chemicals Documentation

Release 1.1.4

Caleb Bell and contributors

Jun 04, 2023

TUTORIAL

CHAPTER

ONE

KEY FEATURES & CAPABILITIES

The chemicals library features an extensive compilation of pure component chemical data that can serve engineers, scientists, technicians, and anyone working with chemicals. The chemicals library facilitates the retrieval and calculation of:

- Chemical constants including formula, molecular weight, normal boiling and melting points, triple point, heat of formation, absolute entropy of formation, heat of fusion, similarity variable, dipole moment, acentric factor, etc.
- Assorted information of safety and toxicity of chemicals.
- Methods (and their respective coefficients) for the calculation of temperature and pressure dependent chemical properties including vapor pressure, heat capacity, molar volume, thermal conductivity, surface tension, dynamic viscosity, heat of vaporization, relative permittivity, etc.
- Methods to solve thermodynamic phase equilibrium, including flash routines, vapor-liquid equilibrium constant correlations, and both numerical and analytical solutions for the Rachford Rice and Li-Johns-Ahmadi equations. Rachford Rice solutions for systems of 3 or more phases are also available.

Data for over 20,000 chemicals are made available as local databanks in this library. All databanks are loaded on-demand, saving loading time and RAM. For example, if only data on the normal boiling point is required, the chemicals library will only load normal boiling point datasets. This on-demand loading feature makes the chemicals library an attractive dependence for software modeling chemical processes. In fact, The Biorefinery Simulation and Techno-Economic Analysis Modules (BioSTEAM) is reliant on the chemicals library for the simulation of unit operations.

The chemicals library also supports integration with Numba, a powerful accelerator that works well with NumPy; Pint Quantity objects to keep track of units of measure; and NumPy vectorized functions.

If you need to know something about a chemical, give chemicals a try.

1.1 Chemicals tutorial

1.1.1 Importing

Chemicals can be imported as a standalone library, or all of its functions and classes may be imported with star imports:

```
>>> import numpy as np
>>> import chemicals # Good practice
>>> from chemicals import * # Bad practice but convenient
```

All functions are available from either the main chemicals module or the submodule; i.e. both chemicals.Antoine and chemicals.vapor_pressure.Antoine are valid ways of accessing a function.

1.1.2 Design philosophy

Like all libraries, this was developed to scratch my own itches.

The bulk of this library's API is considered quite stable; enhancements to functions and classes will still happen, and default methods when using a generic correlation interface may change to newer and more accurate correlations as they are published and reviewed.

All functions are designed to accept inputs in base SI units. However, any set of consistent units given to a function will return a consistent result. The user is directed to unit conversion libraries such as pint to perform unit conversions if they prefer not to work in SI units. The tutorial for using it with chemicals is at chemicals.units.

There are two ways to use numpy arrays with chemicals. The easiest way to use numpy is a *vectorized* module, which wraps all of the chemicals functions with np.vectorize. Instead of importing from chemicals, the user can import from *chemicals.vectorized*:

```
>>> from chemicals.vectorized import *
>>> Antoine(np.linspace(100, 200, 5), A=8.95894, B=510.595, C=-15.95)
array([7.65674361e+02, 1.89116754e+04, 1.41237759e+05, 5.60609191e+05,
       1.53010431e+06])
```

Inputs do not need to be numpy arrays; they can be any iterable:

```
>>> import chemicals.vectorized
>>> chemicals.vectorized.Tc(['108-88-3', '7732-18-5'])
array([591.75 , 647.096])
```

It is possible to switch back and forth between the namespaces with a subsequent import:

```
>>> from chemicals import *
```

The second way is Numba. This optional dependency provides the speed you expect from NumPy arrays - or better. In some cases, much better. The tutorial for using it is at *chemicals.numba*, but in general use it the same way but with a different import.

```
>>> import chemicals.numba_vectorized
```

Note that numba can also be used to speed up scalar calculations without numpy.

```
>>> import chemicals.numba
```

1.1.3 Working with Elements

Chemicals contains a periodic table.

```
>>> from chemicals import *
>>> periodic_table.Na
<Element Sodium (Na), number 11, MW=22.98977>
>>> periodic_table.U.MW
238.02891
>>> periodic_table['Th'].CAS
'7440-29-1'
>>> periodic_table.lead.protons
82
```

(continues on next page)

(continued from previous page)

```
>>> periodic_table['7440-57-5'].symbol
'Au'
>>> len(periodic_table)
118
>>> 'gold' in periodic_table
True
>>> periodic_table.He.protons, periodic_table.He.neutrons, periodic_table.He.electrons #␣
↪Standard number of protons, neutrons, electrons
(2, 2, 2)
>>> periodic_table.He.phase # Phase of the element in the standard state
'g'
>>> periodic_table.He.Hf # Heat of formation in standard state in J/mol - by definition 0
0.0
>>> periodic_table.He.S0 # Absolute entropy (J/(mol*K) in standard state - non-zero)
126.2
>>> periodic_table.Kr.block, periodic_table.Kr.period, periodic_table.Kr.group
('p', 4, 18)
>>> periodic_table.Rn.InChI
'Rn'
>>> periodic_table.Rn.smiles
'[Rn]'
>>> periodic_table.Pu.number
94
>>> periodic_table.Pu.PubChem
23940
>>> periodic_table.Bi.InChI_key
'JCXGWMGPZLAOME-UHFFFAOYSA-N'
```

The periodic table is a singleton of the periodic table class *PeriodicTable*. Each attribute accessed is a reference to an element object *Element*. The elements are the basic building blocks of every chemical.

1.1.4 Working with Chemical Identifiers

Chemicals comes with a large library of chemical identifiers. Chemicals has various ways of searching through its database. There are a number of different support chemical identifiers as well.

CAS numbers - These are the primary identifiers in Chemicals. A CAS number uniquely identifies a chemical molecule. 7732-18-5 is the CAS number for water. Sometimes, it also identifies the phase of the chemical. 7440-44-0 is the CAS number for carbon in general, but 7782-42-5 is the CAS number for graphite and 7782-40-3 is the CAS number for diamond. Note that because these are assigned by people, mistakes are made and often multiple CAS numbers point to the same compound. Common Chemistry lists 57 "retired" CAS numbers which point to the element carbon. The CAS numbers in Chemicals come mostly from PubChem as there was no Common Chemistry project back then.

PubChem IDs - These are the identifiers for each compound in the PubChem database. Most of the metadata in Chemicals came from PubChem. 962 is the Pubchem identifier for water. Each entry in PubChem comes with a structure. Sometimes structures are found to be duplicates of each other and entries are merged; these identifiers are assigned automatically by the NIH.

Smiles - These are actual chemicals structures, rendered into easily readable text. Multiple smiles strings can represent the same compound; they are not unique. Both "C(=O)=O" and "O=C=O" are valid SMILES strings for identifying CO2. Programs like rdkit can create a computational representation of the molecule from a SMILES string. To solve this duplication issue, a concept of a canonical SMILES string was developed which is supposed to be unique, but in

general is not reliable at all and only consistent within the same molecular modeling software. There is in general no organization which controls this format, but a there is an effort in the open source community to standardize the format called opensmiles

Chemical Formula - These are what every student is taught in chemistry class. H2O is the formula for water. Is OH2 also a valid formula? Yes. There is a convention called the Hill convention (implemented in chemicals as *atoms_to_Hill()* which specified the H2O is how the formula should be written. Not all formulas, especially inorganic formulas or older formulas, follow this convention. Formulas are in general NOT unique. Even simple formulas which seem like there should only be one compound with that formula are often duplicated; carbonic acid and performic acid both have the formula "CH2O3". Searching Chemical's databases with a formula is a common mistake by users. While you can do it and you may get a match, there is no guarantee the match you wanted was found. The following snippet of code counts the number of compounds with the same formula as asprin; illustrating why searching by formula is a bad idea.

```
>>> from chemicals.identifiers import pubchem_db
>>> len(list(i for i in pubchem_db if i.formula == 'C9H8O4'))
20
```

Chemical name - Anyone can call a chemical by any name, so predictably names are a mess. A large number of names were retrieved from PubChem, and form the basis for searches by name in Chemicals. Only one chemical hit will be found for each name search. There is an effort by IUPAC to systematically generate names for each chemical structure, called OPSIN. Most chemicals in Chemicals have a correct, associated IUPAC name retrieved from PubChem. There are in the range of a million names that can be looked by in Chemicals.

InChI - Short for the IUPAC International Chemical Identifier, these are programmatically derived strings which represent a compound. A non-profit was established to maintain a software package to manage this format; it is not like SMILES where lots of software implement the format. There contain all the information required to form a structure. There is a variant which is truly unique per compound; this is what is in Chemicals. They have more features than SMILES strings. "C6H14/c1-3-5-6-4-2/h3-6H2,1-2H3" is a sample string, for n-hexane. This is the best possible type of an identifier for a chemical. These can get to be quite long for complex structures.

InChI key - A 27-character hash of the unique InChI identifier. These are also in Chemicals and generated by the same InChI software. These were intended to be unique, and easy to search for as search engines don't search for InChI strings well. Some collisions have been detected. 'VLKZOEOYAKHREP-UHFFFAOYSA-N' is the InChI key for n-hexane as an example.

The main interface for looking up a chemical from one of these identifying markers is *search_chemical()*. The search can be performed with any of the following input forms:

- Name, in IUPAC form or common form or a synonym registered in PubChem
- InChI name, prefixed by 'InChI=1S/' or 'InChI=1/'
- InChI key, prefixed by 'InChIKey='
- PubChem CID, prefixed by 'PubChem='
- SMILES (prefix with 'SMILES=' to ensure smiles parsing; ex. 'C' will return Carbon as it is an element whereas the SMILES interpretation for 'C' is methane)
- CAS number (obsolete numbers may point to the current number)

If the input is an ID representing an element, the following additional inputs may be specified as

- Atomic symbol (ex 'Na')
- Atomic number (as a string)

Some sample queries illustrating the topic:

```
>>> search_chemical('water')
<ChemicalMetadata, name=water, formula=H2O, smiles=O, MW=18.0153>
>>> search_chemical('InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3')
<ChemicalMetadata, name=ethanol, formula=C2H6O, smiles=CCO, MW=46.0684>
>>> search_chemical('CCCCCCCCCC')
<ChemicalMetadata, name=DECANE, formula=C10H22, smiles=CCCCCCCCCC, MW=142.286>
>>> search_chemical('InChIKey=LFQSCWFLJHTTHZ-UHFFFAOYSA-N')
<ChemicalMetadata, name=ethanol, formula=C2H6O, smiles=CCO, MW=46.0684>
>>> search_chemical('pubchem=702')
<ChemicalMetadata, name=ethanol, formula=C2H6O, smiles=CCO, MW=46.0684>
>>> search_chemical('O') # only elements can be specified by symbol
<ChemicalMetadata, name=oxygen, formula=O, smiles=[O], MW=15.9994>
```

Each of those queries returns a *ChemicalMetadata* object. The object holds the chemical metadata. It is an almost unbearable task to assemble a chemical property database. Making a database of chemical metadata is only slightly easier. The chemical metadata database doesn't have any information whatsoever about about any chemical properties; only information about the chemical structure and those identifiers mentioned above. Each of those identifiers is an attribute of the returned object.

```
>>> water = search_chemical('water')
>>> (water.pubchemid, water.formula, water.smiles, water.InChI, water.InChI_key, water.
↪CASs)
(962, 'H2O', 'O', 'H2O/h1H2', 'XLYOFNOQVPJJNP-UHFFFAOYSA-N', '7732-18-5')
>>> water.common_name, water.iupac_name, len(water.synonyms)
('water', 'oxidane', 89)
```

1.2 Acentric Factor (chemicals.acentric)

This module contains a lookup function, a definition function, and correlations for a chemical's *acentric* factor, normally given the variable ω.

A similar variable called the *stiel polar factor* can be calculated from its definition as well.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.2.1 Lookup Functions

chemicals.acentric.**omega**(*CASRN*, *method=None*)

Retrieve a chemical's acentric factor, *omega*.

Automatically select a method to use if no method is provided; returns None if the data is not available.

$$\omega \equiv -\log_{10}\left[\lim_{T/T_c=0.7}(P^{sat}/P_c)\right] - 1.0$$

Parameters

CASRN [str] CASRN [-]

Returns

omega [float] Acentric factor of compound

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'HEOS', 'PSRK', 'PD', or 'YAWS'. All valid values are also held in the variable *omega_all_methods*.

See also:

omega_methods

Notes

A total of four sources are available for this function. They are:

- 'PSRK', a compillation of experimental and estimated data published in the Appendix of [2], the fourth revision of the PSRK model.
- 'PD', an older compillation of data published in (Passut & Danner, 1973) [3].
- 'YAWS', a large compillation of data from a variety of sources; no data points are sourced in the work of [4].
- 'ACENTRIC_DEFINITION', the precalculated results using the *VaporPressure* object of Thermo and the critical properties of *chemicals*.
- 'HEOS', a series of values from the NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids (and other high-precision fundamental equations of state)

References

[1], [2], [3], [4], [5]

Examples

```
>>> omega(CASRN='64-17-5')
0.646
```

chemicals.acentric.**omega_methods**(*CASRN*)

Return all methods available for obtaining omega for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain omega with the given inputs.

See also:

omega

chemicals.acentric.**omega_all_methods = ('HEOS', 'PSRK', 'PD', 'YAWS', 'ACENTRIC_DEFINITION')**
Tuple of method name keys. See the *omega* for the actual references

1.2.2 Definitions

chemicals.acentric.**omega_definition**(*Psat*, *Pc*)
Returns the acentric factor of a fluid according to its fundamental definition using the vapor pressure at a reduced temperature of 0.7Tc.

$$\omega \equiv -\log_{10}\left[\lim_{T/T_c=0.7}(P^{sat}/P_c)\right] - 1.0$$

Parameters

Psat [float] Vapor pressure of the fluid at a reduced temperature of 0.7 [Pa]

Pc [float] Critical pressure of the fluid [Pa]

Returns

omega [float] Acentric factor of the fluid [-]

References

[1]

Examples

Water

```
>>> omega_definition(999542, 22048320.0)
0.3435744558761711
```

chemicals.acentric.**Stiel_polar_factor**(*Psat*, *Pc*, *omega*)
This function handles the calculation of a chemical's Stiel Polar factor, directly through the definition of Stiel-polar factor. Requires the vapor pressure *Psat* at a reduced temperature of 0.6, the critical pressure *Pc*, and the acentric factor *omega*.

$$x = \log_{10} P_r|_{T_r=0.6} + 1.70\omega + 1.552$$

Parameters

Psat [float] Vapor pressure of fluid at a reduced temperature of 0.6 [Pa]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor of the fluid [-]

Returns

factor [float] Stiel polar factor of compound, [-]

Notes

A few points have also been published in [2], which may be used for comparison. Currently this is only used for a surface tension correlation.

References

[1], [2]

Examples

Calculating the factor for water:

```
>>> Stiel_polar_factor(Psat=169745, Pc=22048321.0, omega=0.344)
0.02322146744772713
```

1.2.3 Correlations

chemicals.acentric.**LK_omega**(*Tb*, *Tc*, *Pc*)

Estimates the acentric factor of a fluid using a correlation in [1].

$$\omega = \frac{\ln P_{br}^{sat} - 5.92714 + 6.09648/T_{br} + 1.28862\ln T_{br} - 0.169347T_{br}^6}{15.2518 - 15.6875/T_{br} - 13.4721\ln T_{br} + 0.43577T_{br}^6}$$

Parameters

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

Returns

omega [float] Acentric factor of the fluid [-]

Notes

The units of the above equation are atmosphere and Kelvin; values are converted internally.

References

[1]

Examples

Isopropylbenzene, from Reid (1987).

```
>>> LK_omega(425.6, 631.1, 32.1E5)
0.32544249926397856
```

1.3 Air: Fundamental Equation of State for Air (chemicals.air)

This module contains various thermodynamic functions for air and humid air.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.3.1 Dry Air Basic Solvers

chemicals.air.**lemmon2000_rho**(*T*, *P*)

Calculate the density of air according to the Lemmon (2000) [1] given a temperature T and pressure P.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Molar density of air, [mol/m^3]

Notes

This solution is iterative due to the nature of the equation. This solver has been tested only for gas solutions.

References

[1]

Examples

```
>>> lemmon2000_rho(T=300.0, P=1e6)
402.046613509
```

2 GPa and 2000 K are suggested as upper limits of [1] although there are no hardcoded limits for temperature and pressure.

```
>>> lemmon2000_rho(T=2000.0, P=2e9)
32892.9327834
```

chemicals.air.**lemmon2000_P**(*T*, *rho*)

Calculate the pressure of air according to the (2000) given a temperature *T* and molar density *rho*.

Parameters

T [float] Temperature, [K]

rho [float] Molar density of air, [mol/m^3]

Returns

P [float] Pressure, [Pa]

Notes

Helmholtz equations of state are explicit with inputs of temperature and density, so this is a direct calculation with no iteration required.

References

[1]

Examples

```
>>> lemmon2000_P(330.0, lemmon2000_rho(T=330.0, P=8e5))
8e5
>>> lemmon2000_P(823.0, 40)
273973.0024911
```

chemicals.air.**lemmon2000_T**(*P*, *rho*)

Calculate the temperature of air according to the Lemmon (2000) [1] given a pressure *P* and molar density *rho* .

Parameters

P [float] Pressure, [Pa]

rho [float] Molar density of air, [mol/m^3]

Returns

T [float] Temperature, [K]

Notes

This solution is iterative due to the nature of the equation. This solver has been tested only for gas solutions.

References

[1]

Examples

```
>>> lemmon2000_T(P=1e5, rho=20.0)
601.1393854499
```

1.3.2 Dry Air Bubble/Dew Points

chemicals.air.**lemmon2000_air_P_dew**(*T*)

Calculates the dew pressure of standard dry air according to Lemmon (2000).

$$\ln\left(\frac{P_{dew}}{P_j}\right) = \left(\frac{T_j}{T}\right)\sum_i^8 N_i\theta^{i/2}$$

Parameters

T [float] Temperature, [K]

Returns

P_dew [float] Dew pressure, [Pa]

Notes

The stated range of this ancillary equation is 59.75 K <= T <= 132.6312 K.

Examples

```
>>> lemmon2000_air_P_dew(100.0)
567424.1338937
```

chemicals.air.**lemmon2000_air_P_bubble**(*T*)

Calculates the bubble pressure of standard dry air according to Lemmon (2000).

$$\ln\left(\frac{P_{bubble}}{P_j}\right) = \left(\frac{T_j}{T}\right)\sum_i^8 N_i\theta^{i/2}$$

Parameters

T [float] Temperature, [K]

Returns

P_bubble [float] Bubble pressure, [Pa]

Notes

The stated range of this ancillary equation is 59.75 K <= T <= 132.6312 K.

Examples

```
>>> lemmon2000_air_P_bubble(100.0)
663128.589440
```

chemicals.air.**lemmon2000_air_rho_dew**(*T*)

Calculates the dew molar density of standard dry air according to Lemmon (2000).

$$\ln\left(\frac{\rho_{dew}}{\rho_j}\right) = N_1\theta^{0.41} + N_2\theta + N_3\theta^{2.8} + N_4\theta^{6.5}$$

Parameters

T [float] Temperature, [K]

Returns

rho_dew [float] Dew point molar density, [mol/m^3]

Notes

The stated range of this ancillary equation is 59.75 K <= T <= 132.6312 K.

Examples

```
>>> lemmon2000_air_rho_dew(100.0)
785.7863223794999
```

chemicals.air.**lemmon2000_air_rho_bubble**(*T*)

Calculates the bubble molar density of standard dry air according to Lemmon (2000).

$$\left(\frac{\rho_{bubble}}{rho_j} - 1\right) = N_1\theta^{0.65} + N_2\theta^{0.85} + N_3\theta^{0.95} + N_4\theta^{1.1} + N_5 \ln\frac{T}{T_j}$$

Parameters

T [float] Temperature, [K]

Returns

rho_bubble [float] bubble point molar density, [mol/m^3]

Notes

The stated range of this ancillary equation is 59.75 K <= T <= 132.6312 K.

Examples

```
>>> lemmon2000_air_rho_bubble(100.0)
26530.979020427476
```

1.3.3 Dry Air Constants

chemicals.air.**lemmon2000_air_T_reducing = 132.6312**
Reducing temperature in K for the Lemmon (2000) EOS for dry air

chemicals.air.**lemmon2000_air_rho_reducing = 10447.7**
Reducing molar density in mol/m^3 for the Lemmon (2000) EOS for dry air

chemicals.air.**lemmon2000_air_P_reducing = 3785020.0**
Reducing pressure in Pa for the Lemmon (2000) EOS for dry air

chemicals.air.**lemmon2000_air_MW = 28.9586**
Molecular weight of air in g/mol for the Lemmon (2000) EOS for dry air

chemicals.air.**lemmon2000_air_R = 8.31451**
Molar gas constant in Jlemmon2000_air_R/(mol*K) used in the the Lemmon (2000) EOS for dry air

chemicals.air.**lemmon2000_air_T_max = 2000.0**
Maximum temperature in K valid for the Lemmon (2000) EOS for dry air

chemicals.air.**lemmon2000_air_P_max = 2000000000.0**
Maximum pressure in Pa valid for the Lemmon (2000) EOS for dry air

1.3.4 Dry Air Ideal Gas Terms

chemicals.air.**lemmon2000_air_A0**(*tau*, *delta*)
Calculates the ideal gas Helmholtz energy of air according to Lemmon (2000).

$$\phi^\circ = \ln\delta + \sum_{i=1}^{5} N_i \tau^{i-4} + N_6\tau^{1.5} + N_7 \ln\tau + N_8 \ln[1-\exp(-N_{11}\tau)] + N_9 \ln[1-\exp(-N_{12}\tau)] + N_{10} \ln[2/3+\exp(N_{13}\tau)]$$

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

A0 [float] Ideal gas dimensionless Helmholtz energy A0/(RT) [-]

Notes

The coefficients are as follows:

Ns = [0.605719400E-7, -0.210274769E-4, -0.158860716E-3, -13.841928076, 17.275266575, -0.195363420E-3, 2.490888032, 0.791309509, 0.212236768, -0.197938904, 25.36365, 16.90741, 37.31279]

Examples

```
>>> lemmon2000_air_A0(132.6312/200.0, 13000/10447.7)
-14.65173785639
```

chemicals.air.**lemmon2000_air_dA0_dtau**(*tau*, *delta*)

Calculates the first temperature derivative of ideal gas Helmholtz energy of air according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

dA0_dtau [float] First derivative of *A0/(RT)* Ideal gas dimensionless Helmholtz energy with respect to *tau* [-]

Examples

```
>>> lemmon2000_air_dA0_dtau(132.6312/200.0, 13000/10447.7)
3.749095669249
```

chemicals.air.**lemmon2000_air_d2A0_dtau2**(*tau*, *delta*)

Calculates the second temperature derivative of ideal gas Helmholtz energy of air according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (126.192 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d2A0_dtau2 [float] Second derivative of *A0/(RT)* Ideal gas dimensionless Helmholtz energy with respect to *tau* [-]

Examples

```
>>> lemmon2000_air_d2A0_dtau2(132.6312/200.0, 13000/10447.7)
-5.66675499015
```

chemicals.air.**lemmon2000_air_d3A0_dtau3**(*tau*, *delta*)

Calculates the third temperature derivative of ideal gas Helmholtz energy of air according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (126.192 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d3A0_dtau3 [float] Third derivative of *A0/(RT)* Ideal gas dimensionless Helmholtz energy with respect to *tau* [-]

Examples

```
>>> lemmon2000_air_d3A0_dtau3(132.6312/200.0, 13000/10447.7)
17.10538866838
```

chemicals.air.**lemmon2000_air_d4A0_dtau4**(*tau*, *delta*)
Calculates the fourth temperature derivative of ideal gas Helmholtz energy of air according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d4A0_dtau4 [float] Fourth derivative of *A0/(RT)* Ideal gas dimensionless Helmholtz energy with respect to *tau* [-]

Examples

```
>>> lemmon2000_air_d4A0_dtau4(126.192/200.0, 13000/10447.7)
-94.815532727
```

1.3.5 Dry Air Residual Terms

chemicals.air.**lemmon2000_air_Ar**(*tau*, *delta*)
Calculates the residual Helmholtz energy of air according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

Ar [float] Residual dimensionless Helmholtz energy Ar/(RT) [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt and many multiplies/adds.

