Sohnel, O and Novotny, P. *Densities of Aqueous Solutions of Inorganic Substances.* Elsevier Science, Amsterdam, 1985.
[11] "Permittivity (Dielectric Constant) of Liquids." *CRC Handbook of Chemistry and Physics*, 92nd ed, pp 6-187 - 6-208.

References

Examples

```
>>> water_dielectric_constant(unit('20 degC'))
80.15060...
```

Display an error if 'temperature' is outside the valid range

```
>>> water_dielectric_constant(-5*unit('degC'))
```

pyEQL.water_properties.**water_specific_weight**(*temperature=<Quantity(25, 'degC')>*, *pressure=<Quantity(1, 'atmosphere')>*)

Return the specific weight of water in N/m3 at the specified temperature and pressure.

Parameters

temperature [Quantity, optional] The temperature. Defaults to 25 degC if omitted.

pressure [Quantity, optional] The ambient pressure of the solution. Defaults to atmospheric pressure (1 atm) if omitted.

Returns

Quantity The specific weight of water in N/m3.

See also:

water_density

Examples

```
>>> water_specific_weight()
<Quantity(9777.637025975, 'newton / meter ** 3')>
```

pyEQL.water_properties.**water_viscosity_dynamic**(*temperature=<Quantity(25, 'degC')>*, *pressure=<Quantity(1, 'atmosphere')>*)

Return the dynamic (absolute) viscosity of water in N-s/m2 = Pa-s = kg/m-s at the specified temperature.

Parameters

temperature [Quantity, optional] The temperature. Defaults to 25 degC if omitted.

pressure [Quantity, optional] The ambient pressure of the solution. Defaults to atmospheric pressure (1 atm) if omitted.

Returns

Quantity The dynamic (absolute) viscosity of water in N-s/m2 = Pa-s = kg/m-s

Notes

Implements the international equation for viscosity of water as specified by NIST[12]

Valid for 273 < temperature < 1073 K and 0 < pressure < 100,000,000 Pa

[12] Sengers, J.V. "Representative Equations for the Viscosity of Water Substance." *J. Phys. Chem. Ref. Data* 13(1), 1984.http://www.nist.gov/data/PDFfiles/jpcrd243.pdf

References

Examples

```
>>> water_viscosity_dynamic(20*unit('degC'))
<Quantity(0.000998588610804179, 'kilogram / meter / second')>
>>> water_viscosity_dynamic(unit('100 degC'),unit('25 MPa'))
<Quantity(0.00028165034364318573, 'kilogram / meter / second')>
>>> water_viscosity_dynamic(25*unit('degC'),0.1*unit('MPa'))
<Quantity(0.0008872817880143659, 'kilogram / meter / second')>
```

#TODO - check these again after I implement pressure-dependent density function

pyEQL.water_properties.**water_viscosity_kinematic**(*temperature=<Quantity(25, 'degC')>*, *pressure=<Quantity(1, 'atmosphere')>*)

Return the kinematic viscosity of water in m2/s = Stokes at the specified temperature.

Parameters

temperature [Quantity, optional] The temperature. Defaults to 25 degC if omitted.

pressure [Quantity, optional] The ambient pressure of the solution. Defaults to atmospheric pressure (1 atm) if omitted.

Returns

Quantity The kinematic viscosity of water in Stokes (m2/s)

See also:

water_viscosity_dynamic, *water_density*

Examples

```
>>> water_viscosity_kinematic()
<Quantity(8.899146003595295e-07, 'meter ** 2 / second')>
```

CHAPTER 9

Functions Module

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

copyright 2013-2018 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

Solution A pyEQL Solution object.

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[3]

[1] Millero, Frank J. "The composition of Standard Seawater and the definition of the Reference-Composition Salinity Scale." *Deep-sea Research. Part I* 55(1), 2008, 50-72.

[2] Metcalf & Eddy, Inc. et al. *Wastewater Engineering: Treatment and Resource Recovery*, 5th Ed. McGraw-Hill, 2013.

[3] https://en.wikipedia.org/wiki/Saline_(medicine)

- 'normal saline' or 'NS' - normal saline solution used in medicine[4]
- 'Ringers lacatate' or 'RL' - Ringer's lactate solution used in medicine[5]

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

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

See also:

`get_salt`

Notes

The general equation representing the equilibrium between an external electrolyte solution and an ion-exchange medium containing fixed charges is:[6]

$$\frac{a_-}{\bar{a}_-}^{\frac{1}{z_-}} \frac{\bar{a}_+}{a_+}^{\frac{1}{z_+}} = exp(\frac{\Delta\pi\bar{V}}{RTz_+\nu_+})$$

Where subscripts $+$ and $-$ indicate the cation and anion, respectively, the overbar indicates the membrane phase, a represents activity, z represents charge, ν represents the stoichiometric coefficient, V represents the partial molar volume of the salt, and $\Delta\pi$ is the difference in osmotic pressure between the membrane and the solution phase.

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.

[4] https://en.wikipedia.org/wiki/Ringer%27s_lactate_solution

[5] 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

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

References

Examples

TODO

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

Return the ideal mixing entropy associated with mixing two solutions

Parameters

Solution1, Solution2 [Solution objects] The two solutions to be mixed.

Returns

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

Notes

The ideal entropy of mixing is calculated as follows:[7]

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

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

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

References

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

Return the Gibbs energy change associated with mixing two solutions.

Parameters

Solution1, Solution2 [Solution objects] The two solutions to be mixed.

Returns

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

Notes

The Gibbs energy of mixing is calculated as follows: [#]_

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

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

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

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

References

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, Solution2 [Solution objects] The two solutions to be mixed.

Returns

Solution A Solution object representing the mixed solution.

Python Module Index

p

Index

Symbols

A

C

D

E

G

H

I

L

M

P

S

W