Examples

```
>>> lemmon2000_air_Ar(132.6312/200.0, 13000/10447.7)
-0.34683017661
>>> lemmon2000_air_Ar(0.36842, 0.15880050154579475)
0.0047988122806
```

chemicals.air.**lemmon2000_air_dAr_dtau**(*tau*, *delta*)
Calculates the first derivative of residual Helmholtz energy of air with respect to tau according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

dAr_dtau [float] First derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to tau, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 1 divisions and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_dAr_dtau(132.6312/200.0, 13000/10447.7)
-1.8112257495223263
```

chemicals.air.**lemmon2000_air_d2Ar_dtau2**(*tau*, *delta*)

Calculates the second derivative of residual Helmholtz energy of air with respect to tau according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d2Ar_dtau2 [float] Second derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to tau, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 2 divisions and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d2Ar_dtau2(132.6312/200.0, 13000/10447.7)
-0.7632109061747
```

chemicals.air.**lemmon2000_air_d3Ar_dtau3**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of air with respect to tau according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d3Ar_dtau3 [float] Third derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to tau, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 4 divisions and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d3Ar_dtau3(132.6312/200.0, 13000/10447.7)
0.27922007457420
```

chemicals.air.**lemmon2000_air_d4Ar_dtau4**(*tau*, *delta*)

Calculates the fourth derivative of residual Helmholtz energy of air with respect to tau according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d4Ar_dtau4 [float] Fourth derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to tau, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 4 divisions and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d4Ar_dtau4(132.6312/200.0, 13000/10447.7)
-8.197368061417
```

chemicals.air.**lemmon2000_air_dAr_ddelta**(*tau*, *delta*)

Calculates the first derivative of residual Helmholtz energy of air with respect to delta according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

dAr_ddelta [float] First derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to delta, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_dAr_ddelta(132.6312/200.0, 13000/10447.7)
-0.1367917666005
```

chemicals.air.**lemmon2000_air_d2Ar_ddelta2**(*tau*, *delta*)

Calculates the second derivative of residual Helmholtz energy of air with respect to delta according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d2Ar_ddelta2 [float] Second derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to delta, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d2Ar_ddelta2(132.6312/200.0, 13000/10447.7)
0.27027259528316
```

chemicals.air.**lemmon2000_air_d3Ar_ddelta3**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of air with respect to delta according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d3Ar_ddelta3 [float] Third derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to delta, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d3Ar_ddelta3(132.6312/200.0, 13000/10447.7)
0.1849386546766
```

chemicals.air.**lemmon2000_air_d4Ar_ddelta4**(*tau*, *delta*)

Calculates the fourth derivative of residual Helmholtz energy of air with respect to delta according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d4Ar_ddelta4 [float] Fourth derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to delta, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d4Ar_ddelta4(132.6312/200.0, 13000/10447.7)
0.37902213262258
```

chemicals.air.**lemmon2000_air_d2Ar_ddeltadtau**(*tau*, *delta*)

Calculates the second derivative of residual Helmholtz energy of air with respect to *delta* and *tau* according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d2Ar_ddeltadtau [float] Second derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to *delta* and *tau*, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d2Ar_ddeltadtau(132.6312/200.0, 13000/10447.7)
-1.359976184125
```

chemicals.air.**lemmon2000_air_d3Ar_ddeltadtau2**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of air with respect to *delta* once and *tau* twice according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d3Ar_ddeltadtau2 [float] Third derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to *delta* once and *tau* twice, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 3 divisions, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d3Ar_ddeltadtau2(132.6312/200.0, 13000/10447.7)
-0.19089212184849
```

chemicals.air.**lemmon2000_air_d3Ar_ddelta2dtau**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of air with respect to *delta* twice and *tau* once according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d3Ar_ddelta2dtau [float] Third derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to *delta* twice and *once* twice, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 3 divisions, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d3Ar_ddelta2dtau(132.6312/200.0, 13000/10447.7)
0.01441788198940
```

chemicals.air.**lemmon2000_air_d4Ar_ddelta2dtau2**(*tau*, *delta*)

Calculates the fourth derivative of residual Helmholtz energy of air with respect to *delta* twice and *tau* twice according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d4Ar_ddelta2dtau2 [float] Fourth derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to *delta* twice and *tau* twice, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 2 divisions, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d4Ar_ddelta2dtau2(132.6312/200.0, 13000/10447.7)
0.1196873112730
```

chemicals.air.**lemmon2000_air_d4Ar_ddeltadtau3**(*tau*, *delta*)

Calculates the fourth derivative of residual Helmholtz energy of air with respect to *delta* once and *tau* thrice according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d4Ar_ddeltadtau3 [float] Fourth derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to *delta* once and *tau* thrice, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 1 division, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d4Ar_ddeltadtau3(132.6312/200.0, 13000/10447.7)
2.077739387492
```

chemicals.air.**lemmon2000_air_d4Ar_ddelta3dtau**(*tau*, *delta*)

Calculates the fourth derivative of residual Helmholtz energy of air with respect to *delta* thrice and *tau* once according to Lemmon (2000).

Parameters

tau [float] Dimensionless temperature, (132.6312 K)/T [-]

delta [float] Dimensionless density, rho/(10447.7 mol/m^3), [-]

Returns

d4Ar_ddelta3dtau [float] Fourth derivative of residual dimensionless Helmholtz energy Ar/(RT) with respect to *delta* thrice and *tau* once, [-]

Notes

The cost of this function is 1 power, 3 exp, 2 sqrt, 1 division, and the necessary adds/multiplies.

Examples

```
>>> lemmon2000_air_d4Ar_ddelta3dtau(132.6312/200.0, 13000/10447.7)
-0.26039336747
```

1.3.6 Humid Air Virial Terms

chemicals.air.**TEOS10_BAW_derivatives**(*T*)

Calculates the second molar virial cross coefficient between air and water according to [1].

$$B_{aw}(T) = \frac{1}{\bar{\rho}^*}\sum_{i=1}^{3} c_i(\theta)^{d_i}$$

Where $\theta = T/T^*$ and $T^* = 100$ K and $\bar{\rho} = 10^6$ mol/m^3.

Parameters

T [float] Temperature, [K]

Returns

Baw [float] Air-water second molar virial cross coefficient [m^3/mol]

dBaw_dT [float] First temperature derivative of air-water second molar virial cross coefficient [m^3/(mol*K)]

d2Baw_dT2 [float] Second temperature derivative of air-water second molar virial cross coefficient [m^3/(mol*K^2)]

d3Baw_dT3 [float] Third temperature derivative of air-water second molar virial cross coefficient [m^3/(mol*K^3)]

Notes

The coefficients are as follows:

cis = [0.665687E2, -0.238834E3, -0.176755E3]

dis = [-0.237, -1.048, -3.183]

References

[1]

Examples

```
>>> TEOS10_BAW_derivatives(300.0)
(-2.956727474282386e-05, 2.8009736043809844e-07, -2.425992413058737e-09,  3.
↪0736974302787557e-11)
```

chemicals.air.**TEOS10_CAAW_derivatives**(*T*)

Calculates the third molar virial cross coefficient between air and air-water according to [1].

$$C_{aaw}(T) = \frac{1}{(\bar{\rho}^*)^2}\sum_{i=1}^{5} c_i(\theta)^{1-i}$$

Where $\theta = T/T^*$ and $T^* = 100$ K and $\bar{\rho} = 10^6$ mol/m^3.

Parameters

T [float] Temperature, [K]

Returns

Caaw [float] Air air-water second molar virial cross coefficient [m^6/mol^2]

dCaaw_dT [float] First temperature derivative of air air-water third molar virial cross coefficient [m^6/(mol^2*K)]

d2Caaw_dT2 [float] Second temperature derivative of air air-water third molar virial cross coefficient [m^6/(mol^2*K^2)]

d3Caaw_dT3 [float] Third temperature derivative of air air-water third molar virial cross coefficient [m^6/(mol^2*K^3)]

Notes

The coefficients are as follows:

cis = [0.482737E-9, 1.05678E-7, -6.56394E-5, 0.294442E-1, -3.19317]

References

[1]

Examples

```
>>> TEOS10_CAAW_derivatives(300.0)
(8.019777407407409e-10, -1.9610345679012353e-12, 1.700556378600824e-14, -1.
↪0129827160493832e-16)
```

chemicals.air.**TEOS10_CAWW_derivatives**(*T*)

Calculates the third molar virial cross coefficient between air and water-water according to [1].

$$C_{aww}(T) = \frac{1}{(\bar{\rho}^*)^2} \exp\left[\sum_{i=1}^{4} d_i(\theta)^{1-i}\right]$$

Where $\theta = T/T^*$ and $T^* = 100$ K and $\bar{\rho} = 10^6$ mol/m^3.

Parameters

T [float] Temperature, [K]

Returns

Caww [float] Air water-water second molar virial cross coefficient [m^6/mol^2]

dCaww_dT [float] First temperature derivative of air water-water third molar virial cross coefficient [m^6/(mol^2*K)]

d2Caww_dT2 [float] Second temperature derivative of air water-water third molar virial cross coefficient [m^6/(mol^2*K^2)]

d3Caww_dT3 [float] Third temperature derivative of air water-water third molar virial cross coefficient [m^6/(mol^2*K^3)]

Notes

The coefficients are as follows:

dis = [-0.10728876E2, 0.347802E2, -0.383383E2, 0.334060E2]

References

[1]

Examples

```
>>> TEOS10_CAWW_derivatives(300.0)
(-1.1555278368039349e-07, 2.6136327775413448e-09, -7.513345818045024e-11, 2.
↪601834967770415e-12)
```

1.3.7 Henry's Law for Air in Water

chemicals.air.**iapws04_Henry_air**(*T*)

Calculate the Henry's law constant of air in water according to the IAPWS-04 standard.

Parameters

T [float] Temperature, [K]

Returns

H [float] Henry's law constant, [1/Pa]

Notes

The mole fractions of air in this model are 0.7812 N2, 0.2095 O2 and 0.0093 Ar.

References

[1]

Examples

```
>>> iapws04_Henry_air(320.0)
1.0991553689889531e-10
```

chemicals.air.**iapws04_dHenry_air_dT**(*T*)

Calculate the temperature derivative of Henry's law constant of air in water according to the IAPWS-04 standard. As the actual Henry's law constant must be calculated as well, it is also returned.

Parameters

T [float] Temperature, [K]

Returns

dH_dT [float] First temperature derivative of Henry's law constant, [1/(Pa*K)]

H [float] Henry's law constant, [1/Pa]

Notes

The mole fractions of air in this model are 0.7812 N2, 0.2095 O2 and 0.0093 Ar.

References

[1]

Examples

```
>>> iapws04_dHenry_air_dT(320.0)
(-8.680064421141611e-13, 1.0991553689889561e-10)
```

1.4 Combustion Calculations (chemicals.combustion)

This module contains a series of functions for modeling combustion reactions.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.4.1 Combustion Stoichiometry

chemicals.combustion.**combustion_stoichiometry**(*atoms*, *MW=None*, *missing_handling='elemental'*)

Return a dictionary of stoichiometric coefficients of chemical combustion, given a dictionary of a molecule's constituent atoms and their counts.

This function is based on the combustion of hydrocarbons; the products for some inorganics can be hard to predict, and no special handling is included here for them. This reaction is the standard one at standard pressure with an excess of oxygen; it does not account for partial combustion or nitrous oxides.

Parameters

atoms [dict[str, int]] Dictionary of atoms and their counts, [-]

MW [float, optional] Molecular weight of chemical, used only if *missing_handling* is 'Ash', [g/mol]

missing_handling [str, optional] How to handle compounds which do not appear in the stoichiometric reaction below. If 'elemental', return those atoms in the monatomic state; if 'ash', converts all missing attoms to 'Ash' in the output at a *MW* of 1 g/mol, [-]

Returns

stoichiometry [dict[str, float]] Stoichiometric coefficients of combustion. May inlcude the following keys for complete combustion: 'H2O', 'CO2', 'SO2', 'Br2', 'I2', 'HCl', 'HF' 'P4O10'; if *missing_handling* is 'elemental' can include the other elements; if *missing_handling* is 'ash', Ash will be present in the output if the compounds whose reactions are not included here. 'O2' is always present, with negative values indicating oxygen is required. [-]

Notes

The stoichiometry is given by:

$$C_cH_hO_oN_nS_sBr_bI_iCl_xF_fP_p + kO_2 -> cCO_2 + \frac{b}{2}Br_2 + \frac{i}{2}I + xHCl + fHF + sSO_2 + \frac{n}{2}N_2 + \frac{p}{4}P_4O_{10} + \frac{h+x+f}{2}H_2$$

$$k = c + s + \frac{h}{4} + \frac{5P}{4} - \frac{x+f}{4} - \frac{o}{2}$$

Also included in the results is the moles of O2 required per mole of the mixture of the molecule.

HF and HCl are gaseous products in their standard state. P4O10 is a solid in its standard state. Bromine is a liquid as is iodine. Water depends on the chosen definition of heating value. The other products are gases.

Atoms not in ['C', 'H', 'N', 'O', 'S', 'Br', 'I', 'Cl', 'F', 'P'] are returned as pure species; i.e. sodium hydroxide produces water and pure Na.

Examples

Methane gas burning:

```
>>> combustion_stoichiometry({'C': 1, 'H':4})
{'CO2': 1, 'O2': -2.0, 'H2O': 2.0}
```

chemicals.combustion.**combustion_products_mixture**(*atoms_list*, *zs*, *reactivities=None*, *CASs=None*, *missing_handling='elemental'*, *combustion_stoichiometries=None*)

Calculates the combustion products of a mixture of molecules and their, mole fractions; requires a list of dictionaries of each molecule's constituent atoms and their counts. Products for non-hydrocarbons may not be correct, but are still calculated.

Parameters

atoms_list [list[dict]] List of dictionaries of atoms and their counts, [-]

zs [list[float]] Mole fractions of each molecule in the mixture, [-]

reactivities [list[bool]] Indicators as to whether to combust each molecule, [-]

CASs [list[str]] CAS numbers of all compounds; non-reacted products will appear in the products indexed by their CAS number, [-]

missing_handling [str, optional] How to handle compounds which do not appear in the stoichiometric reaction below. If 'elemental', return those atoms in the monatomic state; if 'Ash', converts all missing attoms to 'Ash' in the output at a *MW* of 1 g/mol, [-]

combustion_stoichiometries [list[dict[str, float]]] List of return values from *combustion_stoichiometry*, can be provided if precomputed [-]

Returns

combustion_producucts [dict] Dictionary of combustion products and their counts, [-]

Notes

Also included in the results is the moles of O2 required per mole of the mixture to be burnt.

Note that if O2 is in the feed, this will be subtracted from the required O2 amount.

HF and HCl are gaseous products in their standard state. P4O10 is a solid in its standard state. Bromine is a liquid as is iodine. Water depends on the chosen definition of heating value. The other products are gases.

Note that if instead of mole fractions, mole flows are given - the results are in terms of mole flows as well!

Examples

Mixture of methane and ethane.

```
>>> combustion_products_mixture([{'H': 4, 'C': 1}, {'H': 6, 'C': 2}, {'Ar': 1}, {'C
↪': 15, 'H': 32}],
... [.9, .05, .04, .01], reactivities=[True, True, True, False],
... CASs=['74-82-8', '74-84-0', '7440-37-1', '629-62-9'])
{'CO2': 1.0, 'O2': -1.975, 'H2O': 1.9500000000000002, 'Ar': 0.04, '629-62-9': 0.01}
```

1.4.2 Heat of Combustion

chemicals.combustion.**HHV_stoichiometry**(*stoichiometry*, *Hf*, *Hf_chemicals=None*)

Return the higher heating value [HHV; in J/mol] based on the theoretical combustion stoichiometry and the heat of formation of the chemical.

Parameters

stoichiometry [dict[str, float]] Stoichiometric coefficients of combustion. May inlcude the following keys: 'H2O', 'CO2', 'SO2', 'Br2', 'I2', 'HCl', 'HF' and 'P4O10'.

Hf [float] Heat of formation [J/mol].

Hf_chemicals [dict[str, float]] Heat of formation of chemicals present in stoichiometry, [J/mol]

Returns

HHV [float] Higher heating value [J/mol].

Notes

The combustion reaction is based on the following equation:

$$C_cH_hO_oN_nS_sBr_bI_iCl_xF_fP_p + kO_2 -> cCO_2 + \frac{b}{2}Br_2 + \frac{i}{2}I + xHCl + fHF + sSO_2 + \frac{n}{2}N_2 + \frac{p}{4}P_4O_{10} + \frac{h+x+f}{2}H_2$$

$$k = c + s + \frac{h}{4} + \frac{5P}{4} - \frac{x+f}{4} - \frac{o}{2}$$

The HHV is calculated as the heat of reaction.

Examples

Burning methane gas:

```
>>> HHV_stoichiometry({'O2': -2.0, 'CO2': 1, 'H2O': 2.0}, -74520.0)
-890604.0
```

chemicals.combustion.**HHV_modified_Dulong**(*mass_fractions*)

Return higher heating value [HHV; in J/g] based on the modified Dulong's equation [1].

Parameters

mass_fractions [dict[str, float]] Dictionary of atomic mass fractions [-].

Returns

HHV [float] Higher heating value [J/mol].

Notes

The heat of combustion in J/mol is given by Dulong's equation [1]:

$$Hc(J/mol) = MW \cdot (338C + 1428(H - O/8) + 95S)$$

This equation is only good for <10 wt. % Oxygen content. Variables C, H, O, and S are atom weight fractions.

References

[1]

Examples

Dry bituminous coal:

```
>>> HHV_modified_Dulong({'C': 0.716, 'H': 0.054, 'S': 0.016, 'N': 0.016, 'O': 0.093,
↪ 'Ash': 0.105})
-304.0395
```

chemicals.combustion.**LHV_from_HHV**(*HHV*, *N_H2O*)

Return the lower heating value [LHV; in J/mol] of a chemical given the higher heating value [HHV; in J/mol] and the number of water molecules formed per molecule burned.

Parameters

HHV [float] Higher heating value [J/mol].

N_H2O [int] Number of water molecules produced [-].

Returns

LHV [float] Lower heating value [J/mol].

Notes

The LHV is calculated as follows:

$$LHV = HHV + H_{vap} \cdot H_2O$$

$$H_{vap} = 44011.496 \frac{J}{molH_2O}$$

$$H_2O = \frac{molH_2O}{mol}$$

Examples

Methanol lower heat of combustion:

```
>>> LHV_from_HHV(-726024.0, 2)
-638001.008
```

1.4.3 Heat of Combustion and Stiochiometry

chemicals.combustion.**combustion_data**(*formula=None*, *stoichiometry=None*, *Hf=None*, *MW=None*, *method=None*, *missing_handling='ash'*)

Return a CombustionData object (a named tuple) that contains the stoichiometry coefficients of the reactants and products, the lower and higher heating values [LHV, HHV; in J/mol], the heat of formation [Hf; in J/mol], and the molecular weight [MW; in g/mol].

Parameters

formula [str, or dict[str, float], optional] Chemical formula as a string or a dictionary of atoms and their counts.

stoichiometry [dict[str, float], optional] Stoichiometry of combustion reaction.

Hf [float, optional] Heat of formation of given chemical [J/mol]. Required if method is "Stoichiometry".

MW [float, optional] Molecular weight of chemical [g/mol].

method ["Stoichiometry" or "Dulong", optional] Method to estimate LHV and HHV.

missing_handling [str, optional] How to handle compounds which do not appear in the stoichiometric reaction below. If 'elemental', return those atoms in the monatomic state; if 'Ash', converts all missing attoms to 'Ash' in the output at a *MW* of 1 g/mol, [-]

Returns

combustion_data [*CombustionData*] A combustion data object with the stoichiometric coefficients of combustion, higher heating value, heat of formation, and molecular weight as attributes named stoichiomery, HHV, Hf, and MW, respectively.

Notes

The combustion reaction is based on the following equation:

$$C_cH_hO_oN_nS_sBr_bI_iCl_xF_fP_p + kO_2 -> cCO_2 + \frac{b}{2}Br_2 + \frac{i}{2}I + xHCl + fHF + sSO_2 + \frac{n}{2}N_2 + \frac{p}{4}P_4O_{10} + \frac{h+x+f}{2}H_2$$

$$k = c + s + \frac{h}{4} + \frac{5P}{4} - \frac{x+f}{4} - \frac{o}{2}$$

If the method is "Stoichiometry", the HHV is found using through an energy balance on the reaction (i.e. heat of reaction). If the method is "Dulong", Dulong's equation is used [1]:

$$Hc(J/mol) = MW \cdot (338C + 1428(H - O/8) + 95S)$$

The LHV is calculated as follows:

$$LHV = HHV + H_{vap} \cdot H_2O$$

$$H_{vap} = 44011.496\frac{J}{molH_2O}$$

$$H_2O = \frac{molH_2O}{mol}$$

References

[1]

Examples

Liquid methanol burning:

```
>>> combustion_data({'H': 4, 'C': 1, 'O': 1}, Hf=-239100)
CombustionData(stoichiometry={'CO2': 1, 'O2': -1.5, 'H2O': 2.0}, HHV=-726024.0, Hf=-
↪239100, MW=32.04186)
```

class chemicals.combustion.**CombustionData**(*stoichiometry*, *HHV*, *Hf*, *MW*)

Return a CombustionData object (a named tuple) that contains the stoichiometry coefficients of the reactants and products, the lower and higher heating values [LHV, HHV; in J/mol], the heat of formation [Hf; in J/mol], and the molecular weight [MW; in g/mol].

Parameters

stoichiometry [dict[str, float]] Stoichiometric coefficients of the reactants and products.

HHV [float] Higher heating value [J/mol].

Hf [float] Heat of formation [J/mol].

MW [float] Molecular weight [g/mol].

Attributes

LHV Lower heating value [LHV; in J/mol]

1.4.4 Basic Combustion Spec Solvers

chemicals.combustion.**fuel_air_spec_solver**(*zs_air*, *zs_fuel*, *CASs*, *atomss*, *n_fuel=None*, *n_air=None*, *n_out=None*, *O2_excess=None*, *frac_out_O2=None*, *frac_out_O2_dry=None*, *ratio=None*, *Vm_air=None*, *Vm_fuel=None*, *MW_air=None*, *MW_fuel=None*, *ratio_basis='mass'*, *reactivities=None*, *combustion_stoichiometries=None*)

Solves the system of equations describing a flow of air mixing with a flow of combustibles and burning completely. All calculated variables are returned as a dictionary.

Supports solving with any 2 of the extensive variables, or one extensive and one intensive variable:

Extensive variables:

- *n_air*
- *n_fuel*
- *n_out*

Intensive variables:

- *O2_excess*
- *frac_out_O2*
- *frac_out_O2_dry*
- *ratio*

The variables *Vm_air*, *Vm_fuel*, *MW_air*, and *MW_fuel* are only required when an air-fuel ratio is given. Howver, the ratios cannot be calculated for the other solve options without them.

Parameters

zs_air [list[float]] Mole fractions of the air; most not contain any combustibles, [-]

zs_fuel [list[float]] Mole fractions of the fuel; can contain inerts and/or oxygen as well, [-]

CASs [list[str]] CAS numbers of all compounds, [-]

atomss [list[dict[float]]] List of dictionaries of elements and their counts for all molecules in the mixtures, [-]

n_fuel [float, optional] Flow rate of fuel, [mol/s]

n_air [float, optional] Flow rate of air, [mol/s]

n_out [float, optional] Flow rate of combustion products, remaining oxygen, and inerts, [mol/s]

O2_excess [float, optional] The excess oxygen coming out; (O2 in)/(O2 required) - 1, [-]

frac_out_O2 [float, optional] The mole fraction of oxygen out, [-]

frac_out_O2_dry [float, optional] The mole fraction of oxygen out on a dry basis, [-]

ratio [float, optional] Air-fuel ratio, in the specified *basis*, [-]

Vm_air [float, optional] Molar volume of air, [m^3/mol]

Vm_fuel [float, optional] Molar volume of fuel, [m^3/mol]

MW_air [float, optional] Molecular weight of air, [g/mol]

MW_fuel [float, optional] Molecular weight of fuel, [g/mol]

ratio_basis [str, optional] One of 'mass', 'mole', or 'volume', [-]

reactivities [list[bool], optional] Optional list which can be used to mark otherwise combustible compounds as incombustible and which will leave unreacted, [-]

combustion_stoichiometries [list[dict[str, float]]] List of return values from *combustion_stoichiometry*, can be provided if precomputed [-]

Returns

results [dict]

- n_fuel : Flow rate of fuel, [mol/s]
- n_air : Flow rate of air, [mol/s]
- n_out : Flow rate of combustion products, remaining oxygen, and inerts, [mol/s]
- O2_excess : The excess oxygen coming out; (O2 in)/(O2 required) - 1, [-]
- frac_out_O2 : The mole fraction of oxygen out, [-]
- frac_out_O2_dry : The mole fraction of oxygen out on a dry basis, [-]
- mole_ratio : Air-fuel mole ratio, [-]
- mass_ratio : Air-fuel mass ratio, [-]
- volume_ratio : Air-fuel volume ratio, [-]
- ns_out : Mole flow rates out, [mol/s]
- zs_out : Mole fractions out, [-]

Notes

Combustion products themselves cannot be set as unreactive.

The function works so long as the flow rates, molar volumes, and molecular weights are in a consistent basis.

The function may also be used to obtain the other ratios, even if both flow rates are known.

Be careful to use standard volumes if the ratio known is at standard conditions!

Examples

```
>>> zs_air = [0.79, 0.205, 0, 0, 0, 0.0045, 0.0005]
>>> zs_fuel = [0.025, 0.025, 0.85, 0.07, 0.029, 0.0005, 0.0005]
>>> CASs = ['7727-37-9', '7782-44-7', '74-82-8', '74-84-0', '74-98-6', '7732-18-5',
↪'124-38-9']
>>> atomss = [{'N': 2}, {'O': 2}, {'H': 4, 'C': 1}, {'H': 6, 'C': 2}, {'H': 8, 'C':␣
↪3}, {'H': 2, 'O': 1}, {'C': 1, 'O': 2}]
>>> ans = fuel_air_spec_solver(zs_air=zs_air, zs_fuel=zs_fuel, CASs=CASs,␣
↪atomss=atomss, n_fuel=5.0, O2_excess=0.3, Vm_air=0.02493, Vm_fuel=0.02488, MW_
↪air=28.79341351, MW_fuel=18.55158039)
>>> [round(i, 5) for i in ans['ns_out']]
[51.99524, 3.135, 0.0, 0.0, 0.0, 10.42796, 5.42033]
>>> [round(i, 5) for i in ans['zs_out']]
[0.73255, 0.04417, 0.0, 0.0, 0.0, 0.14692, 0.07637]
>>> ans['frac_out_O2'], ans['frac_out_O2_dry']
(0.04416828172034148, 0.051774902132807)
>>> ans['mole_ratio'], ans['mass_ratio'], ans['volume_ratio']
```

(continues on next page)

(continued from previous page)

```
(13.131707317073175, 20.381372957130615, 13.15809740412517)
>>> ans['n_air']
65.65853658536588
```

chemicals.combustion.**combustion_spec_solver**(*zs_air*, *zs_fuel*, *zs_third*, *CASs*, *atomss*, *n_third*, *n_fuel=None*, *n_air=None*, *n_out=None*, *O2_excess=None*, *frac_out_O2=None*, *frac_out_O2_dry=None*, *ratio=None*, *Vm_air=None*, *Vm_fuel=None*, *Vm_third=None*, *MW_air=None*, *MW_fuel=None*, *MW_third=None*, *ratio_basis='mass'*, *reactivities=None*, *combustion_stoichiometries=None*)

Solves the system of equations describing a flow of air mixing with two flow of combustibles, one fixed and one potentially variable, and burning completely. All calculated variables are returned as a dictionary.

The variables *Vm_air*, *Vm_fuel*, *Vm_third*, *MW_air*, *MW_fuel* and *MW_third* are only required when an air-fuel ratio is given. Howver, the ratios cannot be calculated for the other solve options without them.

Parameters

zs_air [list[float]] Mole fractions of the air; most not contain any combustibles, [-]

zs_fuel [list[float]] Mole fractions of the fuel; can contain inerts and/or oxygen as well, [-]

zs_third [list[float]]

Mole fractions of the fixed fuel flow; can contain inerts and/or oxygen as well, [-]

CASs [list[str]] CAS numbers of all compounds, [-]

atomss [list[dict[float]]] List of dictionaries of elements and their counts for all molecules in the mixtures, [-]

n_third [float, optional] Flow rate of third stream, (fixed) fuel flow rate, [mol/s]

n_fuel [float, optional] Flow rate of fuel, [mol/s]

n_air [float, optional] Flow rate of air, [mol/s]

n_out [float, optional] Flow rate of combustion products, remaining oxygen, and inerts, [mol/s]

O2_excess [float, optional] The excess oxygen coming out; (O2 in)/(O2 required) - 1, [-]

frac_out_O2 [float, optional] The mole fraction of oxygen out, [-]

frac_out_O2_dry [float, optional] The mole fraction of oxygen out on a dry basis, [-]

ratio [float, optional] Air-fuel ratio, in the specified *basis*, [-]

Vm_air [float, optional] Molar volume of air, [m^3/mol]

Vm_fuel [float, optional] Molar volume of fuel, [m^3/mol]

Vm_third [float, optional] Molar volume of second fuel stream, [m^3/mol]

MW_air [float, optional] Molecular weight of air, [g/mol]

MW_fuel [float, optional] Molecular weight of fuel, [g/mol]

MW_third [float, optional] Molecular weight of second fuel stream, [g/mol]

ratio_basis [str, optional] One of 'mass', 'mole', or 'volume', [-]

reactivities [list[bool], optional] Optional list which can be used to mark otherwise combustible compounds as incombustible and which will leave unreacted, [-]

combustion_stoichiometries [list[dict[str, float]]] List of return values from *combustion_stoichiometry*, can be provided if precomputed [-]

Returns

results [dict]

- n_fuel : Flow rate of fuel, [mol/s]
- n_air : Flow rate of air, [mol/s]
- n_out : Flow rate of combustion products, remaining oxygen, and inerts, [mol/s]
- O2_excess : The excess oxygen coming out; (O2 in)/(O2 required) - 1, [-]
- frac_out_O2 : The mole fraction of oxygen out, [-]
- frac_out_O2_dry : The mole fraction of oxygen out on a dry basis, [-]
- mole_ratio : Air-fuel mole ratio, [-]
- mass_ratio : Air-fuel mass ratio, [-]
- volume_ratio : Air-fuel volume ratio, [-]
- ns_out : Mole flow rates out, [mol/s]
- zs_out : Mole fractions out, [-]

Notes

Combustion products themselves cannot be set as unreactive.

The function works so long as the flow rates, molar volumes, and molecular weights are in a consistent basis.

Handling the case of the air feed containing combustibles is not implemented.

Examples

```
>>> zs_air = [0.79, 0.205, 0, 0, 0, 0.0045, 0.0005]
>>> zs_fuel = [0.025, 0.025, 0.85, 0.07, 0.029, 0.0005, 0.0005]
>>> zs_third = [0.1, 0.005, 0.5, 0.39, 0, 0.005, 0]
>>> CASs = ['7727-37-9', '7782-44-7', '74-82-8', '74-84-0', '74-98-6', '7732-18-5',
↪'124-38-9']
>>> atomss = [{'N': 2}, {'O': 2}, {'H': 4, 'C': 1}, {'H': 6, 'C': 2}, {'H': 8, 'C':␣
↪3}, {'H': 2, 'O': 1}, {'C': 1, 'O': 2}]
>>> combustion_stoichiometries = [combustion_stoichiometry(atoms) for atoms in␣
↪atomss]
>>> ans = combustion_spec_solver(zs_air=zs_air, zs_fuel=zs_fuel, zs_third=zs_third,␣
↪CASs=CASs, atomss=atomss, n_third=1.0, n_fuel=5.0, O2_excess=0.3, Vm_air=0.02493,␣
↪Vm_fuel=0.02488, Vm_third=.024, MW_air=28.79341351, MW_fuel=18.55158039, MW_
↪third=22.0)
>>> ans['n_air']
80.6317073170732
```

chemicals.combustion.**air_fuel_ratio_solver**(*ratio*, *Vm_air*, *Vm_fuel*, *MW_air*, *MW_fuel*, *n_air=None*, *n_fuel=None*, *basis='mass'*)

Calculates molar flow rate of air or fuel from the other, using a specified air-fuel ratio. Supports 'mole', 'mass', and 'volume'.

bases for the ratio variable. The ratio must be of the same units - i.e. kg/kg instead of lb/kg.

The mole, mass, and volume air-fuel ratios are calculated in the process and returned as well.

Parameters

ratio [float] Air-fuel ratio, in the specified *basis*, [-]

Vm_air [float] Molar volume of air, [m^3/mol]

Vm_fuel [float] Molar volume of fuel, [m^3/mol]

MW_air [float] Molecular weight of air, [g/mol]

MW_fuel [float] Molecular weight of fuel, [g/mol]

n_air [float, optional] Molar flow rate of air, [mol/s]

n_fuel [float, optional] Molar flow rate of fuel, [mol/s]

basis [str, optional] One of 'mass', 'mole', or 'volume', [-]

Returns

n_air [float] Molar flow rate of air, [mol/s]

n_fuel [float] Molar flow rate of fuel, [mol/s]

mole_ratio [float] Air-fuel mole ratio, [-]

mass_ratio [float] Air-fuel mass ratio, [-]

volume_ratio [float] Air-fuel volume ratio, [-]

Notes

The function works so long as the flow rates, molar volumes, and molecular weights are in a consistent basis.

The function may also be used to obtain the other ratios, even if both flow rates are known.

Be careful to use standard volumes if the ratio known is at standard conditions!

This function has no provision for mixed units like mass/mole or volume/mass.

Examples

```
>>> Vm_air = 0.024936627188566596
>>> Vm_fuel = 0.024880983160354486
>>> MW_air = 28.850334
>>> MW_fuel = 17.86651
>>> n_fuel = 5.0
>>> n_air = 25.0
>>> air_fuel_ratio_solver(ratio=5.0, Vm_air=Vm_air, Vm_fuel=Vm_fuel,
... MW_air=MW_air, MW_fuel=MW_fuel, n_air=n_air,
... n_fuel=n_fuel, basis='mole')
(25.0, 5.0, 5.0, 8.073858296891782, 5.011182039683378)
```

1.4.5 Engine Combustion

chemicals.combustion.**Perez_Boehman_RON_from_ignition_delay**(*ignition_delay*)
Esimates the research octane number (RON) from a known ignition delay, as shown in [1].

$$\text{RON} = 120.77 - \frac{425.48}{\tau_{ID}}$$

In the above equation, ignition delay is in ms.

Parameters

ignition_delay [float] The ignition delay, [s]

Returns

RON [float] Research Octane Number [-]

Notes

The correlation was developed using 20 components, for a range of approximately 3.6 ms to 67 ms.

References

[1]

Examples

```
>>> Perez_Boehman_RON_from_ignition_delay(1/150)
56.948
```

chemicals.combustion.**Perez_Boehman_MON_from_ignition_delay**(*ignition_delay*)
Esimates the motor octane number (MON) from a known ignition delay, as shown in [1].

$$\text{MON} = 109.93 - \frac{374.73}{\tau_{ID}}$$

In the above equation, ignition delay is in ms.

Parameters

ignition_delay [float] The ignition delay, [s]

Returns

MON [float] Motor Octane Number [-]

Notes

The correlation was developed using 20 components, for a range of approximately 3.6 ms to 67 ms.

References

[1]

Examples

```
>>> Perez_Boehman_MON_from_ignition_delay(1/150)
53.7205
```

chemicals.combustion.**octane_sensitivity**(*RON*, *MON*)
This function calculates the octane sensitivity of a fuel [1].

$$\text{OS} = \text{RON} - \text{MON}$$

Parameters

RON [float] Research octane number, [-]

MON [float] Motor octane number, [-]

Returns

OS [float] Octane sensitivity, [-]

References

[1]

Examples

```
>>> octane_sensitivity(RON=90, MON=74)
16
```

chemicals.combustion.**AKI**(*RON*, *MON*)
This function calculates the anti knock index (AKI) of a fuel, also known as (R+M)/2 and by DON [1].

$$\text{AKI} = 0.5\text{RON} + 0.5\text{MON}$$

Parameters

RON [float] Research octane number, [-]

MON [float] Motor octane number, [-]

Returns

AKI [float] Average of RON and MON, [-]

Notes

This is the number displayed at the gas pumps in North America; in Europe and Asia the RON is displayed.

References

[1]

Examples

```
>>> AKI(RON=90, MON=74)
82.0
```

chemicals.combustion.**IDT_to_DCN**(*IDT*)

This function converts the ignition delay time [1] into a derived cetane number.

If the ignition delay time is between 3.1 and 6.5 ms:

$$\text{DCN} = 4.46 + \frac{186.6}{\text{IDT}}$$

Otherwise:

$$\text{DCN} = \left(83.99(\text{IDT} - 1.512)^{-0.658}\right) + 3.547$$

Parameters

IDT [float] Ignition delay time, [s]

Returns

DCN [float] Derived cetane number, [-]

Notes

This conversion is described in D6890-168.

References

[1], [2]

Examples

```
>>> IDT_to_DCN(4e-3)
51.11
```

1.4.6 Lookup Functions

chemicals.combustion.**RON**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's research octane number (RON). Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 1400 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

RON [float] Research octane number, [-]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in RON_methods

Notes

The available sources are as follows:

- 'FLORIAN_LIMING', the experimental values compiled in [1].
- 'FLORIAN_LIMING_ANN', a set of predicted values using a QSPR-ANN model developed in the author's earlier publication [3], from 260 comonents.
- 'COMBUSTDB', a compilation of values from various sources [2].
- 'COMBUSTDB_PREDICTIONS', a set of predicted values developed by the author of CombustDB (Travis Kessler) using the tool [4].

References

[1], [2], [3], [4]

Examples

```
>>> RON(CASRN='64-17-5')
108.6
```

chemicals.combustion.**RON_methods**(*CASRN*)

Return all methods available to obtain the research octane number (RON) for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the RON with the given inputs.

See also:

RON

chemicals.combustion.**RON_all_methods = ('FLORIAN_LIMING', 'COMBUSTDB', 'FLORIAN_LIMING_ANN', 'COMBUSTDB_PREDICTIONS')**
Tuple of method name keys. See the *RON* for the actual references

chemicals.combustion.**MON**(*CASRN*, *method=None*)
This function handles the retrieval of a chemical's motor octane number (MON). Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 1400 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

MON [float] Research octane number, [-]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in MON_methods

Notes

The available sources are as follows:

- 'FLORIAN_LIMING', the experimental values compiled in [1].
- 'FLORIAN_LIMING_ANN', a set of predicted values using a QSPR-ANN model developed in the author's earlier publication [3], from 260 comonents.
- 'COMBUSTDB', a compilation of values from various sources [2].
- 'COMBUSTDB_PREDICTIONS', a set of predicted values developed by the author of CombustDB (Travis Kessler) using the tool [4].

References

[1], [2], [3], [4]

Examples

```
>>> MON(CASRN='64-17-5')
89.7
```

chemicals.combustion.**MON_methods**(*CASRN*)
Return all methods available to obtain the motor octane number (MON) for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the MON with the given inputs.

See also:

MON

chemicals.combustion.**MON_all_methods = ('FLORIAN_LIMING', 'COMBUSTDB', 'FLORIAN_LIMING_ANN', 'COMBUSTDB_PREDICTIONS')**
Tuple of method name keys. See the *MON* for the actual references

chemicals.combustion.**ignition_delay**(*CASRN*, *method=None*)
This function handles the retrieval of a chemical's ignition delay time (IDT). Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 60 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

ignition_delay [float] Ignition delay time, [s]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in ignition_delay_all_methods

Notes

The available sources are as follows:

- 'DAHMEN_MARQUARDT', the experimental values compiled in [1]; all timings come from the IQT tester device

Note that different measurement devices can give different results.

References

[1]

Examples

```
>>> ignition_delay(CASRN='110-54-3')
0.0043
```

chemicals.combustion.**ignition_delay_methods**(*CASRN*)
Return all methods available to obtain the ignition delay time (IDT) for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the IDT with the given inputs.

See also:

ignition_delay

chemicals.combustion.**ignition_delay_all_methods = ('DAHMEN_MARQUARDT',)**
Tuple of method name keys. See the *ignition_delay* for the actual references

1.5 Critical Properties (chemicals.critical)

This module contains lookup functions for critical temperature, critical pressure, critical volume, and critical compressibility factors. It also includes a few relationships between the critical properties, and a variety of critical mixture property estimation routines.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.5.1 Critical Temperature

chemicals.critical.**Tc**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's critical temperature. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 26000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Tc [float] Critical temperature, [K]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'IUPAC', 'MATTHEWS', 'CRC', 'PD', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'WILSON_JASPERSON', 'JOBACK', 'HEOS'. All valid values are also held in the list *Tc_all_methods*.

See also:

Tc_methods

Notes

The available sources are as follows:

- 'IUPAC', a series of critically evaluated experimental datum for organic compounds in [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], and [12].
- 'MATTHEWS', a series of critically evaluated data for inorganic compounds in [13].
- 'CRC', a compillation of critically evaluated data by the TRC as published in [14].
- 'PSRK', a compillation of experimental and estimated data published in [15].
- 'PD', an older compillation of data published in [16]
- 'YAWS', a large compillation of data from a variety of sources; no data points are sourced in the work of [17].
- 'WEBBOOK', a NIST resource [18] containing mostly experimental and averaged values
- 'JOBACK', an estimation method for organic substances in [19]
- 'WILSON_JASPERSON', an estimation method in [21]
- 'PINAMARTINES', a series of values in the supporting material of [20]
- 'HEOS', a series of values from the NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids (and other high-precision fundamental equations of state)

References

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]

Examples

```
>>> Tc(CASRN='64-17-5')
514.71
```

chemicals.critical.**Tc_methods**(*CASRN*)

Return all methods available to obtain the critical temperature for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain Tc with the given inputs.

See also:

Tc

chemicals.critical.**Tc_all_methods** = ('HEOS', 'IUPAC', 'MATTHEWS', 'CRC', 'PD', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'WILSON_JASPERSON', 'JOBACK')

Tuple of method name keys. See the *Tc* for the actual references

chemicals.critical.**Tc_all_method_types = {'CRC': 'PROCESSED_EXPERIMENTAL', 'HEOS': 'EXPERIMENTAL_REVIEW', 'IUPAC': 'EXPERIMENTAL_REVIEW', 'JOBACK': 'PREDICTED_GC', 'MATTHEWS': 'EXPERIMENTAL_COMPILATION', 'PD': 'EXPERIMENTAL_COMPILATION_SECONDARY', 'PINAMARTINES': 'PROCESSED_EXPERIMENTAL_PREDICTED_SECONDARY', 'PSRK': 'PROCESSED_EXPERIMENTAL_PREDICTED', 'WEBBOOK': 'PROCESSED_EXPERIMENTAL', 'WILSON_JASPERSON': 'PREDICTED_GC', 'YAWS': 'PROCESSED_EXPERIMENTAL_PREDICTED'}**

1.5.2 Critical Pressure

chemicals.critical.**Pc**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's critical pressure. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 26000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Pc [float] Critical pressure, [Pa]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'IUPAC', 'MATTHEWS', 'CRC', 'PD', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'WILSON_JASPERSON', 'JOBACK', 'HEOS'. All valid values are also held in the list *Pc_all_methods*.

See also:

Pc_methods

Notes

The available sources are as follows:

- 'IUPAC', a series of critically evaluated experimental datum for organic compounds in [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], and [12].
- 'MATTHEWS', a series of critically evaluated data for inorganic compounds in [13].
- 'CRC', a compillation of critically evaluated data by the TRC as published in [14].
- 'PSRK', a compillation of experimental and estimated data published in [15].
- 'PD', an older compillation of data published in [16]
- 'YAWS', a large compillation of data from a variety of sources; no data points are sourced in the work of [17].
- 'WEBBOOK', a NIST resource [18] containing mostly experimental and averaged values
- 'JOBACK', an estimation method for organic substances in [19]
- 'PINAMARTINES', a series of values in the supporting material of [20]
- 'WILSON_JASPERSON', an estimation method in [21]
- 'HEOS', a series of values from the NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids (and other high-precision fundamental equations of state)

References

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]

Examples

```
>>> Pc(CASRN='64-17-5')
6268000.0
```

chemicals.critical.**Pc_methods**(*CASRN*)

Return all methods available to obtain the critical pressure for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain Pc with the given inputs.

See also:

Pc

chemicals.critical.**Pc_all_methods = ('HEOS', 'IUPAC', 'MATTHEWS', 'CRC', 'PD', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'WILSON_JASPERSON', 'JOBACK')**

Tuple of method name keys. See the *Pc* for the actual references

1.5.3 Critical Volume

chemicals.critical.**Vc**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's critical volume. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Preferred sources are 'IUPAC' for organic chemicals, and 'MATTHEWS' for inorganic chemicals. Function has data for approximately 25000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Vc [float] Critical volume, [m^3/mol]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'IUPAC', 'MATTHEWS', 'CRC', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'FEDORS', 'JOBACK', 'HEOS'. All valid values are also held in the list *Vc_all_methods*.

See also:

Vc_methods

Notes

The available sources are as follows:

- 'IUPAC', a series of critically evaluated experimental datum for organic compounds in [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], and [12].
- 'MATTHEWS', a series of critically evaluated data for inorganic compounds in [13].
- 'CRC', a compillation of critically evaluated data by the TRC as published in [14].
- 'PSRK', a compillation of experimental and estimated data published in [15].
- 'YAWS', a large compillation of data from a variety of sources; no data points are sourced in the work of [16].
- 'WEBBOOK', a NIST resource [17] containing mostly experimental and averaged values
- 'JOBACK', an estimation method for organic substances in [18]
- 'FEDORS', an estimation methid in [20]
- 'PINAMARTINES', a series of values in the supporting material of [19]
- 'HEOS', a series of values from the NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids (and other high-precision fundamental equations of state)

References

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]

Examples

```
>>> Vc(CASRN='64-17-5')
0.000168634064081
```

chemicals.critical.**Vc_methods**(*CASRN*)

Return all methods available to obtain the critical volume for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain Vc with the given inputs.

See also:

Vc

chemicals.critical.**Vc_all_methods = ('HEOS', 'IUPAC', 'MATTHEWS', 'CRC', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'FEDORS', 'JOBACK')**

Tuple of method name keys. See the *Vc* for the actual references

chemicals.critical.**Mersmann_Kind_predictor**(*atoms*, *coeff=3.645*, *power=0.5*, *covalent_radii={'Br': 1.14, 'C': 0.77, 'Cl': 0.99, 'F': 0.71, 'H': 0.37, 'I': 1.33, 'N': 0.71, 'O': 0.6, 'S': 1.04, 'Si': 1.17}*)

Predicts the critical molar volume of a chemical based only on its atomic composition according to [1] and [2]. This is a crude approach, but provides very reasonable estimates in practice. Optionally, the *coeff* used and the

power in the fraction as well as the atomic contributions can be adjusted; this method is general and atomic contributions can be regressed to predict other properties with this routine.

$$\frac{\left(\frac{V_c}{n_a N_A}\right)^{1/3}}{d_a} = \frac{3.645}{\left(\frac{r_a}{r_H}\right)^{1/2}}$$

$$r_a = d_a/2$$

$$d_a = 2\frac{\sum_i (n_i r_i)}{n_a}$$

In the above equations, n_i is the number of atoms of species i in the molecule, r_i is the covalent atomic radius of the atom, and n_a is the total number of atoms in the molecule.

Parameters

atoms [dict] Dictionary of atoms and their counts, [-]

coeff [float, optional] Coefficient used in the relationship, [m^2]

power [float, optional] Power applied to the relative atomic radius, [-]

covalent_radii [dict or indexable, optional] Object which can be indexed to atomic contrinbutions (by symbol), [-]

Returns

Vc [float] Predicted critical volume of the chemical, [m^3/mol]

Notes

Using the *chemicals.elements.periodic_table* covalent radii (from RDKit), the coefficient and power should be 4.261206523632586 and 0.5597281770786228 respectively for best results.

References

[1], [2]

Examples

Prediction of critical volume of decane:

```
>>> Mersmann_Kind_predictor({'C': 10, 'H': 22})
0.0005851858957767497
```

This is compared against the experimental value, 0.000624 (a 6.2% relative error)

Using custom fitted coefficients we can do a bit better:

```
>>> from chemicals.critical import rcovs_regressed
>>> Mersmann_Kind_predictor({'C': 10, 'H': 22}, coeff=4.261206523632586,
... power=0.5597281770786228, covalent_radii=rcovs_regressed)
0.0005956870915974391
```

The relative error is only 4.5% now. This is compared to an experimental uncertainty of 5.6%.

Evaluating 1321 critical volumes in the database, the average relative error is 5.0%; standard deviation 6.8%; and worst value of 79% relative error for phosphorus.

1.5.4 Critical Compressibility Factor

chemicals.critical.**Zc**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's critical compressibility. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Preferred sources are 'IUPAC' for organic chemicals, and 'MATTHEWS' for inorganic chemicals. Function has data for approximately 25000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Zc [float] Critical compressibility, [-]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'IUPAC', 'MATTHEWS', 'CRC', 'PSRK', 'YAWS', 'HEOS'. All valid values are also held in *Zc_all_methods*.

See also:

Zc_methods

Notes

The available sources are as follows:

- 'IUPAC', a series of critically evaluated experimental datum for organic compounds in [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], and [12].
- 'MATTHEWS', a series of critically evaluated data for inorganic compounds in [13].
- 'CRC', a compillation of critically evaluated data by the TRC as published in [14].
- 'PSRK', a compillation of experimental and estimated data published in [15].
- 'YAWS', a large compillation of data from a variety of sources; no data points are sourced in the work of [16].
- 'WEBBOOK', a NIST resource [17] containing mostly experimental and averaged values
- 'JOBACK', an estimation method for organic substances in [18]
- 'PINAMARTINES', a series of values in the supporting material of [19]
- 'HEOS', a series of values from the NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids (and other high-precision fundamental equations of state)

References

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20]

Examples

```
>>> Zc(CASRN='64-17-5')
0.247
```

chemicals.critical.**Zc_methods**(*CASRN*)

Return all methods available to obtain the critical compressibility for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain Zc with the given inputs.

See also:

Zc

chemicals.critical.**Zc_all_methods = ('HEOS', 'IUPAC', 'MATTHEWS', 'CRC', 'WEBBOOK', 'PSRK', 'PINAMARTINES', 'YAWS', 'JOBACK')**

Tuple of method name keys. See the *Zc* for the actual references

1.5.5 Critical Property Relationships

chemicals.critical.**critical_surface**(*Tc=None*, *Pc=None*, *Vc=None*, *method=None*)

Function for calculating a critical property of a substance from its other two critical properties. Calls functions Ihmels, Meissner, and Grigoras, each of which use a general 'Critical surface' type of equation. Limited accuracy is expected due to very limited theoretical backing.

Parameters

Tc [float] Critical temperature of fluid (optional) [K].

Pc [float] Critical pressure of fluid (optional) [Pa].

Vc [float] Critical volume of fluid (optional) [m^3/mol].

method [string] Request calculation uses the requested method.

Returns

Tc, Pc or Vc [float] Critical property of fluid [K], [Pa], or [m^3/mol].

See also:

critical_surface_methods_methods

Examples

Decamethyltetrasiloxane [141-62-8]

```
>>> critical_surface(Tc=599.4, Pc=1.19E6, method='IHMELS')
0.0010927333333333334
```

chemicals.critical.**critical_surface_methods**(*Tc=None*, *Pc=None*, *Vc=None*)
Return all methods available to obtain the third critial property for the desired chemical.

Parameters

Tc [float] Critical temperature of fluid (optional) [K].

Pc [float] Critical pressure of fluid (optional) [Pa].

Vc [float] Critical volume of fluid (optional) [m^3/mol].

Returns

methods [list[str]] Methods which can be used to obtain the third critical property with the given inputs.

See also:

critical_surface

chemicals.critical.**critical_surface_all_methods = ('IHMELS', 'MEISSNER', 'GRIGORAS')**
Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

chemicals.critical.**third_property**(*CASRN=None*, *T=False*, *P=False*, *V=False*)
Function for calculating a critical property of a substance from its other two critical properties, but retrieving the actual other critical values for convenient calculation. Calls functions Ihmels, Meissner, and Grigoras, each of which use a general 'Critical surface' type of equation. Limited accuracy is expected due to very limited theoretical backing.

Parameters

CASRN [str] The CAS number of the desired chemical

T [bool] Estimate critical temperature

P [bool] Estimate critical pressure

V [bool] Estimate critical volume

Returns

Tc, Pc or Vc [float] Critical property of fluid [K], [Pa], or [m^3/mol]

Examples

Decamethyltetrasiloxane [141-62-8]

```
>>> third_property('141-62-8', V=True)
0.001135732
```

Succinic acid [110-15-6]

```
>>> third_property('110-15-6', P=True)
6095016.233766234
```

chemicals.critical.**Ihmels**(*Tc=None*, *Pc=None*, *Vc=None*)

Most recent, and most recommended method of estimating critical properties from each other. Two of the three properties are required. This model uses the "critical surface", a general plot of Tc vs Pc vs Vc. The model used 421 organic compounds to derive equation. The general equation is in [1]:

$$P_c = -0.025 + 2.215\frac{T_c}{V_c}$$

Parameters

Tc [float] Critical temperature of fluid (optional) [K]

Pc [float] Critical pressure of fluid (optional) [Pa]

Vc [float] Critical volume of fluid (optional) [m^3/mol]

Returns

Tc, Pc or Vc [float] Critical property of fluid [K], [Pa], or [m^3/mol]

Notes

The prediction of Tc from Pc and Vc is not tested, as this is not necessary anywhere, but it is implemented. Internal units are MPa, cm^3/mol, and K. A slight error occurs when Pa, cm^3/mol and K are used instead, on the order of <0.2%. Their equation was also compared with 56 inorganic and elements. Devations of 20% for <200K or >1000K points.

References

[1]

Examples

Succinic acid [110-15-6]

```
>>> Ihmels(Tc=851.0, Vc=0.000308)
6095016.233766234
```

chemicals.critical.**Meissner**(*Tc=None*, *Pc=None*, *Vc=None*)

Old (1942) relationship for estimating critical properties from each other. Two of the three properties are required. This model uses the "critical surface", a general plot of Tc vs Pc vs Vc. The model used 42 organic and inorganic compounds to derive the equation. The general equation is in [1]:

$$P_c = \frac{2.08T_c}{V_c - 8}$$

Parameters

Tc [float, optional] Critical temperature of fluid [K]

Pc [float, optional] Critical pressure of fluid [Pa]

Vc [float, optional] Critical volume of fluid [m^3/mol]

Returns

Tc, Pc or Vc [float] Critical property of fluid [K], [Pa], or [m^3/mol]

Notes

The prediction of Tc from Pc and Vc is not tested, as this is not necessary anywhere, but it is implemented. Internal units are atm, cm^3/mol, and K. A slight error occurs when Pa, cm^3/mol and K are used instead, on the order of <0.2%. This equation is less accurate than that of Ihmels, but surprisingly close. The author also proposed means of estimated properties independently.

References

[1]

Examples

Succinic acid [110-15-6]

```
>>> Meissner(Tc=851.0, Vc=0.000308)
5978445.199999999
```

chemicals.critical.**Grigoras**(*Tc=None*, *Pc=None*, *Vc=None*)

Relatively recent (1990) relationship for estimating critical properties from each other. Two of the three properties are required. This model uses the "critical surface", a general plot of Tc vs Pc vs Vc. The model used 137 organic and inorganic compounds to derive the equation. The general equation is in [1]:

$$P_c = 2.9 + 20.2\frac{T_c}{V_c}$$

Parameters

Tc [float, optional] Critical temperature of fluid [K]

Pc [float, optional] Critical pressure of fluid [Pa]

Vc [float, optional] Critical volume of fluid [m^3/mol]

Returns

Tc, Pc or Vc [float] Critical property of fluid [K], [Pa], or [m^3/mol]

Notes

The prediction of Tc from Pc and Vc is not tested, as this is not necessary anywhere, but it is implemented. Internal units are bar, cm^3/mol, and K. A slight error occurs when Pa, cm^3/mol and K are used instead, on the order of <0.2%. This equation is less accurate than that of Ihmels, but surprisingly close. The author also investigated an early QSPR model.

References

[1]

Examples

Succinic acid [110-15-6]

```
>>> Grigoras(Tc=851.0, Vc=0.000308)
5871233.766233766
```

chemicals.critical.**Hekayati_Raeissi**(*MW*, *V_sat=None*, *Tc=None*, *Pc=None*, *Vc=None*)

Estimation model for missing critical constants of a fluid according to [1]. Based on the molecular weight and saturation molar volume of a fluid, and requires one of *Tc* or *Pc*. Optionally, *Vc* can be provided to increase the accuracy of the prediction of *Tc* or *Pc* a little.

Parameters

MW [float] Molecular weight of fluid, [g/mol]

V_sat [float, optional] Molar volume of liquid at the saturation pressure of the fluid at 298.15 K. Used if *Vc* is not provided. [m^3/mol]

Tc [float, optional] Critical temperature of fluid (optional) [K]

Pc [float, optional] Critical pressure of fluid (optional) [Pa]

Vc [float, optional] Critical volume of fluid (optional) [m^3/mol]

Returns

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Vc [float] Critical volume of fluid [m^3/mol]

Notes

Internal units are kPa, m^3/kmol, and K.

References

[1]

Examples

Toluene

```
>>> Hekayati_Raeissi(MW=92.13842, V_sat=0.00010686, Pc=4108000.0)
(599.7965819136947, 4108000.0, 0.000314909150453723)
```

chemicals.critical.**Tb_Tc_relationship**(*Tb=None*, *Tc=None*, *fit='Perry8E'*)

This function relates the normal boiling point and the critical point of a compound. It is inspired by the relationship shown in [1] on page 2-468 for inorganic compounds.

$$T_c = 1.64 T_b$$

Parameters

Tb [float, optional] Normal boiling temperature of fluid [K]

Tc [float, optional] Critical temperature of fluid [K]

fit [str, optional] One of 'Perry8E', 'Chemicals2021FitInorganic', 'Chemicals2021FitElements', 'Chemicals2021FitBinary', 'Chemicals2021FitTernary', Chemicals2021FitOrganic', 'Chemicals2021FitBr', 'Chemicals2021FitC', 'Chemicals2021FitCl', 'Chemicals2021FitF', 'Chemicals2021FitI', 'Chemicals2021FitN', 'Chemicals2021FitO', ' 'Chemicals2021FitSi'.

Returns

Tc or Tb [float] The temperature variable not provided [K]

Notes

Chemicals2021FitBinary applies for inorganic compounds with two types of atoms; *Chemicals2021FitTernary* for three; and the various models *Chemicals2021FitO*, *Chemicals2021FitC*, etc apply for inorganic compounds with those elements in them.

The quality of this relationship is low, but if no further information is available it can be used to obtain an approximate value.

References

[1]

Examples

Tetrabromosilane has a known boiling point of 427.15 K and a critical temperature of 663.0 K.

```
>>> Tb_Tc_relationship(Tb=427.15, fit='Perry8E')
700.526
>>> Tb_Tc_relationship(Tb=427.15, fit='Chemicals2021FitBr')
668.0626
>>> Tb_Tc_relationship(Tb=427.15, fit='Chemicals2021FitSi')
```

(continues on next page)

(continued from previous page)

```
651.8309
>>> Tb_Tc_relationship(Tb=427.15, fit='Chemicals2021FitBinary')
669.7712
>>> Tb_Tc_relationship(Tb=427.15, fit='Chemicals2021FitInorganic')
686.0029
```

The performance of the fits is fairly representative. However, because this method should only be used on compounds that don't have experimental critical points measured, many of the worst outlier chemicals have already been measured and the performance may be better than expected.

It is recommended to use the methods *Chemicals2021FitElements*, *Chemicals2021FitBinary*, and *Chemicals2021FitTernary*.

1.5.6 Critical Temperature of Mixtures

chemicals.critical.**Li**(*zs*, *Tcs*, *Vcs*)

Calculates critical temperature of a mixture according to mixing rules in [1]. Better than simple mixing rules.

$$T_{cm} = \sum_{i=1}^{n} \Phi_i T_{ci}$$

$$\Phi = \frac{x_i V_{ci}}{\sum_{j=1}^{n} x_j V_{cj}}$$

Parameters

zs [array-like] Mole fractions of all components

Tcs [array-like] Critical temperatures of all components, [K]

Vcs [array-like] Critical volumes of all components, [m^3/mol]

Returns

Tcm [float] Critical temperatures of the mixture, [K]

Notes

Reviewed in many papers on critical mixture temperature.

Second example is from Najafi (2015), for ethylene, Benzene, ethylbenzene. This is similar to but not identical to the result from the article. The experimental point is 486.9 K.

2rd example is from Najafi (2015), for: butane/pentane/hexane 0.6449/0.2359/0.1192 mixture, exp: 450.22 K. Its result is identical to that calculated in the article.

References

[1]

Examples

Nitrogen-Argon 50/50 mixture

```
>>> Li([0.5, 0.5], [126.2, 150.8], [8.95e-05, 7.49e-05])
137.40766423357667
```

butane/pentane/hexane 0.6449/0.2359/0.1192 mixture, exp: 450.22 K.

```
>>> Li([0.6449, 0.2359, 0.1192], [425.12, 469.7, 507.6],
... [0.000255, 0.000313, 0.000371])
449.68261498555444
```

chemicals.critical.**Chueh_Prausnitz_Tc**(*zs*, *Tcs*, *Vcs*, *taus*)

Calculates critical temperature of a mixture according to mixing rules in [1].

$$T_{cm} = \sum_i^n \theta_i T c_i + \sum_i^n \sum_j^n (\theta_i \theta_j \tau_{ij}) T_{ref}$$

$$\theta = \frac{x_i V_{ci}^{2/3}}{\sum_{j=1}^n x_j V_{cj}^{2/3}}$$

For a binary mxiture, this simplifies to:

$$T_{cm} = \theta_1 T_{c1} + \theta_2 T_{c2} + 2\theta_1\theta_2\tau_{12}$$

Parameters

zs [array-like] Mole fractions of all components

Tcs [array-like] Critical temperatures of all components, [K]

Vcs [array-like] Critical volumes of all components, [m^3/mol]

taus [array-like of shape *zs* by *zs*] Interaction parameters, [-]

Returns

Tcm [float] Critical temperatures of the mixture, [K]

Notes

All parameters, even if zero, must be given to this function.

References

[1], [2]

Examples

butane/pentane/hexane 0.6449/0.2359/0.1192 mixture, exp: 450.22 K.

```
>>> Chueh_Prausnitz_Tc([0.6449, 0.2359, 0.1192], [425.12, 469.7, 507.6],
... [0.000255, 0.000313, 0.000371], [[0, 1.92681, 6.80358],
... [1.92681, 0, 1.89312], [ 6.80358, 1.89312, 0]])
450.122576472349
```

chemicals.critical.**Grieves_Thodos**(*zs*, *Tcs*, *Aijs*)

Calculates critical temperature of a mixture according to mixing rules in [1].

$$T_{cm} = \sum_i \frac{T_{ci}}{1 + (1/x_i)\sum_j A_{ij} x_j}$$

For a binary mxiture, this simplifies to:

$$T_{cm} = \frac{T_{c1}}{1 + (x_2/x_1)A_{12}} + \frac{T_{c2}}{1 + (x_1/x_2)A_{21}}$$

Parameters

zs [array-like] Mole fractions of all components

Tcs [array-like] Critical temperatures of all components, [K]

Aijs [array-like of shape *zs* by *zs*] Interaction parameters

Returns

Tcm [float] Critical temperatures of the mixture, [K]

Notes

All parameters, even if zero, must be given to this function. Giving 0s gives really bad results however.

References

[1], [2]

Examples

butane/pentane/hexane 0.6449/0.2359/0.1192 mixture, exp: 450.22 K.

```
>>> Grieves_Thodos([0.6449, 0.2359, 0.1192], [425.12, 469.7, 507.6], [[0, 1.2503, 1.
↪516], [0.799807, 0, 1.23843], [0.659633, 0.807474, 0]])
450.1839618758971
```

chemicals.critical.**modified_Wilson_Tc**(*zs*, *Tcs*, *Aijs*)

Calculates critical temperature of a mixture according to mixing rules in [1]. Equation

$$T_{cm} = \sum_i x_i T_{ci} + C \sum_i x_i \ln\left(x_i + \sum_j x_j A_{ij}\right) T_{ref}$$

For a binary mxiture, this simplifies to:

$$T_{cm} = x_1 T_{c1} + x_2 T_{c2} + C[x_1 \ln(x_1 + x_2 A_{12}) + x_2 \ln(x_2 + x_1 A_{21})]$$

Parameters

zs [float] Mole fractions of all components

Tcs [float] Critical temperatures of all components, [K]

Aijs [matrix] Interaction parameters

Returns

Tcm [float] Critical temperatures of the mixture, [K]

Notes

The equation and original article has been reviewed. [1] has 75 binary systems, and additional multicomponent mixture parameters. All parameters, even if zero, must be given to this function.

2rd example is from [2], for: butane/pentane/hexane 0.6449/0.2359/0.1192 mixture, exp: 450.22 K. Its result is identical to that calculated in the article.

References

[1], [2]

Examples

```
>>> modified_Wilson_Tc([0.6449, 0.2359, 0.1192], [425.12, 469.7, 507.6],
... [[0, 1.174450, 1.274390], [0.835914, 0, 1.21038],
... [0.746878, 0.80677, 0]])
450.03059668230316
```

1.5.7 Critical Volume of Mixtures

chemicals.critical.**Chueh_Prausnitz_Vc**(*zs*, *Vcs*, *nus*)

Calculates critical volume of a mixture according to mixing rules in [1] with an interaction parameter.

$$V_{cm} = \sum_i^n \theta_i V_{ci} + \sum_i^n \sum_j^n (\theta_i \theta_j \nu_{ij}) V_{ref} \theta = \frac{x_i V_{ci}^{2/3}}{\sum_{j=1}^n x_j V_{cj}^{2/3}}$$

Parameters

zs [float] Mole fractions of all components

Vcs [float] Critical volumes of all components, [m^3/mol]

nus [matrix] Interaction parameters, [cm^3/mol]

Returns

Vcm [float] Critical volume of the mixture, [m^3/mol]

Notes

All parameters, even if zero, must be given to this function. nu parameters are in cm^3/mol, but are converted to m^3/mol inside the function

References

[1], [2]

Examples

1-butanol/benzene 0.4271/0.5729 mixture, Vcm = 268.096 mL/mol.

```
>>> Chueh_Prausnitz_Vc([0.4271, 0.5729], [0.000273, 0.000256], [[0, 5.61847], [5.
↪61847, 0]])
0.00026620503424517445
```

chemicals.critical.**modified_Wilson_Vc**(*zs*, *Vcs*, *Aijs*)

Calculates critical volume of a mixture according to mixing rules in [1] with parameters. Equation

$$V_{cm} = \sum_i x_i V_{ci} + C \sum_i x_i \ln\left(x_i + \sum_j x_j A_{ij}\right) V_{ref}$$

For a binary mxiture, this simplifies to:

$$V_{cm} = x_1 V_{c1} + x_2 V_{c2} + C[x_1 \ln(x_1 + x_2 A_{12}) + x_2 \ln(x_2 + x_1 A_{21})]$$

Parameters

zs [float] Mole fractions of all components

Vcs [float] Critical volumes of all components, [m^3/mol]

Aijs [matrix] Interaction parameters, [cm^3/mol]

Returns

Vcm [float] Critical volume of the mixture, [m^3/mol]

Notes

The equation and original article has been reviewed. All parameters, even if zero, must be given to this function. C = -2500

All parameters, even if zero, must be given to this function. nu parameters are in cm^3/mol, but are converted to m^3/mol inside the function

References

[1], [2]

Examples

1-butanol/benzene 0.4271/0.5729 mixture, Vcm = 268.096 mL/mol.

```
>>> modified_Wilson_Vc([0.4271, 0.5729], [0.000273, 0.000256],
... [[0, 0.6671250], [1.3939900, 0]])
0.0002664335032706881
```

1.6 Dipole Moment (chemicals.dipole)

This module contains lookup functions for the property dipole moment.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.6.1 Lookup Functions

chemicals.dipole.**dipole_moment**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's dipole moment. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Preferred source is 'CCCBDB'. Considerable variation in reported data has found.

Parameters

CASRN [str] CASRN [-]

Returns

dipole [float] Dipole moment, [debye]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'CCCBDB', 'MULLER', or 'POLING', 'PSI4_2022A'. All valid values are also held in the list *dipole_all_methods*.

See also:

dipole_moment_methods

Notes

A total of three sources are available for this function. They are:

- 'CCCBDB', a series of critically evaluated data for compounds in [1], intended for use in predictive modeling.
- 'MULLER', a collection of data in a group-contribution scheme in [2].
- 'POLING', in the appendix in [3].
- 'PSI4_2022A', values computed using the Psi4 version 1.3.2 quantum chemistry software, with initialized positions from rdkit's EmbedMolecule method, the basis set 6-31G** and the method mp2 [4].

This function returns dipole moment in units of Debye. This is actually a non-SI unit; to convert to SI, multiply by 3.33564095198e-30 and its units will be in ampere*second^2 or equivalently and more commonly given, coulomb*second. The constant is the result of 1E-21/c, where c is the speed of light.

References

[1], [2], [3], [4]

Examples

```
>>> dipole_moment(CASRN='64-17-5')
1.44
```

chemicals.dipole.**dipole_moment_methods**(*CASRN*)

Return all methods available to obtain the dipole moment for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the dipole moment with the given inputs.

See also:

dipole_moment

chemicals.dipole.**dipole_moment_all_methods = ('CCCBDB', 'MULLER', 'POLING', 'PSI4_2022A')**

Tuple of method name keys. See the *dipole* for the actual references

1.7 DIPPR Fit Equations (chemicals.dippr)

This module contains implementations of various numered property equations used by the DIPPR, the Design Institude for Physical Property Research.

No actual data is included in this module; it is just functional implementations of the formulas and some of their derivatives/integrals.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.7.1 Equations

chemicals.dippr.**EQ100**(*T*, *A=0*, *B=0*, *C=0*, *D=0*, *E=0*, *F=0*, *G=0*, *order=0*)

DIPPR Equation # 100. Used in calculating the molar heat capacities of liquids and solids, liquid thermal conductivity, and solid density. All parameters default to zero. As this is a straightforward polynomial, no restrictions on parameters apply. Note that high-order polynomials like this may need large numbers of decimal places to avoid unnecessary error.

$$Y = A + BT + CT^2 + DT^3 + ET^4 + FT^5 + GT^6$$

Parameters

T [float] Temperature, [K]

A-G [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T, integral with respect to T, and integral over T with respect to T are computed as follows. All derivatives and integrals are easily computed with SymPy.

$$\frac{dY}{dT} = B + 2CT + 3DT^2 + 4ET^3 + 5FT^4 + 6GT^5$$

$$\int Y dT = AT + \frac{BT^2}{2} + \frac{CT^3}{3} + \frac{DT^4}{4} + \frac{ET^5}{5} + \frac{FT^6}{6} + \frac{GT^7}{7}$$

$$\int \frac{Y}{T} dT = A \ln(T) + BT + \frac{CT^2}{2} + \frac{DT^3}{3} + \frac{ET^4}{4} + \frac{FT^5}{5} + \frac{GT^6}{6}$$

References

[1]

Examples

Water liquid heat capacity; DIPPR coefficients normally listed in J/kmol/K.

```
>>> EQ100(300, 276370., -2090.1, 8.125, -0.014116, 0.0000093701)
75355.81000000003
```

chemicals.dippr.**EQ101**(*T*, *A*, *B*, *C=0.0*, *D=0.0*, *E=0.0*, *order=0*)

DIPPR Equation # 101. Used in calculating vapor pressure, sublimation pressure, and liquid viscosity. All 5 parameters are required. E is often an integer. As the model is exponential, a sufficiently high temperature will cause an OverflowError. A negative temperature (or just low, if fit poorly) may cause a math domain error.

$$Y = \exp\left(A + \frac{B}{T} + C \cdot \ln T + D \cdot T^E\right)$$

Parameters

T [float] Temperature, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for *n*, the *nth* derivative of the property is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float] Property [constant-specific]

Notes

This function is not integrable for either dT or Y/T dT.

$$\frac{dY}{dT} = \left(-\frac{B}{T^2} + \frac{C}{T} + \frac{DET^E}{T}\right) e^{A+\frac{B}{T}+C\log(T)+DT^E}$$

$$\frac{d^2Y}{dT^2} = \frac{\left(\frac{2B}{T} - C + DE^2T^E - DET^E + \left(-\frac{B}{T} + C + DET^E\right)^2\right) e^{A+\frac{B}{T}+C\log(T)+DT^E}}{T^2}$$

$$\frac{d^3Y}{dT^3} = \frac{\left(-\frac{6B}{T} + 2C + DE^3T^E - 3DE^2T^E + 2DET^E + \left(-\frac{B}{T} + C + DET^E\right)^3 + 3\left(-\frac{B}{T} + C + DET^E\right)\left(\frac{2B}{T} - C + DE^2\right.\right.}{T^3}$$

References

[1]

Examples

Water vapor pressure; DIPPR coefficients normally listed in Pa.

```
>>> EQ101(300, 73.649, -7258.2, -7.3037, 4.1653E-6, 2)
3537.44834545549
```

chemicals.dippr.**EQ102**(*T*, *A*, *B*, *C=0.0*, *D=0.0*, *order=0*)

DIPPR Equation # 102. Used in calculating vapor viscosity, vapor thermal conductivity, and sometimes solid heat capacity. High values of B raise an OverflowError. All 4 parameters are required. C and D are often 0.

$$Y = \frac{A \cdot T^B}{1 + \frac{C}{T} + \frac{D}{T^2}}$$

Parameters

T [float] Temperature, [K]

A-D [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T, integral with respect to T, and integral over T with respect to T are computed as follows. The first derivative is easily computed; the two integrals required Rubi to perform the integration.

$$\frac{dY}{dT} = \frac{ABT^B}{T\left(\frac{C}{T} + \frac{D}{T^2} + 1\right)} + \frac{AT^B\left(\frac{C}{T^2} + \frac{2D}{T^3}\right)}{\left(\frac{C}{T} + \frac{D}{T^2} + 1\right)^2}$$

$$\int Y\,dT = -\frac{2AT^{B+3}\,\text{hyp2f1}\left(1, B+3, B+4, -\frac{2T}{C-\sqrt{C^2-4D}}\right)}{(B+3)\left(C+\sqrt{C^2-4D}\right)\sqrt{C^2-4D}} + \frac{2AT^{B+3}\,\text{hyp2f1}\left(1, B+3, B+4, -\frac{2T}{C+\sqrt{C^2-4D}}\right)}{(B+3)\left(C-\sqrt{C^2-4D}\right)\sqrt{C^2-4D}}$$

$$\int \frac{Y}{T}\,dT = -\frac{2AT^{B+2}\,\text{hyp2f1}\left(1, B+2, B+3, -\frac{2T}{C+\sqrt{C^2-4D}}\right)}{(B+2)\left(C+\sqrt{C^2-4D}\right)\sqrt{C^2-4D}} + \frac{2AT^{B+2}\,\text{hyp2f1}\left(1, B+2, B+3, -\frac{2T}{C-\sqrt{C^2-4D}}\right)}{(B+2)\left(C-\sqrt{C^2-4D}\right)\sqrt{C^2-4D}}$$

References

[1]

Examples

Water vapor viscosity; DIPPR coefficients normally listed in Pa*s.

```
>>> EQ102(300, 1.7096E-8, 1.1146, 0, 0)
9.860384711890639e-06
```

chemicals.dippr.**EQ104**(*T*, *A*, *B*, *C=0.0*, *D=0.0*, *E=0.0*, *order=0*)

DIPPR Equation #104. Often used in calculating second virial coefficients of gases. All 5 parameters are required. C, D, and E are normally large values.

$$Y = A + \frac{B}{T} + \frac{C}{T^3} + \frac{D}{T^8} + \frac{E}{T^9}$$

Parameters

T [float] Temperature, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T, integral with respect to T, and integral over T with respect to T are computed as follows. All expressions can be obtained with SymPy readily.

$$\frac{dY}{dT} = -\frac{B}{T^2} - \frac{3C}{T^4} - \frac{8D}{T^9} - \frac{9E}{T^{10}}$$

$$\int Y dT = AT + B\ln(T) - \frac{1}{56T^8}\left(28CT^6 + 8DT + 7E\right)$$

$$\int \frac{Y}{T} dT = A\ln(T) - \frac{1}{72T^9}\left(72BT^8 + 24CT^6 + 9DT + 8E\right)$$

References

[1]

Examples

Water second virial coefficient; DIPPR coefficients normally dimensionless.

```
>>> EQ104(300, 0.02222, -26.38, -16750000, -3.894E19, 3.133E21)
-1.1204179007265156
```

chemicals.dippr.**EQ105**(*T*, *A*, *B*, *C*, *D*, *order=0*)

DIPPR Equation #105. Often used in calculating liquid molar density. All 4 parameters are required. C is sometimes the fluid's critical temperature.

$$Y = \frac{A}{B^{1+\left(1-\frac{T}{C}\right)^D}}$$

Parameters

T [float] Temperature, [K]

A-D [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, 2, and 3, that derivative of the property is returned; No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float] Property [constant-specific]

Notes

This expression can be integrated in terms of the incomplete gamma function for dT, however nans are the only output from that function. For Y/T dT no integral could be found.

$$\frac{dY}{dT} = \frac{AB^{-\left(1-\frac{T}{C}\right)^D-1}D\left(1-\frac{T}{C}\right)^D\log\left(B\right)}{C\left(1-\frac{T}{C}\right)}$$

$$\frac{d^2Y}{dT^2} = \frac{AB^{-\left(1-\frac{T}{C}\right)^D-1}D\left(1-\frac{T}{C}\right)^D\left(D\left(1-\frac{T}{C}\right)^D\log\left(B\right)-D+1\right)\log\left(B\right)}{C^2\left(1-\frac{T}{C}\right)^2}$$

$$\frac{d^3Y}{dT^3} = \frac{AB^{-\left(1-\frac{T}{C}\right)^D-1}D\left(1-\frac{T}{C}\right)^D\left(D^2\left(1-\frac{T}{C}\right)^{2D}\log\left(B\right)^2-3D^2\left(1-\frac{T}{C}\right)^D\log\left(B\right)+D^2+3D\left(1-\frac{T}{C}\right)^D\log\left(B\right)-3D\right.}{C^3\left(1-\frac{T}{C}\right)^3}$$

References

[1]

Examples

Hexane molar density; DIPPR coefficients normally in kmol/m^3.

```
>>> EQ105(300., 0.70824, 0.26411, 507.6, 0.27537)
7.593170096339237
```

chemicals.dippr.**EQ106**(*T*, *Tc*, *A*, *B*, *C=0.0*, *D=0.0*, *E=0.0*, *order=0*)

DIPPR Equation #106. Often used in calculating liquid surface tension, and heat of vaporization. Only parameters A and B parameters are required; many fits include no further parameters. Critical temperature is also required.

$$Y = A(1 - T_r)^{B + CT_r + DT_r^2 + ET_r^3}$$

$$Tr = \frac{T}{Tc}$$

Parameters

T [float] Temperature, [K]

Tc [float] Critical temperature, [K]

A-D [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, 2, and 3, that derivative of the property is returned; No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float] Property [constant-specific]

Notes

This form is used by Yaws with only the parameters *A* and *B*.

The integral could not be found, but the integral over T actually could, again in terms of hypergeometric functions.

$$\frac{dY}{dT} = A\left(-\frac{T}{T_c} + 1\right)^{B + \frac{CT}{T_c} + \frac{DT^2}{T_c^2} + \frac{eT^3}{T_c^3}} \left(\left(\frac{C}{T_c} + \frac{2DT}{T_c^2} + \frac{3eT^2}{T_c^3}\right)\log\left(-\frac{T}{T_c} + 1\right) - \frac{B + \frac{CT}{T_c} + \frac{DT^2}{T_c^2} + \frac{eT^3}{T_c^3}}{T_c\left(-\frac{T}{T_c} + 1\right)}\right)$$

$$\frac{d^2Y}{dT^2} = \frac{A\left(-\frac{T}{T_c} + 1\right)^{B + \frac{CT}{T_c} + \frac{DT^2}{T_c^2} + \frac{eT^3}{T_c^3}} \left(2\left(D + \frac{3eT}{T_c}\right)\log\left(-\frac{T}{T_c} + 1\right) + \left(\left(C + \frac{2DT}{T_c} + \frac{3eT^2}{T_c^2}\right)\log\left(-\frac{T}{T_c} + 1\right) + \frac{B + \frac{CT}{T_c} + \frac{DT^2}{T_c^2}}{\frac{T}{T_c} - 1}\right.\right.}{T_c^2}$$

$$\frac{d^3Y}{dT^3} = \frac{A\left(-\frac{T}{T_c} + 1\right)^{B + \frac{CT}{T_c} + \frac{DT^2}{T_c^2} + \frac{eT^3}{T_c^3}} \left(\frac{6\left(D + \frac{3eT}{T_c}\right)}{\frac{T}{T_c} - 1} + \left(\left(C + \frac{2DT}{T_c} + \frac{3eT^2}{T_c^2}\right)\log\left(-\frac{T}{T_c} + 1\right) + \frac{B + \frac{CT}{T_c} + \frac{DT^2}{T_c^2} + \frac{eT^3}{T_c^3}}{\frac{T}{T_c} - 1}\right)^3 + 3\left(\left(C\right.\right.\right.}$$

References

[1]

Examples

Water surface tension; DIPPR coefficients normally in Pa*s.

```
>>> EQ106(300, 647.096, 0.17766, 2.567, -3.3377, 1.9699)
0.07231499373541
```

chemicals.dippr.**EQ107**(*T*, *A=0*, *B=0*, *C=0*, *D=0*, *E=0*, *order=0*)

DIPPR Equation #107. Often used in calculating ideal-gas heat capacity. All 5 parameters are required. Also called the Aly-Lee equation.

$$Y = A + B\left[\frac{C/T}{\sinh(C/T)}\right]^2 + D\left[\frac{E/T}{\cosh(E/T)}\right]^2$$

Parameters

T [float] Temperature, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T, integral with respect to T, and integral over T with respect to T are computed as follows. The derivative is obtained via SymPy; the integrals from Wolfram Alpha.

$$\frac{dY}{dT} = \frac{2BC^3\cosh\left(\frac{C}{T}\right)}{T^4\sinh^3\left(\frac{C}{T}\right)} - \frac{2BC^2}{T^3\sinh^2\left(\frac{C}{T}\right)} + \frac{2DE^3\sinh\left(\frac{E}{T}\right)}{T^4\cosh^3\left(\frac{E}{T}\right)} - \frac{2DE^2}{T^3\cosh^2\left(\frac{E}{T}\right)}$$

$$\int Y\,dT = AT + \frac{BC}{\tanh\left(\frac{C}{T}\right)} - DE\tanh\left(\frac{E}{T}\right)$$

$$\int \frac{Y}{T}dT = A\ln(T) + \frac{BC}{T\tanh\left(\frac{C}{T}\right)} - B\ln\left(\sinh\left(\frac{C}{T}\right)\right) - \frac{DE}{T}\tanh\left(\frac{E}{T}\right) + D\ln\left(\cosh\left(\frac{E}{T}\right)\right)$$

References

[1], [2]

Examples

Water ideal gas molar heat capacity; DIPPR coefficients normally in J/kmol/K

```
>>> EQ107(300., 33363., 26790., 2610.5, 8896., 1169.)
33585.90452768923
```

chemicals.dippr.**EQ114**(*T*, *Tc*, *A*, *B*, *C*, *D*, *order=0*)

DIPPR Equation #114. Rarely used, normally as an alternate liquid heat capacity expression. All 4 parameters are required, as well as critical temperature.

$$Y = \frac{A^2}{\tau} + B - 2AC\tau - AD\tau^2 - \frac{1}{3}C^2\tau^3 - \frac{1}{2}CD\tau^4 - \frac{1}{5}D^2\tau^5$$

$$\tau = 1 - \frac{T}{Tc}$$

Parameters

T [float] Temperature, [K]

Tc [float] Critical temperature, [K]

A-D [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T, integral with respect to T, and integral over T with respect to T are computed as follows. All expressions can be obtained with SymPy readily.

$$\frac{dY}{dT} = \frac{A^2}{T_c\left(-\frac{T}{T_c}+1\right)^2} + \frac{2A}{T_c}C + \frac{2A}{T_c}D\left(-\frac{T}{T_c}+1\right) + \frac{C^2}{T_c}\left(-\frac{T}{T_c}+1\right)^2 + \frac{2C}{T_c}D\left(-\frac{T}{T_c}+1\right)^3 + \frac{D^2}{T_c}\left(-\frac{T}{T_c}+1\right)^4$$

$$\int Y dT = -A^2 T_c \ln\left(T - T_c\right) + \frac{D^2T^6}{30T_c^5} - \frac{T^5}{10T_c^4}\left(CD + 2D^2\right) + \frac{T^4}{12T_c^3}\left(C^2 + 6CD + 6D^2\right) - \frac{T^3}{3T_c^2}\left(AD + C^2 + 3CD + 2D\right.$$

$$\int \frac{Y}{T} dT = -A^2 \ln\left(T + \frac{-60A^2T_c + 60ACT_c + 30ADT_c - 30BT_c + 10C^2T_c + 15CDT_c + 6D^2T_c}{60A^2 - 60AC - 30AD + 30B - 10C^2 - 15CD - 6D^2}\right) + \frac{D^2T^5}{25T_c^5} - \frac{T^4}{8T_c^4}\left(CD\right.$$

Strictly speaking, the integral over T has an imaginary component, but only the real component is relevant and the complex part discarded.

References

[1]

Examples

Hydrogen liquid heat capacity; DIPPR coefficients normally in J/kmol/K.

```
>>> EQ114(20, 33.19, 66.653, 6765.9, -123.63, 478.27)
19423.948911676463
```

chemicals.dippr.**EQ115**(*T*, *A*, *B*, *C=0*, *D=0*, *E=0*, *order=0*)

DIPPR Equation #115. No major uses; has been used as an alternate liquid viscosity expression, and as a model for vapor pressure. Only parameters A and B are required.

$$Y = \exp\left(A + \frac{B}{T} + C\ln T + DT^2 + \frac{E}{T^2}\right)$$

Parameters

T [float] Temperature, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, 2, and 3, that derivative of the property is returned; No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float] Property [constant-specific]

Notes

No coefficients found for this expression. This function is not integrable for either dT or Y/T dT.

$$\frac{dY}{dT} = \left(-\frac{B}{T^2} + \frac{C}{T} + 2DT - \frac{2E}{T^3}\right) e^{A+\frac{B}{T}+C\log(T)+DT^2+\frac{E}{T^2}}$$

$$\frac{d^2Y}{dT^2} = \left(\frac{2B}{T^3} - \frac{C}{T^2} + 2D + \frac{6E}{T^4} + \left(\frac{B}{T^2} - \frac{C}{T} - 2DT + \frac{2E}{T^3}\right)^2\right) e^{A+\frac{B}{T}+C\log(T)+DT^2+\frac{E}{T^2}}$$

$$\frac{d^3Y}{dT^3} = -\left(3\left(\frac{2B}{T^3} - \frac{C}{T^2} + 2D + \frac{6E}{T^4}\right)\left(\frac{B}{T^2} - \frac{C}{T} - 2DT + \frac{2E}{T^3}\right) + \left(\frac{B}{T^2} - \frac{C}{T} - 2DT + \frac{2E}{T^3}\right)^3 + \frac{2\left(\frac{3B}{T} - C + \frac{12E}{T^2}\right)}{T^3}\right) e$$

References

[1]

chemicals.dippr.**EQ116**(*T*, *Tc*, *A*, *B*, *C*, *D*, *E*, *order=0*)

DIPPR Equation #116. Used to describe the molar density of water fairly precisely; no other uses listed. All 5 parameters are needed, as well as the critical temperature.

$$Y = A + B\tau^{0.35} + C\tau^{2/3} + D\tau + E\tau^{4/3}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature, [K]

Tc [float] Critical temperature, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T and integral with respect to T are computed as follows. The integral divided by T with respect to T has an extremely complicated (but still elementary) integral which can be read from the source. It was computed with Rubi; the other expressions can readily be obtained with SymPy.

$$\frac{dY}{dT} = -\frac{7B}{20T_c\left(-\frac{T}{T_c}+1\right)^{\frac{13}{20}}} - \frac{2C}{3T_c\sqrt[3]{-\frac{T}{T_c}+1}} - \frac{D}{T_c} - \frac{4E}{3T_c}\sqrt[3]{-\frac{T}{T_c}+1}$$

$$\int Y\,dT = AT - \frac{20B}{27}T_c\left(-\frac{T}{T_c}+1\right)^{\frac{27}{20}} - \frac{3C}{5}T_c\left(-\frac{T}{T_c}+1\right)^{\frac{5}{3}} + D\left(-\frac{T^2}{2T_c}+T\right) - \frac{3E}{7}T_c\left(-\frac{T}{T_c}+1\right)^{\frac{7}{3}}$$

References

[1]

Examples

Water liquid molar density; DIPPR coefficients normally in kmol/m^3.

```
>>> EQ116(300., 647.096, 17.863, 58.606, -95.396, 213.89, -141.26)
55.17615446406527
```

chemicals.dippr.**EQ127**(*T*, *A*, *B*, *C*, *D*, *E*, *F*, *G*, *order=0*)

DIPPR Equation #127. Rarely used, and then only in calculating ideal-gas heat capacity. All 7 parameters are required.

$$Y = A + B\left[\frac{\left(\frac{C}{T}\right)^2\exp\left(\frac{C}{T}\right)}{\left(\exp\frac{C}{T}-1\right)^2}\right] + D\left[\frac{\left(\frac{E}{T}\right)^2\exp\left(\frac{E}{T}\right)}{\left(\exp\frac{E}{T}-1\right)^2}\right] + F\left[\frac{\left(\frac{G}{T}\right)^2\exp\left(\frac{G}{T}\right)}{\left(\exp\frac{G}{T}-1\right)^2}\right]$$

Parameters

T [float] Temperature, [K]

A-G [float] Parameter for the equation; chemical and property specific [-]

order [int, optional] Order of the calculation. 0 for the calculation of the result itself; for 1, the first derivative of the property is returned, for -1, the indefinite integral of the property with respect to temperature is returned; and for -1j, the indefinite integral of the property divided by temperature with respect to temperature is returned. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

Y [float]

Property [constant-specific; if order == 1, property/K; if order == -1, property*K; if order == -1j, unchanged from default]

Notes

The derivative with respect to T, integral with respect to T, and integral over T with respect to T are computed as follows. All expressions can be obtained with SymPy readily.

$$\frac{dY}{dT} = -\frac{BC^3e^{\frac{C}{T}}}{T^4\left(e^{\frac{C}{T}}-1\right)^2} + \frac{2BC^3e^{\frac{2C}{T}}}{T^4\left(e^{\frac{C}{T}}-1\right)^3} - \frac{2BC^2e^{\frac{C}{T}}}{T^3\left(e^{\frac{C}{T}}-1\right)^2} - \frac{DE^3e^{\frac{E}{T}}}{T^4\left(e^{\frac{E}{T}}-1\right)^2} + \frac{2DE^3e^{\frac{2E}{T}}}{T^4\left(e^{\frac{E}{T}}-1\right)^3} - \frac{2DE^2e^{\frac{E}{T}}}{T^3\left(e^{\frac{E}{T}}-1\right)^2} - \frac{FG^3}{T^4\left(e^{\frac{G}{T}}\right.}$$

$$\int Y dT = AT + \frac{BC^2}{Ce^{\frac{C}{T}} - C} + \frac{DE^2}{Ee^{\frac{E}{T}} - E} + \frac{FG^2}{Ge^{\frac{G}{T}} - G}$$

$$\int \frac{Y}{T} dT = A\ln(T) + BC^2\left(\frac{1}{CTe^{\frac{C}{T}} - CT} + \frac{1}{CT} - \frac{1}{C^2}\ln\left(e^{\frac{C}{T}} - 1\right)\right) + DE^2\left(\frac{1}{ETe^{\frac{E}{T}} - ET} + \frac{1}{ET} - \frac{1}{E^2}\ln\left(e^{\frac{E}{T}} - 1\right)\right)$$

References

[1]

Examples

Ideal gas heat capacity of methanol; DIPPR coefficients normally in J/kmol/K

```
>>> EQ127(20., 3.3258E4, 3.6199E4, 1.2057E3, 1.5373E7, 3.2122E3, -1.5318E7, 3.
↪2122E3)
33258.0
```

1.7.2 Jacobians (for fitting)

chemicals.dippr.**EQ101_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *D*, *E*)

Compute and return the Jacobian of the property predicted by DIPPR Equation # 101 with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

Returns

jac [list[list[float, 5], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.dippr.**EQ102_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *D*)

Compute and return the Jacobian of the property predicted by DIPPR Equation # 102 with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

A-D [float] Parameter for the equation; chemical and property specific [-]

Returns

jac [list[list[float, 4], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.dippr.**EQ105_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *D*)

Compute and return the Jacobian of the property predicted by DIPPR Equation # 105 with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

A-D [float] Parameter for the equation; chemical and property specific [-]

Returns

jac [list[list[float, 4], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.dippr.**EQ106_fitting_jacobian**(*Ts*, *Tc*, *A*, *B*, *C*, *D*, *E*)

Compute and return the Jacobian of the property predicted by DIPPR Equation # 106 with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

Tc [float] Critical temperature, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

Returns

jac [list[list[float, 5], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.dippr.**EQ107_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *D*, *E*)

Compute and return the Jacobian of the property predicted by DIPPR Equation # 107 with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

A-E [float] Parameter for the equation; chemical and property specific [-]

Returns

jac [list[list[float, 5], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

1.8 Periodic Table (chemicals.elements)

This module contains a complete periodic table, routines for working with chemical formulas, computing molecular weight, computing mass fractions and atom fractions, and assorted other tasks.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.8.1 Periodic Table and Elements

chemicals.elements.**periodic_table = <chemicals.elements.PeriodicTable object>**

Single instance of the PeriodicTable class. Use this, not the PeriodicTable class directly.

A brief overview of using the periodic table and its elements:

```
>>> periodic_table.Na
<Element Sodium (Na), number 11, MW=22.98977>
>>> periodic_table.U.MW
238.02891
>>> periodic_table['Th'].CAS
'7440-29-1'
>>> periodic_table.lead.protons
82
>>> periodic_table['7440-57-5'].symbol
'Au'
>>> len(periodic_table)
118
>>> 'gold' in periodic_table
True
>>> periodic_table.He.protons, periodic_table.He.neutrons, periodic_table.He.
↪electrons # Standard number of protons, neutrons, electrons
(2, 2, 2)
>>> periodic_table.He.phase # Phase of the element in the standard state
'g'
>>> periodic_table.He.Hf # Heat of formation in standard state in J/mol - by␣
↪definition 0
0.0
>>> periodic_table.He.S0 # Absolute entropy (J/(mol*K) in standard state - non-zero)
126.2
>>> periodic_table.Kr.block, periodic_table.Kr.period, periodic_table.Kr.group
('p', 4, 18)
>>> periodic_table.Rn.InChI
'Rn'
>>> periodic_table.Rn.smiles
'[Rn]'
>>> periodic_table.Pu.number
94
```

(continues on next page)

(continued from previous page)

```
>>> periodic_table.Pu.PubChem
23940
>>> periodic_table.Bi.InChI_key
'JCXGWMGPZLAOME-UHFFFAOYSA-N'
```

class chemicals.elements.**Element**(*number*, *symbol*, *name*, *MW*, *CAS*, *AReneg*, *rcov*, *rvdw*, *maxbonds*, *elneg*, *ionization*, *elaffinity*, *period*, *group*, *PubChem*, *phase*, *Hf*, *S0*, *InChI_key=None*)

Class for storing data on chemical elements. Supports most common properties. If a property is not available, it is set to None.

The elements are created automatically and should be accessed via the *periodic_table* interface.

Attributes

number [int] Atomic number, [-]

name [str] name, [-]

symbol [str] Elemental symbol, [-]

MW [float] Molecular weight, [g/mol]

CAS [str] CAS number, [-]

period [str] Period in the periodic table, [-]

group [str] Group in the periodic table, [-]

block [str] Which block of the periodic table the element is in.

AReneg [float] Allred and Rochow electronegativity, [-]

rcov [float] Covalent radius, [Angstrom]

rvdw [float] Van der Waals radius, [Angstrom]

maxbonds [float] Maximum valence of a bond with this element, [-]

elneg [float] Pauling electronegativity, [-]

ionization [float] Ionization potential, [eV]

ionization [float] elaffinity affinity, [eV]

protons [int] The number of protons of the element.

electrons [int] The number of electrons of the element.

InChI [str] The InChI identifier of the element.

InChI_key [str] 25-character hash of the compound's InChI, [-]

smiles [str] The SMILES identification string of the element.

PubChem [int] PubChem Compound identifier (CID) of the chemical, [-]

phase [str] Standard state at 1 atm and 298.15 K, [-]

Hf [float] Enthalpy of formation of the element in its standard state (0 by definition), [J/mol]

S0 [float] Standard absolute entropy of the element in its standard state (1 bar, 298.15 K), [J/mol/K]

class chemicals.elements.**PeriodicTable**(*elements*)
Periodic Table object for use in dealing with elements.

As there is only one periodic table of elements, this is automatically initialized into the object *periodic_table*; there is no need to construct a new instance of this class.

Parameters

elements [list[Element]] List of Element objects, [-]

See also:

periodic_table

Element

Notes

Can be checked to sese if an element in in this, can be iterated over, and as a current length of 118 elements.

References

[1]

1.8.2 Working with Formulas

chemicals.elements.**simple_formula_parser**(*formula*)
Basic formula parser, primarily for obtaining element counts from formulas as formated in PubChem. Handles formulas with integer or decimal counts (with period separator), but no brackets, no hydrates, no charges, no isotopes, and no group multipliers.

Strips charges from the end of a formula first. Accepts repeated chemical units. Performs no sanity checking that elements are actually elements. As it uses regular expressions for matching, errors are mostly just ignored.

Parameters

formula [str] Formula string, very simply formats only.

Returns

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

Notes

Inspiration taken from the thermopyl project, at https://github.com/choderalab/thermopyl.

Examples

```
>>> simple_formula_parser('CO2')
{'C': 1, 'O': 2}
```

chemicals.elements.**nested_formula_parser**(*formula*, *check=True*)

Improved formula parser which handles braces and their multipliers, as well as rational element counts.

Strips charges from the end of a formula first. Accepts repeated chemical units. Performs no sanity checking that elements are actually elements. As it uses regular expressions for matching, errors are mostly just ignored.

Parameters

formula [str] Formula string, very simply formats only.

check [bool] If *check* is True, a simple check will be performed to determine if a formula is not a formula and an exception will be raised if it is not, [-]

Returns

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

Notes

Inspired by the approach taken by CrazyMerlyn on a reddit DailyProgrammer challenge, at https://www.reddit.com/r/dailyprogrammer/comments/6eerfk/20170531_challenge_317_intermediate_counting/

Examples

```
>>> nested_formula_parser('Pd(NH3)4.0001+2')
{'Pd': 1, 'N': 4.0001, 'H': 12.0003}
```

chemicals.elements.**charge_from_formula**(*formula*)

Basic formula parser to determine the charge from a formula - given that the charge is already specified as one element of the formula.

Performs no sanity checking that elements are actually elements.

Parameters

formula [str] Formula string, very simply formats only, ending in one of '+x', '-x', n*'+', or n*'-' or any of them surrounded by brackets but always at the end of a formula.

Returns

charge [int] Charge of the molecule, [faraday]

Examples

```
>>> charge_from_formula('Br3-')
-1
>>> charge_from_formula('Br3(-)')
-1
```

chemicals.elements.**serialize_formula**(*formula*)

Basic formula serializer to construct a consistently-formatted formula. This is necessary for handling user-supplied formulas, which are not always well formatted.

Performs no sanity checking that elements are actually elements.

Parameters

formula [str] Formula string as parseable by the method nested_formula_parser, [-]

Returns

formula [str] A consistently formatted formula to describe a molecular formula, [-]

Examples

```
>>> serialize_formula('Pd(NH3)4+3')
'H12N4Pd+3'
```

chemicals.elements.**atoms_to_Hill**(*atoms*)

Determine the Hill formula of a compound, given a dictionary of its atoms and their counts, in the format {symbol: count}.

Parameters

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

Returns

Hill_formula [str] Hill formula, [-]

Notes

The Hill system is as follows:

If the chemical has 'C' in it, this is listed first, and then if it has 'H' in it as well as 'C', then that goes next. All elements are sorted alphabetically afterwards, including 'H' if 'C' is not present. All elements are followed by their count, unless it is 1.

References

[1]

Examples

```
>>> atoms_to_Hill({'H': 5, 'C': 2, 'Br': 1})
'C2H5Br'
```

1.8.3 Working with Parsed Formulas

chemicals.elements.**molecular_weight**(*atoms*)

Calculates molecular weight of a molecule given a dictionary of its atoms and their counts, in the format {symbol: count}.

$$MW = \sum_i n_i MW_i$$

Parameters

atoms [dict] Dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

Returns

MW [float] Calculated molecular weight [g/mol]

Notes

Elemental data is from rdkit, with CAS numbers added. An exception is raised if an incorrect element symbol is given. Elements up to 118 are supported, as are deutreium and tritium.

References

[1]

Examples

```
>>> molecular_weight({'H': 12, 'C': 20, 'O': 5}) # DNA
332.30628
```

chemicals.elements.**similarity_variable**(*atoms*, *MW=None*)

Calculates the similarity variable of an compound, as defined in [1]. Currently only applied for certain heat capacity estimation routines.

$$\alpha = \frac{N}{MW} = \frac{\sum_i n_i}{\sum_i n_i MW_i}$$

Parameters

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

MW [float, optional] Molecular weight, [g/mol]

Returns

similarity_variable [float] Similarity variable as defined in [1], [mol/g]

Notes

Molecular weight is optional, but speeds up the calculation slightly. It is calculated using the function *molecular_weight* if not specified.

References

[1]

Examples

```
>>> similarity_variable({'H': 32, 'C': 15})
0.2212654140784498
```

chemicals.elements.**index_hydrogen_deficiency**(*atoms*)

Calculate the index of hydrogen deficiency of a compound, given a dictionary of its atoms and their counts, in the format {symbol: count}.

Parameters

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

Returns

HDI [float] Hydrogen deficiency index, [-]

Notes

The calculation is according to:

$$\text{IDH} = 0.5\left(2C + 2 + N - H - X + 0O\right)$$

where X is the number of halogen atoms. The number of oxygen atoms does not impact this calculation.

References

[1]

Examples

Agelastatin A:

```
>>> index_hydrogen_deficiency({'C': 12, 'H': 13, 'Br': 1, 'N': 4, 'O': 3})
8.0
```

chemicals.elements.**atom_fractions**(*atoms*)

Calculates the atomic fractions of each element in a compound, given a dictionary of its atoms and their counts, in the format {symbol: count}.

$$a_i = \frac{n_i}{\sum_i n_i}$$

Parameters

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

Returns

afracs [dict] dictionary of atomic fractions of individual atoms, indexed by symbol with proper capitalization, [-]

Notes

No actual data on the elements is used, so incorrect or custom compounds would not raise an error.

References

[1]

Examples

```
>>> atom_fractions({'H': 12, 'C': 20, 'O': 5})
{'H': 0.32432432432432434, 'C': 0.5405405405405406, 'O': 0.1351351351351514}
```

chemicals.elements.**mass_fractions**(*atoms*, *MW=None*)

Calculates the mass fractions of each element in a compound, given a dictionary of its atoms and their counts, in the format {symbol: count}.

$$w_i = \frac{n_i MW_i}{\sum_i n_i MW_i}$$

Parameters

atoms [dict] Dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

MW [float, optional] Molecular weight, [g/mol]

Returns

mfracs [dict] Dictionary of mass fractions of individual atoms, indexed by symbol with proper capitalization, [-]

Notes

Molecular weight is optional, but speeds up the calculation slightly. It is calculated using the function *molecular_weight* if not specified.

Elemental data is from rdkit, with CAS numbers added. An exception is raised if an incorrect element symbol is given. Elements up to 118 are supported.

References

[1]

Examples

```
>>> mass_fractions({'H': 12, 'C': 20, 'O': 5})
{'H': 0.03639798802478244, 'C': 0.7228692758981262, 'O': 0.24073273607709128}
```

chemicals.elements.**mixture_atomic_composition**(*atomss*, *zs*)

Simple function to calculate the atomic average composition of a mixture, using the mole fractions of each species and their own atomic compositions.

Parameters

atomss [list[dict[(str, int)]]] List of dictionaries of atomic compositions, [-]

zs [list[float]] Mole fractions of each component, [-]

Returns

atoms [dict[(str, int)]] Atomic composition

Examples

```
>>> mixture_atomic_composition([{'O': 2}, {'N': 1, 'O': 2}, {'C': 1, 'H': 4}], [0.
↪95, 0.025, .025])
{'O': 1.95, 'N': 0.025, 'C': 0.025, 'H': 0.1}
```

chemicals.elements.**mixture_atomic_composition_ordered**(*atomss*, *zs*)

Simple function to calculate the atomic average composition of a mixture, using the mole fractions of each species and their own atomic compositions. Returns the result as a sorted list with atomic numbers from low to high.

Parameters

atomss [list[dict[(str, int)]]] List of dictionaries of atomic compositions, [-]

zs [list[float]] Mole fractions of each component; this can also be a molar flow rate and then the *abundances* will be flows, [-]

Returns

abundances [list[float]] Number of atoms of each element per mole of the feed, [-]

atom_keys [list[str]] Atomic elements, sorted from lowest atomic number to highest

Notes

Useful to ensure a matrix order is consistent in multiple steps.

Examples

```
>>> mixture_atomic_composition_ordered([{'O': 2}, {'N': 1, 'O': 2}, {'C': 1, 'H': 4}
↪], [0.95, 0.025, .025])
([0.1, 0.025, 0.025, 1.95], ['H', 'C', 'N', 'O'])
```

chemicals.elements.**atom_matrix**(*atomss*, *atom_IDs=None*)

Simple function to create a matrix of elements in each compound, where each row has the same elements.

Parameters

atomss [list[dict[(str, int)]]] List of dictionaries of atomic compositions, [-]

atom_IDs [list[str], optional] Optionally, a subset (or simply ordered differently) of elements to consider, [-]

Returns

matrix [list[list[float]]] The number of each element in each compound as a matrix, indexed as [compound][element], [-]

Examples

```
>>> atom_matrix([{'C': 1, 'H': 4}, {'C': 2, 'H': 6}, {'N': 2}, {'O': 2}, {'H': 2, 'O
↪': 1}, {'C': 1, 'O': 2}])
[[4, 1, 0.0, 0.0], [6, 2, 0.0, 0.0], [0.0, 0.0, 2, 0.0], [0.0, 0.0, 0.0, 2], [2, 0.
↪0, 0.0, 1], [0.0, 1, 0.0, 2]]
```

1.9 Environmental Properties (chemicals.environment)

This module contains lookup functions for three important environmental properties - Global Warming Potential, Ozone Depletion Potential, and octanol-water partition coefficient.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.9.1 Global Warming Potential

chemicals.environment.**GWP**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's Global Warming Potential, relative to CO2. Lookup is based on CASRNs.

There are three sources of data:

- IPCC Fifth Assessment Report (AR5) from 2014 [2]
- IPCC Fourth Assessment Report (AR4) from 2007 [1]
- IPCC Second Assesment Report or (SAR) from 1995 [1]

This function returns the GWP for the 20yr outlook from the AR5 by default.

Parameters

CASRN [str] CASRN [-]

Returns

GWP [float] Global warming potential, [(impact/mass chemical)/(impact/mass CO2)]

Other Parameters

method [string, optional] The method name to use. Accepted methods are ('IPCC (2014) 100yr', 'IPCC (2014) 20yr', 'IPCC (2007) 100yr', 'IPCC (2007) 20yr', 'IPCC (2007) 500yr', 'IPCC (1995) 100yr'). All valid values are also held in the variable *GWP_all_methods*.

See also:

GWP_methods

Notes

"Fossil methane" is included in the IPCC reports to take into account different isotopic composition, but as that has the same CAS number it is not included in this function.

Six of the entries in [2] are actually duplicates; the entries with data similar to more recent data [3] were prefered.

References

[1], [2], [3]

Examples

Methane, 20-yr outlook AR5

```
>>> GWP(CASRN='74-82-8')
84.0
```

Methane, specifying the default method explicitly (this is recommended the default data source may be updated in the future)

```
>>> GWP(CASRN='74-82-8', method='IPCC (2014) 100yr')
28.0
```

Methane, 20-year values from 1995 and 2007

```
>>> (GWP(CASRN='74-82-8', method='IPCC (1995) 100yr'), GWP(CASRN='74-82-8', method=
↪'IPCC (2007) 100yr'))
(21.0, 25.0)
```

chemicals.environment.**GWP_methods**(*CASRN*)

Return all methods available to obtain GWP for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain GWP with the given inputs.

See also:

GWP

chemicals.environment.**GWP_all_methods = ('IPCC (2014) 100yr', 'IPCC (2014) 20yr', 'IPCC (2007) 100yr', 'IPCC (2007) 20yr', 'IPCC (2007) 500yr', 'IPCC (1995) 100yr')**

Tuple of method name keys. See the *GWP* for the actual references

1.9.2 Ozone Depletion Potential

chemicals.environment.**ODP**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's Ozone Depletion Potential, relative to CFC-11 (trichlorofluoromethane). Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Returns the ODP of a chemical according to [2] when a method is not specified. If a range is provided in [2], the highest value is returned.

Parameters

CASRN [str] CASRN [-]

Returns

ODP [float or str] Ozone Depletion potential, [(impact/mass chemical)/(impact/mass CFC-11)]; if method selected has *string* in it, this will be returned as a string regardless of if a range is given or a number

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'ODP2 Max', 'ODP2 Min', 'ODP2 string', 'ODP2 logarithmic average', and methods for older values are 'ODP1 Max', 'ODP1 Min', 'ODP1 string', and 'ODP1 logarithmic average'. All valid values are also held in the list ODP_methods.

Notes

Values are tabulated only for a small number of halogenated hydrocarbons, responsible for the largest impact. The original values of ODP as defined in the Montreal Protocol are also available, as methods with the *ODP1* prefix.

All values are somewhat emperical, as actual reaction rates of chemicals with ozone depend on temperature which depends on latitude, longitude, time of day, weather, and the concentrations of other pollutants.

All data is from [1]. Several mixtures listed in [1] are not included here as they are not pure species. Methods for values in [2] are 'ODP2 Max', 'ODP2 Min', 'ODP2 string', 'ODP2 logarithmic average', and methods for older values are 'ODP1 Max', 'ODP1 Min', 'ODP1 string', and 'ODP1 logarithmic average'.

References

[1], [2]

Examples

Dichlorotetrafluoroethane, according to [2].

```
>>> ODP(CASRN='76-14-2')
0.58
```

chemicals.environment.**ODP_methods**(*CASRN*)

Return all methods available to obtain ODP for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain ODP with the given inputs.

See also:

ODP

chemicals.environment.**ODP_all_methods = ('ODP2 Max', 'ODP1 Max', 'ODP2 logarithmic average', 'ODP1 logarithmic average', 'ODP2 Min', 'ODP1 Min', 'ODP2 string', 'ODP1 string')**

Tuple of method name keys. See the *ODP* for the actual references

1.9.3 Octanol-Water Partition Coefficient

chemicals.environment.**logP**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's octanol-water partition coefficient. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

logP [float] Octanol-water partition coefficient, [-]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'SYRRES', 'CRC', and 'WIKIDATA'. All valid values are also held in the list logP_methods.

Notes

Although matimatically this could be expressed with a logarithm in any base, reported values are published using a base 10 logarithm.

$$\log_{10} P_{oct/wat} = \log_{10}\left(\frac{[solute]^{un-ionized}_{octanol}}{[solute]^{un-ionized}_{water}}\right)$$

References

[1], [2]

Examples

```
>>> logP('67-56-1')
-0.74
>>> logP('100-66-3', 'WIKIDATA')
2.11
```

chemicals.environment.**logP_methods**(*CASRN*)

Return all methods available to obtain logP for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain logP with the given inputs.

See also:

logP

chemicals.environment.**logP_all_methods = ('SYRRES', 'CRC', 'WIKIDATA')**

Tuple of method name keys. See the *logP* for the actual references

1.10 Exceptions Generated by Chemicals (chemicals.exceptions)

This module contains various exception classes that may be raised by chemicals.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

class chemicals.exceptions.**UnderspecifiedError**

Generic error to raise when not enough values are given.

Attributes

args

Methods

with_traceback	Exception.with_traceback(tb) -- set self.__traceback__ to tb and return self.

class chemicals.exceptions.**OverspeficiedError**
Generic error to raise when too many values are given.

Attributes

args

Methods

with_traceback	Exception.with_traceback(tb) -- set self.__traceback__ to tb and return self.

class chemicals.exceptions.**TrivialSolutionError**(*message*, *comp_difference=None*, *iterations=None*, *err=None*)
Error raised SS converges to trivial solution.

Attributes

args

Methods

with_traceback	Exception.with_traceback(tb) -- set self.__traceback__ to tb and return self.

class chemicals.exceptions.**PhaseCountReducedError**(*message*, *zs=None*, *Ks=None*)
Error raised SS inner flash loop says all Ks are under 1 or above 1.

Attributes

args

Methods

with_traceback	Exception.with_traceback(tb) -- set self.__traceback__ to tb and return self.

class chemicals.exceptions.**PhaseExistenceImpossible**(*message*, *zs=None*, *T=None*, *P=None*)
Error raised SS inner flash loop says all Ks are under 1 or above 1.

Attributes

args

Methods

with_traceback	Exception.with_traceback(tb) -- set self.__traceback__ to tb and return self.

1.11 Ideal VLE and Flash Initialization (chemicals.flash_basic)

This module contains the ideal flash solver; two flash initialization routines; a vapor-liquid equilibrium constant correlation; a liquid-water equilibrium constant correlation, and a definition function to show the commonly used calculation frameworks.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.11.1 Ideal Flash Function

chemicals.flash_basic.**flash_ideal**(*zs*, *funcs*, *Tcs=None*, *T=None*, *P=None*, *VF=None*)

PVT flash model using ideal, composition-independent equation. Solves the various cases of composition-independent models.

Capable of solving with two of *T*, *P*, and *VF* for the other one; that results in three solve modes, but for *VF=1* and *VF=0*, there are additional solvers; for a total of seven solvers implemented.

The function takes a list of callables that take *T* in Kelvin as an argument, and return vapor pressure. The callables can include the effect of non-ideal pure component fugacity coefficients. For the (*T*, *P*) and (*P*, *VF*) cases, the Poynting correction factor can be easily included as well but not the (*T*, *VF*) case as the callable only takes *T* as an argument. Normally the Poynting correction factor is used with activity coefficient models with composition dependence.

Both *flash_wilson* and *flash_Tb_Tc_Pc* are specialized cases of this function and have the same functionality but with the model built right in.

Even when using more complicated models, this is useful for obtaining initial

This model uses *flash_inner_loop* to solve the Rachford-Rice problem.

Parameters

zs [list[float]] Mole fractions of the phase being flashed, [-]

funcs [list[Callable]] Functions to calculate ideal or real vapor pressures, take temperature in Kelvin and return pressure in Pa, [-]

Tcs [list[float], optional] Critical temperatures of all species; uses as upper bounds and only for the case that *T* is not specified; if they are needed and not given, it is assumed a method *solve_prop* exists in each of *funcs* which will accept *P* in Pa and return temperature in *K*, [K]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

VF [float, optional] Molar vapor fraction, [-]

Returns

T [float] Temperature, [K]

P [float] Pressure, [Pa]

VF [float] Molar vapor fraction, [-]

xs [list[float]] Mole fractions of liquid phase, [-]

ys [list[float]] Mole fractions of vapor phase, [-]

Notes

For the cases where *VF* is 1 or 0 and T is known, an explicit solution is used. For the same cases where *P* and *VF* are known, there is no explicit solution available.

There is an internal *Tmax* parameter, set to 50000 K; which, in the event of convergence of the Secant method, is used as a bounded for a bounded solver. It is used in the PVF solvers.

Examples

Basic case with four compounds, usingthe Antoine equation as a model and solving for vapor pressure:

```
>>> from chemicals import Antoine, Ambrose_Walton
>>> Tcs = [369.83, 425.12, 469.7, 507.6]
>>> Antoine_As = [8.92828, 8.93266, 8.97786, 9.00139]
>>> Antoine_Bs = [803.997, 935.773, 1064.84, 1170.88]
>>> Antoine_Cs = [-26.11, -34.361, -41.136, -48.833]
>>> Psat_funcs = []
>>> for i in range(4):
...     def Psat_func(T, A=Antoine_As[i], B=Antoine_Bs[i], C=Antoine_Cs[i]):
...         return Antoine(T, A, B, C)
...     Psat_funcs.append(Psat_func)
>>> zs = [.4, .3, .2, .1]
>>> T, P, VF, xs, ys = flash_ideal(T=330.55, P=1e6, zs=zs, funcs=Psat_funcs,␣
↪Tcs=Tcs)
>>> round(VF, 10)
1.00817e-05
```

Similar case, using the Ambrose-Walton corresponding states method to estimate vapor pressures:

```
>>> Tcs = [369.83, 425.12, 469.7, 507.6]
>>> Pcs = [4248000.0, 3796000.0, 3370000.0, 3025000.0]
>>> omegas = [0.152, 0.193, 0.251, 0.2975]
>>> Psat_funcs = []
>>> for i in range(4):
...     def Psat_func(T, Tc=Tcs[i], Pc=Pcs[i], omega=omegas[i]):
...         return Ambrose_Walton(T, Tc, Pc, omega)
...     Psat_funcs.append(Psat_func)
>>> _, P, VF, xs, ys = flash_ideal(T=329.151, VF=0, zs=zs, funcs=Psat_funcs,␣
↪Tcs=Tcs)
>>> round(P, 3)
1000013.343
```

Case with fugacities in the liquid phase, vapor phase, activity coefficients in the liquid phase, and Poynting correction factors.

```
>>> Tcs = [647.14, 514.0]
>>> Antoine_As = [10.1156, 10.3368]
>>> Antoine_Bs = [1687.54, 1648.22]
>>> Antoine_Cs = [-42.98, -42.232]
>>> gammas = [1.1, .75]
>>> fugacities_gas = [.995, 0.98]
>>> fugacities_liq = [.9999, .9998]
>>> Poyntings = [1.000001, .999999]
>>> zs = [.5, .5]
>>> funcs = []
>>> for i in range(2):
...     def K_over_P(T, A=Antoine_As[i], B=Antoine_Bs[i], C=Antoine_Cs[i],␣
↪fl=fugacities_liq[i],
...                  fg=fugacities_gas[i], gamma=gammas[i], poy=Poyntings[i]):
...         return Antoine(T, A, B, C)*gamma*poy*fl/fg
...     funcs.append(K_over_P)
>>> _, _, VF, xs, ys = flash_ideal(zs, funcs, Tcs=Tcs, P=1e5, T=364.0)
>>> VF, xs, ys
(0.5108639717, [0.55734934039, 0.44265065960], [0.44508982795, 0.554910172040])
```

Note that while this works for PT composition independent flashes - an outer iterating loop is needed for composition dependence!

1.11.2 Flash Initialization

chemicals.flash_basic.**flash_wilson**(*zs*, *Tcs*, *Pcs*, *omegas*, *T=None*, *P=None*, *VF=None*)

PVT flash model using Wilson's equation - useful for obtaining initial guesses for more rigorous models, or it can be used as its own model. Capable of solving with two of *T*, *P*, and *VF* for the other one; that results in three solve modes, but for *VF=1* and *VF=0*, there are additional solvers; for a total of seven solvers implemented.

This model uses *flash_inner_loop* to solve the Rachford-Rice problem.

$$K_i = \frac{P_c}{P}\exp\left(5.37(1+\omega)\left[1-\frac{T_c}{T}\right]\right)$$

Parameters

zs [list[float]] Mole fractions of the phase being flashed, [-]

Tcs [list[float]] Critical temperatures of all species, [K]

Pcs [list[float]] Critical pressures of all species, [Pa]

omegas [list[float]] Acentric factors of all species, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

VF [float, optional] Molar vapor fraction, [-]

Returns

T [float] Temperature, [K]

P [float] Pressure, [Pa]

VF [float] Molar vapor fraction, [-]

xs [list[float]] Mole fractions of liquid phase, [-]

ys [list[float]] Mole fractions of vapor phase, [-]

Notes

For the cases where *VF* is 1 or 0 and T is known, an explicit solution is used. For the same cases where *P* and *VF* are known, there is no explicit solution available.

There is an internal *Tmax* parameter, set to 50000 K; which, in the event of convergence of the Secant method, is used as a bounded for a bounded solver. It is used in the PVF solvers. This typically allows pressures up to 2 GPa to be converged to. However, for narrow-boiling mixtures, the PVF failure may occur at much lower pressures.

Examples

```
>>> Tcs = [305.322, 540.13]
>>> Pcs = [4872200.0, 2736000.0]
>>> omegas = [0.099, 0.349]
>>> zs = [0.4, 0.6]
>>> flash_wilson(zs=zs, Tcs=Tcs, Pcs=Pcs, omegas=omegas, T=300, P=1e5)
(300, 100000.0, 0.422194532936, [0.02093881508003, 0.979061184919], [0.918774185622,
↪ 0.0812258143])
```

chemicals.flash_basic.**flash_Tb_Tc_Pc**(*zs*, *Tbs*, *Tcs*, *Pcs*, *T=None*, *P=None*, *VF=None*)

PVT flash model using a model published in [1], which provides a PT surface using only each compound's boiling temperature and critical temperature and pressure. This is useful for obtaining initial guesses for more rigorous models, or it can be used as its own model. Capable of solving with two of *T*, *P*, and *VF* for the other one; that results in three solve modes, but for *VF=1* and *VF=0*, there are additional solvers; for a total of seven solvers implemented.

This model uses *flash_inner_loop* to solve the Rachford-Rice problem.

$$K_i = \frac{P_{c,i}^{\left(\frac{1}{T}-\frac{1}{T_{b,i}}\right)/\left(\frac{1}{T_{c,i}}-\frac{1}{T_{b,i}}\right)}}{P}$$

Parameters

zs [list[float]] Mole fractions of the phase being flashed, [-]

Tbs [list[float]] Boiling temperatures of all species, [K]

Tcs [list[float]] Critical temperatures of all species, [K]

Pcs [list[float]] Critical pressures of all species, [Pa]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

VF [float, optional] Molar vapor fraction, [-]

Returns

T [float] Temperature, [K]

P [float] Pressure, [Pa]

VF [float] Molar vapor fraction, [-]

xs [list[float]] Mole fractions of liquid phase, [-]

ys [list[float]] Mole fractions of vapor phase, [-]

Notes

For the cases where *VF* is 1 or 0 and T is known, an explicit solution is used. For the same cases where *P* and *VF* are known, there is no explicit solution available.

There is an internal *Tmax* parameter, set to 50000 K; which, in the event of convergence of the Secant method, is used as a bounded for a bounded solver. It is used in the PVF solvers. This typically allows pressures up to 2 MPa to be converged to. Failures may still occur for other conditions.

This model is based on [1], which aims to estimate dew and bubble points using the same K value formulation as used here. While this implementation uses a numerical solver to provide an exact bubble/dew point estimate, [1] suggests a sequential substitution and flowchart based solver with loose tolerances. That model was also implemented, but found to be slower and less reliable than this implementation.

References

[1]

Examples

```
>>> Tcs = [305.322, 540.13]
>>> Pcs = [4872200.0, 2736000.0]
>>> Tbs = [184.55, 371.53]
>>> zs = [0.4, 0.6]
>>> flash_Tb_Tc_Pc(zs=zs, Tcs=Tcs, Pcs=Pcs, Tbs=Tbs, T=300, P=1e5)
(300, 100000.0, 0.3807040748145, [0.0311578430365, 0.968842156963], [0.
↪9999999998827, 1.1729141887e-10])
```

1.11.3 Equilibrium Constants

chemicals.flash_basic.**K_value**(*P=None*, *Psat=None*, *phi_l=None*, *phi_g=None*, *gamma=None*, *Poynting=1.0*)

Calculates the equilibrium K-value assuming Raoult's law, or an equation of state model, or an activity coefficient model, or a combined equation of state-activity model.

The calculation procedure will use the most advanced approach with the provided inputs:

- If *P*, *Psat*, *phi_l*, *phi_g*, and *gamma* are provided, use the combined approach.
- If *P*, *Psat*, and *gamma* are provided, use the modified Raoult's law.
- If *phi_l* and *phi_g* are provided, use the EOS only method.
- If *P* and *Psat* are provided, use Raoult's law.

Definitions:

$$K_i = \frac{y_i}{x_i}$$

Raoult's law:

$$K_i = \frac{P_i^{sat}}{P}$$

Activity coefficient, no EOS (modified Raoult's law):

$$K_i = \frac{\gamma_i P_i^{sat}}{P}$$

Equation of state only:

$$K_i = \frac{\phi_i^l}{\phi_i^v} = \frac{f_i^l y_i}{f_i^v x_i}$$

Combined approach (liquid reference fugacity coefficient is normally calculated the saturation pressure for it as a pure species; vapor fugacity coefficient calculated normally):

$$K_i = \frac{\gamma_i P_i^{sat} \phi_i^{l,ref}}{\phi_i^v P}$$

Combined approach, with Poynting Correction Factor (liquid molar volume in the integral is for i as a pure species only):

$$K_i = \frac{\gamma_i P_i^{sat} \phi_i^{l,ref} \exp\left[\frac{\int_{P_i^{sat}}^{P} V_i^l dP}{RT}\right]}{\phi_i^v P}$$

Parameters

P [float] System pressure, optional

Psat [float] Vapor pressure of species i, [Pa]

phi_l [float] Fugacity coefficient of species i in the liquid phase, either at the system conditions (EOS-only case) or at the saturation pressure of species i as a pure species (reference condition for the combined approach), optional [-]

phi_g [float] Fugacity coefficient of species i in the vapor phase at the system conditions, optional [-]

gamma [float] Activity coefficient of species i in the liquid phase, optional [-]

Poynting [float] Poynting correction factor, optional [-]

Returns

K [float] Equilibrium K value of component i, calculated with an approach depending on the provided inputs [-]

Notes

The Poynting correction factor is normally simplified as follows, due to a liquid's low pressure dependency:

$$K_i = \frac{\gamma_i P_i^{sat} \phi_i^{l,ref} \exp\left[\frac{V_l(P-P_i^{sat})}{RT}\right]}{\phi_i^v P}$$

References

[1], [2]

Examples

Raoult's law:

```
>>> K_value(101325, 3000.)
0.029607698001480384
```

Modified Raoult's law:

```
>>> K_value(P=101325, Psat=3000, gamma=0.9)
0.026646928201332347
```

EOS-only approach:

```
>>> K_value(phi_l=1.6356, phi_g=0.88427)
1.8496613025433408
```

Gamma-phi combined approach:

```
>>> K_value(P=1E6, Psat=1938800, phi_l=1.4356, phi_g=0.88427, gamma=0.92)
2.8958055544121137
```

Gamma-phi combined approach with a Poynting factor:

```
>>> K_value(P=1E6, Psat=1938800, phi_l=1.4356, phi_g=0.88427, gamma=0.92,
... Poynting=0.999)
2.8929097488577016
```

chemicals.flash_basic.**Wilson_K_value**(*T*, *P*, *Tc*, *Pc*, *omega*)

Calculates the equilibrium K-value for a component using Wilson's heuristic mode. This is very useful for initialization of stability tests and flashes.

$$K_i = \frac{P_c}{P}\exp\left(5.37(1+\omega)\left[1-\frac{T_c}{T}\right]\right)$$

Parameters

T [float] System temperature, [K]

P [float] System pressure, [Pa]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

K [float] Equilibrium K value of component, calculated via the Wilson heuristic [-]

Notes

There has been little literature exploration of other formlulas for the same purpose. This model may be useful even for activity coefficient models.

Note the K-values are independent of composition; the correlation is applicable up to 3.5 MPa.

A description for how this function was generated can be found in [2].

References

[1], [2]

Examples

Ethane at 270 K and 76 bar:

```
>>> Wilson_K_value(270.0, 7600000.0, 305.4, 4880000.0, 0.098)
0.2963932297479371
```

The "vapor pressure" predicted by this equation can be calculated by multiplying by pressure:

```
>>> Wilson_K_value(270.0, 7600000.0, 305.4, 4880000.0, 0.098)*7600000.0
2252588.546084322
```

chemicals.flash_basic.**PR_water_K_value**(*T*, *P*, *Tc*, *Pc*)

Calculates the equilibrium K-value for a component against water according to the Peng and Robinson (1976) heuristic.

$$K_i = 10^6 \frac{P_{ri}}{T_{ri}}$$

Parameters

T [float] System temperature, [K]

P [float] System pressure, [Pa]

Tc [float] Critical temperature of chemical [K]

Pc [float] Critical pressure of chemical [Pa]

Returns

K [float] Equilibrium K value of component with water as the other phase (not as the reference), calculated via this heuristic [-]

Notes

Note the K-values are independent of composition.

References

[1]

Examples

Octane at 300 K and 1 bar:

```
>>> PR_water_K_value(300, 1e5, 568.7, 2490000.0)
76131.19143239626
```

1.12 Heat Capacity (chemicals.heat_capacity)

This module contains many heat capacity model equations, heat capacity estimation equations, enthalpy and entropy integrals of those heat capacity equations, enthalpy/entropy flash initialization routines, and many dataframes of coefficients.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.12.1 Gas Heat Capacity Model Equations

chemicals.heat_capacity.**TRCCp**(*T*, *a0*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*, *a7*)

Calculates ideal gas heat capacity using the model developed in [1]. The ideal gas heat capacity is given by:

$$C_p = R\left(a_0 + (a_1/T^2)\exp(-a_2/T) + a_3 y^2 + (a_4 - a_5/(T-a_7)^2)y^j\right)$$

$$y = \frac{T - a_7}{T + a_6} \text{ for } T > a_7 \text{ otherwise } 0$$

Parameters

T [float] Temperature [K]

a1-a7 [float] Coefficients

Returns

Cp [float] Ideal gas heat capacity , [J/mol/K]

Notes

j is set to 8. Analytical integrals are available for this expression.

References

[1]

Examples

```
>>> TRCCp(300, 4.0, 7.65E5, 720., 3.565, -0.052, -1.55E6, 52., 201.)
42.065271080974654
```

chemicals.heat_capacity.**TRCCp_integral**(*T*, *a0*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*, *a7*, *I=0*)

Integrates ideal gas heat capacity using the model developed in [1]. Best used as a delta only. The difference in enthalpy with respect to 0 K is given by:

$$\frac{H(T) - H^{ref}}{RT} = a_0 + a_1 x(a_2)/(a_2 T) + I/T + h(T)/T$$

$$h(T) = (a_5 + a_7)\left[(2a_3 + 8a_4)\ln(1-y) + \left\{a_3\left(1 + \frac{1}{1-y}\right) + a_4\left(7 + \frac{1}{1-y}\right)\right\} y + a_4\left\{3y^2 + (5/3)y^3 + y^4 + (3/5)y^5 +\right.\right.$$

$$h(T) = 0 \text{ for } T \le a_7 y = \frac{T - a_7}{T + a_6} \text{ for } T > a_7 \text{ otherwise } 0$$

Parameters

T [float] Temperature [K]

a1-a7 [float] Coefficients

I [float, optional] Integral offset

Returns

H-H(0) [float] Difference in enthalpy from 0 K , [J/mol]

Notes

Analytical integral as provided in [1] and verified with numerical integration.

References

[1]

Examples

```
>>> TRCCp_integral(298.15, 4.0, 7.65E5, 720., 3.565, -0.052, -1.55E6, 52.,
... 201., 1.2)
10802.536262068483
```

chemicals.heat_capacity.**TRCCp_integral_over_T**(*T*, *a0*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*, *a7*, *J=0*)

Integrates ideal gas heat capacity over T using the model developed in [1]. Best used as a delta only. The difference in ideal-gas entropy with respect to 0 K is given by:

$$\frac{S^\circ}{R} = J + a_0 \ln T + \frac{a_1}{a_2^2}\left(1 + \frac{a_2}{T}\right) x(a_2) + s(T)s(T) = \left[\left\{a_3 + \left(\frac{a_4 a_7^2 - a_5}{a_6^2}\right)\left(\frac{a_7}{a_6}\right)^4\right\}\left(\frac{a_7}{a_6}\right)^2 \ln z + (a_3 + a_4)\ln\left(\frac{T + a_6}{a_6 + a\right.\right.$$

$$s(T) = 0 \text{ for } T \le a_7$$

$$z = \frac{T}{T + a_6} \cdot \frac{a_7 + a_6}{a_7}$$

$$y = \frac{T - a_7}{T + a_6} \text{ for } T > a_7 \text{ otherwise } 0$$

Parameters

T [float] Temperature [K]

a1-a7 [float] Coefficients

J [float, optional] Integral offset

Returns

S-S(0) [float] Difference in entropy from 0 K , [J/mol/K]

Notes

Analytical integral as provided in [1] and verified with numerical integration.

References

[1]

Examples

```
>>> TRCCp_integral_over_T(300, 4.0, 124000, 245, 50.539, -49.469,
... 220440000, 560, 78)
213.80156219151888
```

chemicals.heat_capacity.**Shomate**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculates heat capacity using the Shomate polynomial model [1]. The heat capacity is given by:

$$C_p = A + BT + CT^2 + DT^3 + \frac{E}{T^2}$$

Parameters

T [float] Temperature [K]

A [float] Parameter, [J/(mol*K)]

B [float] Parameter, [J/(mol*K^2)]

C [float] Parameter, [J/(mol*K^3)]

D [float] Parameter, [J/(mol*K^4)]

E [float] Parameter, [J*K/(mol)]

Returns

Cp [float] Heat capacity , [J/mol/K]

Notes

Analytical integrals are available for this expression. In some sources such as [1], the equation is written with temperature in units of kilokelvin. The coefficients can be easily adjusted to be in the proper SI form.

References

[1]

Examples

Coefficients for water vapor from [1]:

```
>>> water_low_gas_coeffs = [30.09200, 6.832514/1e3, 6.793435/1e6, -2.534480/1e9, 0.
↪082139*1e6]
>>> Shomate(500, *water_low_gas_coeffs)
35.21836175
```

chemicals.heat_capacity.**Shomate_integral**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculates the enthalpy integral using the Shomate polynomial model [1]. The difference in enthalpy with respect to 0 K is given by:

$$H(T) - H^0 = AT + \frac{BT^2}{2} + \frac{CT^3}{3} + \frac{DT^4}{4} - \frac{E}{T}$$

Parameters

T [float] Temperature [K]

A [float] Parameter, [J/(mol*K)]

B [float] Parameter, [J/(mol*K^2)]

C [float] Parameter, [J/(mol*K^3)]

D [float] Parameter, [J/(mol*K^4)]

E [float] Parameter, [J*K/(mol)]

Returns

H-H(0) [float] Difference in enthalpy from 0 K , [J/mol]

References

[1]

Examples

Coefficients for water vapor from [1]:

```
>>> water_low_gas_coeffs = [30.09200, 6.832514/1e3, 6.793435/1e6, -2.534480/1e9, 0.
↪082139*1e6]
>>> Shomate_integral(500, *water_low_gas_coeffs)
15979.2447
```

chemicals.heat_capacity.**Shomate_integral_over_T**(*T*, *A*, *B*, *C*, *D*, *E*)

Integrates the heat capacity over T using the model developed in [1]. The difference in entropy with respect to 0 K is given by:

$$s(T) = A\log(T) + BT + \frac{CT^2}{2} + \frac{DT^3}{3} - \frac{E}{2T^2}$$

Parameters

T [float] Temperature [K]

A [float] Parameter, [J/(mol*K)]

B [float] Parameter, [J/(mol*K^2)]

C [float] Parameter, [J/(mol*K^3)]

D [float] Parameter, [J/(mol*K^4)]

E [float] Parameter, [J*K/(mol)]

Returns

S-S(0) [float] Difference in entropy from 0 K , [J/mol/K]

References

[1]

Examples

Coefficients for water vapor from [1]:

```
>>> water_low_gas_coeffs = [30.09200, 6.832514/1e3, 6.793435/1e6, -2.534480/1e9, 0.
↪082139*1e6]
>>> Shomate_integral_over_T(500, *water_low_gas_coeffs)
191.00554
```

class chemicals.heat_capacity.**ShomateRange**(*coeffs*, *Tmin*, *Tmax*)

Implementation of a range of the Shomate equation presented in [1] for calculating the heat capacity of a chemical. Implements the enthalpy and entropy integrals as well.

Parameters

coeffs [list[float]] Six coefficients for the equation, [-]

Tmin [float] Minimum temperature any experimental data was available at, [K]

Tmax [float] Maximum temperature any experimental data was available at, [K]

References

[1]

Methods

calculate(T)	Return heat capacity as a function of temperature.
calculate_integral(Ta, Tb)	Return the enthalpy integral of heat capacity from *Ta* to *Tb*.
calculate_integral_over_T(Ta, Tb)	Return the entropy integral of heat capacity from *Ta* to *Tb*.

calculate(*T*)
Return heat capacity as a function of temperature.

Parameters

T [float] Temperature, [K]

Returns

Cp [float] Liquid heat capacity as T, [J/mol/K]

calculate_integral(*Ta*, *Tb*)
Return the enthalpy integral of heat capacity from *Ta* to *Tb*.

Parameters

Ta [float] Initial temperature, [K]

Tb [float] Final temperature, [K]

Returns

dH [float] Enthalpy difference between *Ta* and *Tb*, [J/mol]

calculate_integral_over_T(*Ta*, *Tb*)
Return the entropy integral of heat capacity from *Ta* to *Tb*.

Parameters

Ta [float] Initial temperature, [K]

Tb [float] Final temperature, [K]

Returns

dS [float] Entropy difference between *Ta* and *Tb*, [J/mol/K]

chemicals.heat_capacity.**Poling**(*T*, *a*, *b*, *c*, *d*, *e*)
Return the ideal-gas molar heat capacity of a chemical using polynomial regressed coefficients as described by Poling et. al. [1].

Parameters

T [float] Temperature, [K]

a,b,c,d,e [float] Regressed coefficients.

Returns

Cpgm [float] Gas molar heat capacity, [J/mol/K]

See also:

Poling_integral

Poling_integral_over_T

Notes

The ideal gas heat capacity is given by:

$$C_n = R * (a + bT + cT^2 + dT^3 + eT^4)$$

The data is based on the Poling data bank.

References

[1]

Examples

Compute the gas heat capacity of Methane at 300 K:

```
>>> Poling(T=300., a=4.568, b=-0.008975, c=3.631e-05, d=-3.407e-08, e=1.091e-11)
35.850973388425
```

chemicals.heat_capacity.**Poling_integral**(*T*, *a*, *b*, *c*, *d*, *e*)

Return the integral of the ideal-gas constant-pressure heat capacity of a chemical using polynomial regressed coefficients as described by Poling et. al. [1].

Parameters

T [float] Temperature, [K]

a,b,c,d,e [float] Regressed coefficients.

Returns

H [float] Difference in enthalpy from 0 K, [J/mol]

See also:

Poling

Poling_integral_over_T

Notes

Integral was computed with SymPy.

References

[1]

Examples

Compute the gas enthalpy of Methane at 300 K (with reference to 0 K):

```
>>> Poling_integral(T=300., a=4.568, b=-0.008975, c=3.631e-05, d=-3.407e-08, e=1.
↪091e-11)
10223.67533722261
```

chemicals.heat_capacity.**Poling_integral_over_T**(*T*, *a*, *b*, *c*, *d*, *e*)

Return the integral over temperature of the ideal-gas constant-pressure heat capacity of a chemical using polynomial regressed coefficients as described by Poling et. al. [1].

Parameters

T [float] Temperature, [K]

a,b,c,d,e [float] Regressed coefficients.

Returns

S [float] Difference in entropy from 0 K, [J/mol/K]

See also:

Poling

Poling_integral

Notes

Integral was computed with SymPy.

References

[1]

Examples

Compute the gas entropy of Methane at 300 K (with reference to 0 K):

```
>>> Poling_integral_over_T(T=300., a=4.568, b=-0.008975, c=3.631e-05, d=-3.407e-08,␣
↪e=1.091e-11)
205.46526328058
```

chemicals.heat_capacity.**PPDS2**(*T*, *Ts*, *C_low*, *C_inf*, *a1*, *a2*, *a3*, *a4*, *a5*)

Calculates the ideal-gas heat capacity using the [1] emperical (parameter-regressed) method, called the PPDS 2 equation for heat capacity.

$$\frac{C_p^0}{R} = C_{low} + (C_{\inf} - C_{low}) y^2 \left(1 + (y-1)\left[\sum_{i=0}^{4} a_i y^i\right]\right)$$

$$y = \frac{T}{T + T_s}$$

Parameters

T [float] Temperature of fluid [K]

Ts [float] Fit temperature; no physical meaning [K]

C_low [float] Fit parameter equal to Cp/R at a low temperature, [-]

C_inf [float] Fit parameter equal to Cp/R at a high temperature, [-]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

a5 [float] Regression parameter, [-]

Returns

Cpgm [float] Gas molar heat capacity, [J/mol/K]

References

[1]

Examples

n-pentane at 350 K from [1]

```
>>> PPDS2(T=350.0, Ts=462.493, C_low=4.54115, C_inf=9.96847, a1=-103.419, a2=695.
↪484, a3=-2006.1, a4=2476.84, a5=-1186.47)
136.46338956689
```

1.12.2 Gas Heat Capacity Estimation Models

chemicals.heat_capacity.**Lastovka_Shaw**(*T*, *similarity_variable*, *cyclic_aliphatic=False*, *MW=None*, *term_A=None*)

Calculate ideal-gas constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

$$term_A = A1 + A2 * a \text{ if cyclic aliphatic}$$

$$term_A = \left(A_2 + \frac{A_1 - A_2}{1 + \exp(\frac{\alpha - A_3}{A_4})}\right) \text{ if not cyclic aliphatic}$$

$$C_p^0 = term_A + (B_{11} + B_{12}\alpha)\left(-\frac{(C_{11} + C_{12}\alpha)}{T}\right)^2 \frac{\exp(-(C_{11} + C_{12}\alpha)/T)}{[1 - \exp(-(C_{11} + C_{12}\alpha)/T)]^2} + (B_{21} + B_{22}\alpha)\left(-\frac{(C_{21} + C_{22}\alpha)}{T}\right)^2 \frac{}{[1}$$

Parameters

T [float] Temperature of gas [K]

similarity_variable [float] Similarity variable as defined in [1], [mol/g]

cyclic_aliphatic: bool, optional Whether or not chemical is cyclic aliphatic, [-]

MW [float, optional] Molecular weight, [g/mol]

term_A [float, optional] Term A in Lastovka-Shaw equation, [J/g]

Returns

Cpg [float] Gas constant-pressure heat capacity, J/mol/K if MW given; J/kg/K otherwise

Notes

Original model is in terms of J/g/K.

A1 = -0.1793547 text{ if cyclic aliphatic}

A1 = 0.58 text{ if not cyclic aliphatic}

A2 = 3.86944439 text{ if cyclic aliphatic}

A2 = 1.25 text{ if not cyclic aliphatic}

A3 = 0.17338003

A4 = 0.014

B11 = 0.73917383

B12 = 8.88308889

C11 = 1188.28051

C12 = 1813.04613

B21 = 0.0483019

B22 = 4.35656721

C21 = 2897.01927

C22 = 5987.80407

References

[1]

Examples

Estimate the heat capacity of n-decane gas in J/kg/K:

```
>>> Lastovka_Shaw(1000.0, 0.22491)
3730.2807601773725
```

Estimate the heat capacity of n-decane gas in J/mol/K:

```
>>> Lastovka_Shaw(1000.0, 0.22491, MW=142.28)
530.7443465580366
```

chemicals.heat_capacity.**Lastovka_Shaw_integral**(*T*, *similarity_variable*, *cyclic_aliphatic=False*, *MW=None*, *term_A=None*)

Calculate the integral of ideal-gas constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

Parameters

T [float] Temperature of gas [K]

cyclic_aliphatic: bool, optional Whether or not chemical is cyclic aliphatic, [-]

MW [float, optional] Molecular weight, [g/mol]

term_A [float, optional] Term A in Lastovka-Shaw equation, [J/g]

Returns

H [float] Difference in enthalpy from 0 K, J/mol if MW given; J/kg otherwise

See also:

Lastovka_Shaw

Lastovka_Shaw_integral_over_T

Notes

Original model is in terms of J/g/K. Integral was computed with SymPy.

References

[1]

Examples

```
>>> Lastovka_Shaw_integral(300.0, 0.1333)
5283095.816018478
```

chemicals.heat_capacity.**Lastovka_Shaw_integral_over_T**(*T*, *similarity_variable*, *cyclic_aliphatic=False*, *MW=None*, *term_A=None*)

Calculate the integral over temperature of ideal-gas constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

Parameters

T [float] Temperature of gas [K]

similarity_variable [float] Similarity variable as defined in [1], [mol/g]

cyclic_aliphatic: bool, optional Whether or not chemical is cyclic aliphatic, [-]

MW [float, optional] Molecular weight, [g/mol]

term_A [float, optional] Term A in Lastovka-Shaw equation, [J/g]

Returns

S [float] Difference in entropy from 0 K, [J/mol/K if MW given; J/kg/K otherwise]

See also:

Lastovka_Shaw

Lastovka_Shaw_integral

Notes

Original model is in terms of J/g/K. Note that the model is for predicting mass heat capacity, not molar heat capacity like most other methods! Integral was computed with SymPy.

References

[1]

Examples

```
>>> Lastovka_Shaw_integral_over_T(300.0, 0.1333)
3609.791928945323
```

chemicals.heat_capacity.**Lastovka_Shaw_T_for_Hm**(*Hm*, *MW*, *similarity_variable*, *T_ref=298.15*, *factor=1.0*, *cyclic_aliphatic=None*, *term_A=None*)

Uses the Lastovka-Shaw ideal-gas heat capacity correlation to solve for the temperature which has a specified *Hm*, as is required in PH flashes, as shown in [1].

Parameters

Hm [float] Molar enthalpy spec, [J/mol]

MW [float] Molecular weight of the pure compound or mixture average, [g/mol]

similarity_variable [float] Similarity variable as defined in [1], [mol/g]

T_ref [float, optional] Reference enthlapy temperature, [K]

factor [float, optional] A factor to increase or decrease the predicted value of the method, [-]

cyclic_aliphatic: bool, optional Whether or not chemical is cyclic aliphatic, [-]

term_A [float, optional] Term A in Lastovka-Shaw equation, [J/g]

Returns

T [float] Temperature of gas to meet the molar enthalpy spec, [K]

See also:

Lastovka_Shaw

Lastovka_Shaw_integral

Lastovka_Shaw_integral_over_T

References

[1]

Examples

```
>>> Lastovka_Shaw_T_for_Hm(Hm=55000, MW=80.0, similarity_variable=0.23)
600.0943429567602
```

chemicals.heat_capacity.**Lastovka_Shaw_T_for_Sm**(*Sm*, *MW*, *similarity_variable*, *T_ref=298.15*, *factor=1.0*, *cyclic_aliphatic=None*, *term_A=None*)

Uses the Lastovka-Shaw ideal-gas heat capacity correlation to solve for the temperature which has a specified *Sm*, as is required in PS flashes, as shown in [1].

Parameters

Sm [float] Molar entropy spec, [J/mol/K]

MW [float] Molecular weight of the pure compound or mixture average, [g/mol]

similarity_variable [float] Similarity variable as defined in [1], [mol/g]

T_ref [float, optional] Reference enthlapy temperature, [K]

factor [float, optional] A factor to increase or decrease the predicted value of the method, [-]

cyclic_aliphatic: bool, optional Whether or not chemical is cyclic aliphatic, [-]

term_A [float, optional] Term A in Lastovka-Shaw equation, [J/g]

Returns

T [float] Temperature of gas to meet the molar entropy spec, [K]

See also:

Lastovka_Shaw

Lastovka_Shaw_integral

Lastovka_Shaw_integral_over_T

References

[1]

Examples

```
>>> Lastovka_Shaw_T_for_Sm(Sm=112.80, MW=72.151, similarity_variable=0.2356)
603.4298291570276
```

chemicals.heat_capacity.**Lastovka_Shaw_term_A**(*similarity_variable*, *cyclic_aliphatic*)

Return Term A in Lastovka-Shaw equation.

Parameters

similarity_variable [float] Similarity variable as defined in [1], [mol/g]

cyclic_aliphatic: bool, optional Whether or not chemical is cyclic aliphatic, [-]

Returns

term_A [float] Term A in Lastovka-Shaw equation, [J/g]

See also:

Lastovka_Shaw

Lastovka_Shaw_integral

Lastovka_Shaw_integral_over_T

References

[1]

1.12.3 Gas Heat Capacity Theory

chemicals.heat_capacity.**Cpg_statistical_mechanics**(*T*, *thetas*, *linear=False*)

Calculates the ideal-gas heat capacity using of a molecule using its characteristic temperatures, themselves calculated from each of the frequencies of vibration of the molecule. These can be obtained from spectra or quantum mechanical calculations.

$$\frac{C_p^0}{R} = \frac{C_p^0}{R}\text{rotational} + \frac{C_p^0}{R}\text{translational} + \frac{C_p^0}{R}\text{vibrational}$$

$$\frac{C_p^0}{R}\text{rotational} = 2.5$$

$$\frac{C_p^0}{R}\text{translational} = 1 \text{ if linear else } 1.5$$

$$\frac{C_p^0}{R}\text{vibrational} = \sum_{i=1}^{3n_A-6+\delta} \left(\frac{\theta_i}{T}\right)^2 \left[\frac{\exp(\theta_i/T)}{(\exp(\theta_i/T)-1)^2}\right]$$

In the above equation, $delta$ is 1 if the molecule is linear otherwise 0.

Parameters

T [float] Temperature of fluid [K]

thetas [list[float]] Characteristic temperatures, [K]

Returns

Cpgm [float] Gas molar heat capacity at specified temperature, [J/mol/K]

Notes

This equation implies that there is a maximum heat capacity for an ideal gas, and all diatomic or larger gases

Monoatomic gases have a simple heat capacity of 2.5R, the lower limit for ideal gas heat capacity. This function does not cover that type of a gas. At very low temperatures hydrogen behaves like a monoatomic gas as well.

References

[1]

Examples

Sample calculation in [1] for ammonia:

```
>>> thetas = [1360, 2330, 2330, 4800, 4880, 4880]
>>> Cpg_statistical_mechanics(300.0, thetas)
35.55983440173097
```

chemicals.heat_capacity.**Cpg_statistical_mechanics_integral**(*T*, *thetas*, *linear=False*)

Calculates the integral of ideal-gas heat capacity using of a molecule using its characteristic temperatures.

$$\int C_p^0 = 2.5RT + RT \text{ if linear else } 1.5RT + \int C_p^0 \text{vibrational}$$

$$\int C_p^0 \text{vibrational} = R \sum_{i=1}^{3n_A-6+\delta} \frac{\theta_i}{\exp(\theta_i/T) - 1}$$

Parameters

T [float] Temperature of fluid [K]

thetas [list[float]] Characteristic temperatures, [K]

Returns

H [float] Integrated gas molar heat capacity at specified temperature, [J/mol]

Examples

```
>>> thetas = [1360, 2330, 2330, 4800, 4880, 4880]
>>> Cpg_statistical_mechanics_integral(300.0, thetas)
10116.6053294
```

chemicals.heat_capacity.**Cpg_statistical_mechanics_integral_over_T**(*T*, *thetas*, *linear=False*)

Calculates the integral over T of ideal-gas heat capacity using of a molecule using its characteristic temperatures.

$$\int \frac{C_p^0}{T} = 2.5R\log(T) + 1R\log(T) \text{ if linear else } 1.5R\log(T) + \int \frac{C_p^0}{T} \text{vibrational}$$

$$\int \frac{C_p^0}{T} \text{vibrational} = \sum_{i=1}^{3n_A-6+\delta} \frac{\theta_i}{T\exp(\theta_i/T) - T} - \log(\exp(\theta_i/T) - 1) + \theta_i/T$$

Parameters

T [float] Temperature of fluid [K]

thetas [list[float]] Characteristic temperatures, [K]

Returns

S [float] Entropy integral of gas molar heat capacity at specified temperature, [J/mol/K]

Examples

```
>>> thetas = [1360, 2330, 2330, 4800, 4880, 4880]
>>> Cpg_statistical_mechanics_integral_over_T(300.0, thetas)
190.25658088
```

chemicals.heat_capacity.**vibration_frequency_cm_to_characteristic_temperature**(*frequency*, *scale=1*)

Convert a vibrational frequency in units of 1/cm to a characteristic temperature for use in calculating heat capacity.

$$\theta = \frac{100 \cdot h \cdot c \cdot \text{scale}}{k}$$

Parameters

frequency [float] Vibrational frequency, [1/cm]

scale [float] A scale factor used to adjust the frequency for differences in experimental vs. calculated values, [-]

Returns

theta [float] Characteristic temperature [K]

Notes

In the equation, k is Boltzmann's constant, c is the speed of light, and h is the Planck constant.

A scale factor for the MP2/6-31G** method recommended by NIST is 0.9365. Using this scale factor will not improve results in all cases however.

Examples

```
>>> vibration_frequency_cm_to_characteristic_temperature(667)
959.6641613636505
```

1.12.4 Liquid Heat Capacity Model Equations

chemicals.heat_capacity.**Zabransky_quasi_polynomial**(*T*, *Tc*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*)

Calculates liquid heat capacity using the model developed in [1].

$$\frac{C}{R} = A_1 \ln(1 - T_r) + \frac{A_2}{1 - T_r} + \sum_{j=0}^{m} A_{j+3} T_r^j$$

Parameters

T [float] Temperature [K]

Tc [float] Critical temperature of fluid, [K]

a1-a6 [float] Coefficients

Returns

Cp [float] Liquid heat capacity, [J/mol/K]

Notes

Used only for isobaric heat capacities, not saturation heat capacities. Designed for reasonable extrapolation behavior caused by using the reduced critical temperature. Used by the authors of [1] when critical temperature was available for the fluid. Analytical integrals are available for this expression.

References

[1]

Examples

```
>>> Zabransky_quasi_polynomial(330, 591.79, -3.12743, 0.0857315, 13.7282, 1.28971,␣
↪6.42297, 4.10989)
165.472878778683
```

chemicals.heat_capacity.**Zabransky_quasi_polynomial_integral**(*T*, *Tc*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*)
Calculates the integral of liquid heat capacity using the quasi-polynomial model developed in [1].

Parameters

T [float] Temperature [K]

a1-a6 [float] Coefficients

Returns

H [float] Difference in enthalpy from 0 K, [J/mol]

Notes

The analytical integral was derived with SymPy; it is a simple polynomial plus some logarithms.

References

[1]

Examples

```
>>> H2 = Zabransky_quasi_polynomial_integral(300, 591.79, -3.12743,
... 0.0857315, 13.7282, 1.28971, 6.42297, 4.10989)
>>> H1 = Zabransky_quasi_polynomial_integral(200, 591.79, -3.12743,
... 0.0857315, 13.7282, 1.28971, 6.42297, 4.10989)
>>> H2 - H1
14662.031376528757
```

chemicals.heat_capacity.**Zabransky_quasi_polynomial_integral_over_T**(*T*, *Tc*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*)
Calculates the integral of liquid heat capacity over T using the quasi-polynomial model developed in [1].

Parameters

T [float] Temperature [K]

a1-a6 [float] Coefficients

Returns

S [float] Difference in entropy from 0 K, [J/mol/K]

Notes

The analytical integral was derived with Sympy. It requires the Polylog(2,x) function, which is unimplemented in SciPy. A very accurate numerical approximation was implemented as `fluids.numerics.polylog2`. Relatively slow due to the use of that special function.

References

[1]

Examples

```
>>> S2 = Zabransky_quasi_polynomial_integral_over_T(300, 591.79, -3.12743,
... 0.0857315, 13.7282, 1.28971, 6.42297, 4.10989)
>>> S1 = Zabransky_quasi_polynomial_integral_over_T(200, 591.79, -3.12743,
... 0.0857315, 13.7282, 1.28971, 6.42297, 4.10989)
>>> S2 - S1
59.16999297436473
```

chemicals.heat_capacity.**Zabransky_cubic**(*T*, *a1*, *a2*, *a3*, *a4*)

Calculates liquid heat capacity using the model developed in [1].

$$\frac{C}{R} = \sum_{j=0}^{3} A_{j+1} \left(\frac{T}{100\text{K}} \right)^j$$

Parameters

T [float] Temperature [K]

a1 [float] Coefficient, [-]

a2 [float] Coefficient, [-]

a3 [float] Coefficient, [-]

a4 [float] Coefficient, [-]

Returns

Cp [float] Liquid heat capacity, [J/mol/K]

Notes

Most often form used in [1]. Analytical integrals are available for this expression.

References

[1]

Examples

```
>>> Zabransky_cubic(298.15, 20.9634, -10.1344, 2.8253, -0.256738)
75.31465144297
```

chemicals.heat_capacity.**Zabransky_cubic_integral**(*T*, *a1*, *a2*, *a3*, *a4*)
Calculates the integral of liquid heat capacity using the model developed in [1].

Parameters

T [float] Temperature [K]

a1 [float] Coefficient, [-]

a2 [float] Coefficient, [-]

a3 [float] Coefficient, [-]

a4 [float] Coefficient, [-]

Returns

H [float] Difference in enthalpy from 0 K, [J/mol]

Notes

The analytical integral was derived with Sympy; it is a simple polynomial.

References

[1]

Examples

```
>>> Zabransky_cubic_integral(298.15, 20.9634, -10.1344, 2.8253, -0.256738)
31051.690370364
```

chemicals.heat_capacity.**Zabransky_cubic_integral_over_T**(*T*, *a1*, *a2*, *a3*, *a4*)
Calculates the integral of liquid heat capacity over T using the model developed in [1].

Parameters

T [float] Temperature [K]

a1 [float] Coefficient, [-]

a2 [float] Coefficient, [-]

a3 [float] Coefficient, [-]

a4 [float] Coefficient, [-]

Returns

S [float] Difference in entropy from 0 K, [J/mol/K]

Notes

The analytical integral was derived with Sympy; it is a simple polynomial, plus a logarithm

References

[1]

Examples

```
>>> Zabransky_cubic_integral_over_T(298.15, 20.9634, -10.1344, 2.8253,
... -0.256738)
24.732465342840
```

class chemicals.heat_capacity.**ZabranskySpline**(*coeffs*, *Tmin*, *Tmax*)

Implementation of the cubic spline method presented in [1] for calculating the heat capacity of a chemical. Implements the enthalpy and entropy integrals as well.

$$\frac{C}{R} = \sum_{j=0}^{3} A_{j+1} \left(\frac{T}{100}\right)^j$$

Parameters

coeffs [list[float]] Six coefficients for the equation, [-]

Tmin [float] Minimum temperature any experimental data was available at, [K]

Tmax [float] Maximum temperature any experimental data was available at, [K]

References

[1]

Methods

calculate(T)	Return heat capacity as a function of temperature.
calculate_integral(Ta, Tb)	Return the enthalpy integral of heat capacity from *Ta* to *Tb*.
calculate_integral_over_T(Ta, Tb)	Return the entropy integral of heat capacity from *Ta* to *Tb*.

calculate(*T*)

Return heat capacity as a function of temperature.

Parameters

T [float] Temperature, [K]

Returns

Cp [float] Liquid heat capacity as T, [J/mol/K]

calculate_integral(*Ta*, *Tb*)
Return the enthalpy integral of heat capacity from *Ta* to *Tb*.

Parameters

Ta [float] Initial temperature, [K]

Tb [float] Final temperature, [K]

Returns

dH [float] Enthalpy difference between *Ta* and *Tb*, [J/mol]

calculate_integral_over_T(*Ta*, *Tb*)
Return the entropy integral of heat capacity from *Ta* to *Tb*.

Parameters

Ta [float] Initial temperature, [K]

Tb [float] Final temperature, [K]

Returns

dS [float] Entropy difference between *Ta* and *Tb*, [J/mol/K]

class chemicals.heat_capacity.**ZabranskyQuasipolynomial**(*coeffs*, *Tc*, *Tmin*, *Tmax*)
Quasi-polynomial object for calculating the heat capacity of a chemical. Implements the enthalpy and entropy integrals as well.

$$\frac{C}{R} = A_1 \ln(1 - T_r) + \frac{A_2}{1 - T_r} + \sum_{j=0}^{m} A_{j+3} T_r^j$$

Parameters

coeffs [list[float]] Six coefficients for the equation, [-]

Tc [float] Critical temperature of the chemical, as used in the formula, [K]

Tmin [float] Minimum temperature any experimental data was available at, [K]

Tmax [float] Maximum temperature any experimental data was available at, [K]

References

[1]

Methods

calculate(T)	Return the heat capacity as a function of temperature.
calculate_integral(Ta, Tb)	Return the enthalpy integral of heat capacity from *Ta* to *Tb*.
calculate_integral_over_T(Ta, Tb)	Return the entropy integral of heat capacity from *Ta* to *Tb*.

calculate(*T*)
Return the heat capacity as a function of temperature.

Parameters

T [float] Temperature, [K]

Returns

Cp [float] Liquid heat capacity as T, [J/mol/K]

`calculate_integral`(*Ta*, *Tb*)
Return the enthalpy integral of heat capacity from *Ta* to *Tb*.

Parameters

Ta [float] Initial temperature, [K]

Tb [float] Final temperature, [K]

Returns

dH [float] Enthalpy difference between *Ta* and *Tb*, [J/mol]

`calculate_integral_over_T`(*Ta*, *Tb*)
Return the entropy integral of heat capacity from *Ta* to *Tb*.

Parameters

Ta [float] Initial temperature, [K]

Tb [float] Final temperature, [K]

Returns

dS [float] Entropy difference between *Ta* and *Tb*, [J/mol/K]

chemicals.heat_capacity.**PPDS15**(*T*, *Tc*, *a0*, *a1*, *a2*, *a3*, *a4*, *a5*)
Calculates the saturation liquid heat capacity using the [1] emperical (parameter-regressed) method, called the PPDS 15 equation for heat capacity.

$$\frac{C_{p,l}}{R} = \frac{a_0}{\tau} + a_1 + a_2\tau + a_3\tau^2 + a_4\tau^3 + a_5\tau^4$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a0 [float] Regression parameter, [-]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

a5 [float] Regression parameter, [-]

Returns

Cplm [float] Liquid molar saturation heat capacity, [J/mol/K]

References

[1]

Examples

Benzene at 400 K from [1]

```
>>> PPDS15(T=400.0, Tc=562.05, a0=0.198892, a1=24.1389, a2=-20.2301, a3=5.72481,␣
↪a4=4.43613e-7, a5=-3.10751e-7)
161.8983143509
```

chemicals.heat_capacity.**TDE_CSExpansion**(*T*, *Tc*, *b*, *a1*, *a2=0.0*, *a3=0.0*, *a4=0.0*)

Calculates the saturation liquid heat capacity using the [1] CSExpansion method from NIST's TDE:

$$C_{p,l} = \frac{b}{\tau} + a_1 + a_2 T + a_3 T^2 + a_4 T^3$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

b [float] Regression parameter, [-]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

Returns

Cplm [float] Liquid molar saturation heat capacity, [J/mol/K]

References

[1]

Examples

2-methylquinoline at 550 K from [1]

```
>>> TDE_CSExpansion(550.0, 778.0, 0.626549, 120.705, 0.255987, 0.000381027, -3.
↪03077e-7)
328.472042686
```

1.12.5 Liquid Heat Capacity Estimation Models

chemicals.heat_capacity.**Rowlinson_Poling**(*T*, *Tc*, *omega*, *Cpgm*)

Calculate liquid constant-pressure heat capacity with the [1] CSP method. This equation is not terrible accurate.

The heat capacity of a liquid is given by:

$$\frac{Cp^L - Cp^g}{R} = 1.586 + \frac{0.49}{1-T_r} + \omega\left[4.2775 + \frac{6.3(1-T_r)^{1/3}}{T_r} + \frac{0.4355}{1-T_r}\right]$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor for fluid, [-]

Cpgm [float] Constant-pressure gas heat capacity, [J/mol/K]

Returns

Cplm [float] Liquid constant-pressure heat capacity, [J/mol/K]

Notes

Poling compared 212 substances, and found error at 298K larger than 10% for 18 of them, mostly associating. Of the other 194 compounds, AARD is 2.5%.

References

[1]

Examples

```
>>> Rowlinson_Poling(350.0, 435.5, 0.203, 91.21)
143.80196224081436
```

chemicals.heat_capacity.**Rowlinson_Bondi**(*T*, *Tc*, *omega*, *Cpgm*)

Calculate liquid constant-pressure heat capacity with the CSP method shown in [1].

The heat capacity of a liquid is given by:

$$\frac{Cp^L - Cp^{ig}}{R} = 1.45 + 0.45(1-T_r)^{-1} + 0.25\omega[17.11 + 25.2(1-T_r)^{1/3}T_r^{-1} + 1.742(1-T_r)^{-1}]$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor for fluid, [-]

Cpgm [float] Constant-pressure gas heat capacity, [J/mol/K]

Returns

Cplm [float] Liquid constant-pressure heat capacity, [J/mol/K]

Notes

Less accurate than *Rowlinson_Poling*.

References

[1], [2], [3]

Examples

```
>>> Rowlinson_Bondi(T=373.28, Tc=535.55, omega=0.323, Cpgm=119.342)
175.3976263003074
```

chemicals.heat_capacity.**Dadgostar_Shaw**(*T*, *similarity_variable*, *MW=None*, *terms=None*)

Calculate liquid constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

$$C_p = 24.5(a_{11}\alpha + a_{12}\alpha^2) + (a_{21}\alpha + a_{22}\alpha^2)T + (a_{31}\alpha + a_{32}\alpha^2)T^2$$

Parameters

T [float] Temperature of liquid [K]

similarity_variable [float] similarity variable as defined in [1], [mol/g]

MW [float, optional] Molecular weight of the pure compound or mixture average, [g/mol]

terms [float, optional] Terms in Dadgostar-Shaw equation as computed by *Dadgostar_Shaw_terms*

Returns

Cpl [float] Liquid constant-pressure heat capacity, J/mol/K if MW given; J/kg/K otherwise

Notes

Many restrictions on its use. Original model is in terms of J/g/K. Note that the model is for predicting mass heat capacity, not molar heat capacity like most other methods! a11 = -0.3416; a12 = 2.2671; a21 = 0.1064; a22 = -0.3874l; a31 = -9.8231E-05; a32 = 4.182E-04

References

[1]

Examples

```
>>> Dadgostar_Shaw(355.6, 0.139)
1802.5291501191516
```

chemicals.heat_capacity.**Dadgostar_Shaw_integral**(*T*, *similarity_variable*, *MW=None*, *terms=None*)

Calculate the integral of liquid constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

Parameters

T [float] Temperature of gas [K]

similarity_variable [float] similarity variable as defined in [1], [mol/g]

MW [float, optional] Molecular weight of the pure compound or mixture average, [g/mol]

terms [float, optional] Terms in Dadgostar-Shaw equation as computed by *Dadgostar_Shaw_terms*

Returns

H [float] Difference in enthalpy from 0 K, J/mol if MW given; J/kg otherwise

See also:

Dadgostar_Shaw

Dadgostar_Shaw_integral_over_T

Notes

Original model is in terms of J/g/K. Note that the model is for predicting mass heat capacity, not molar heat capacity like most other methods! Integral was computed with SymPy.

References

[1]

Examples

```
>>> Dadgostar_Shaw_integral(300.0, 0.1333)
238908.15142664989
```

chemicals.heat_capacity.**Dadgostar_Shaw_integral_over_T**(*T*, *similarity_variable*, *MW=None*, *terms=None*)

Calculate the integral of liquid constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

Parameters

T [float] Temperature of gas [K]

similarity_variable [float] similarity variable as defined in [1], [mol/g]

MW [float, optional] Molecular weight of the pure compound or mixture average, [g/mol]

terms [float, optional] Terms in Dadgostar-Shaw equation as computed by *Dadgostar_Shaw_terms*

Returns

S [float] Difference in entropy from 0 K, J/mol/K if MW given; J/kg/K otherwise

See also:

Dadgostar_Shaw

Dadgostar_Shaw_integral

Notes

Original model is in terms of J/g/K. Note that the model is for predicting mass heat capacity, not molar heat capacity like most other methods! Integral was computed with SymPy.

References

[1]

Examples

```
>>> Dadgostar_Shaw_integral_over_T(300.0, 0.1333)
1201.1409113147918
```

chemicals.heat_capacity.**Dadgostar_Shaw_terms**(*similarity_variable*)
Return terms for the computation of Dadgostar-Shaw heat capacity equation.

Parameters

similarity_variable [float] Similarity variable, [mol/g]

Returns

first [float] First term, [-]

second [float] Second term, [-]

third [float] Third term, [-]

See also:

Dadgostar_Shaw

1.12.6 Solid Heat Capacity Estimation Models

chemicals.heat_capacity.**Perry_151**(*T*, *a*, *b*, *c*, *d*)
Return the solid molar heat capacity of a chemical using the Perry 151 method, as described in [1].

Parameters

a,b,c,d [float] Regressed coefficients.

Returns

Cps [float] Solid constant-pressure heat capacity, [J/mol/K]

Notes

The solid heat capacity is given by:

$$C_n = 4.184(a + bT + \frac{c}{T^2} + dT^2)$$

Coefficients are listed in section 2, table 151 of [1]. Note that the original model was in a Calorie basis, but has been translated to Joules.

References

[1]

Examples

Heat capacity of solid aluminum at 300 K:

```
>>> Perry_151(300, 4.8, 0.00322, 0., 0.)
24.124944
```

chemicals.heat_capacity.**Lastovka_solid**(*T*, *similarity_variable*, *MW=None*)

Calculate solid constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

$$C_p = 3(A_1\alpha + A_2\alpha^2)R\left(\frac{\theta}{T}\right)^2\frac{\exp(\theta/T)}{[\exp(\theta/T)-1]^2} + (C_1\alpha + C_2\alpha^2)T + (D_1\alpha + D_2\alpha^2)T^2$$

Parameters

T [float] Temperature of solid [K]

similarity_variable [float] similarity variable as defined in [1], [mol/g]

MW [float, optional] Molecular weight of the pure compound or mixture average, [g/mol]

Returns

Cps [float] Solid constant-pressure heat capacity, J/mol/K if MW given; J/kg/K otherwise

Notes

Many restrictions on its use. Trained on data with MW from 12.24 g/mol to 402.4 g/mol, C mass fractions from 61.3% to 95.2%, H mass fractions from 3.73% to 15.2%, N mass fractions from 0 to 15.4%, O mass fractions from 0 to 18.8%, and S mass fractions from 0 to 29.6%. Recommended for organic compounds with low mass fractions of hetero-atoms and especially when molar mass exceeds 200 g/mol. This model does not show and effects of phase transition but should not be used passed the triple point. Original model is in terms of J/g/K. Note that the model s for predicting mass heat capacity, not molar heat capacity like most other methods!

A1 = 0.013183

A2 = 0.249381

θ = 151.8675

C1 = 0.026526

C2 = -0.024942

D1 = 0.000025

D2 = -0.000123

References

[1]

Examples

```
>>> Lastovka_solid(300, 0.2139)
1682.0637469909211
```

chemicals.heat_capacity.**Lastovka_solid_integral**(*T*, *similarity_variable*, *MW=None*)

Integrates solid constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

uses an explicit form as derived with Sympy.

Parameters

T [float] Temperature of solid [K]

similarity_variable [float] similarity variable as defined in [1], [mol/g]

MW [float, optional] Molecular weight of the pure compound or mixture average, [g/mol]

Returns

H [float] Difference in enthalpy from 0 K, J/mol if MW given; J/kg otherwise

See also:

Lastovka_solid

Notes

Original model is in terms of J/g/K. Note that the model is for predicting mass heat capacity, not molar heat capacity like most other methods!

References

[1]

Examples

```
>>> Lastovka_solid_integral(300, 0.2139)
283246.1519409122
```

chemicals.heat_capacity.**Lastovka_solid_integral_over_T**(*T*, *similarity_variable*, *MW=None*)

Integrates over T solid constant-pressure heat capacity with the similarity variable concept and method as shown in [1].

uses an explicit form as derived with Sympy.

Parameters

T [float] Temperature of solid [K]

similarity_variable [float] similarity variable as defined in [1], [mol/g]

MW [float, optional] Molecular weight of the pure compound or mixture average, [g/mol]

Returns

S [float] Difference in entropy from 0 K, J/mol/K if MW given; J/kg/K otherwise

See also:

Lastovka_solid

Notes

Original model is in terms of J/g/K. Note that the model is for predicting mass heat capacity, not molar heat capacity like most other methods!

References

[1]

Examples

```
>>> Lastovka_solid_integral_over_T(300, 0.2139)
1947.5537561495564
```

1.12.7 Utility methods

class chemicals.heat_capacity.**PiecewiseHeatCapacity**(*models*)

Create a PiecewiseHeatCapacity object for calculating heat capacity and the enthalpy and entropy integrals using piecewise models.

Parameters

models [Iterable[HeatCapacity]] Piecewise heat capacity objects, [-]

Attributes

Tmax

Tmin

models

Methods

calculate(T)	Return the heat capacity as a function of temperature.
calculate_integral(Ta, Tb)	Return the enthalpy integral of heat capacity from *Ta* to *Tb*.
calculate_integral_over_T(Ta, Tb)	Return the entropy integral of heat capacity from *Ta* to *Tb*.
force_calculate(T)	Return the heat capacity as a function of temperature.
force_calculate_integral(Ta, Tb)	Return the enthalpy integral of heat capacity from *Ta* to *Tb*.
force_calculate_integral_over_T(Ta, Tb)	Return the entropy integral of heat capacity from *Ta* to *Tb*.

1.12.8 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.heat_capacity.**Cp_data_Poling**
: Constains data for gases and liquids from [3]. Simple polynomials for gas heat capacity (not suitable for extrapolation) are available for 308 chemicals. Additionally, constant values in at 298.15 K are available for 348 gases. Constant values in at 298.15 K are available for 245 liquids.

chemicals.heat_capacity.**TRC_gas_data**
: A rigorous expression from [1] for modeling gas heat capacity. Coefficients for 1961 chemicals are available.

chemicals.heat_capacity.**CRC_standard_data**
: Constant values tabulated in [4] at 298.15 K. Data is available for 533 gases. Data is available for 433 liquids. Data is available for 529 solids.

chemicals.heat_capacity.**Cp_dict_PerryI**
: Simple polynomials from [5] with vaious exponents selected for each expression. Coefficients are in units of calories/mol/K. The full expression is $C_p = a + bT + c/T^2 + dT^2$. Data is available for 284 compounds. Some compounds have gas data, some have liquid data, and have solid (crystal structure) data, sometimes multiple coefficients for different solid phases.

chemicals.heat_capacity.**zabransky_dicts**
: Complicated fits covering different cases and with different forms from [2].

chemicals.heat_capacity.**Cp_dict_characteristic_temperatures_adjusted_psi4_2022a**
: Theoretically calculated chatacteristic temperatures from vibrational frequencies using psi4

chemicals.heat_capacity.**Cp_dict_characteristic_temperatures_psi4_2022a**
: Theoretically calculated chatacteristic temperatures from vibrational frequencies using psi4, adjusted using a recommended coefficient

```
In [1]: import chemicals

In [2]: chemicals.heat_capacity.Cp_data_Poling
Out[2]:
                                Chemical   Tmin  ...     Cpg     Cpl
CAS                                               ...
56-23-5                tetrachloromethane  200.0  ...   83.43  131.60
60-29-7                     diethyl ether  100.0  ...  119.46  172.60
62-53-3           benzeneamine (aniline)   50.0  ...  107.90  191.90
64-17-5                          ethanol   50.0  ...   65.21  112.25
64-18-6      methanoic acid (formic acid)   50.0  ...   53.45   99.17
...                                   ...    ...  ...     ...     ...
14940-65-9                 tritium oxide    NaN  ...   34.96     NaN
16747-38-9    2,3,3,4-tetramethylpentane  200.0  ...  218.30  275.70
20291-95-6        2,2,5-trimethylheptane  200.0  ...  229.20  306.40
800000-51-5             hydrogen, normal    NaN  ...   28.83     NaN
800000-54-8            deuterium, normal    NaN  ...   29.20     NaN

[368 rows x 10 columns]

In [3]: chemicals.heat_capacity.TRC_gas_data
Out[3]:
                                Chemical   Tmin  ...       J       Hfg
CAS                                               ...
50-00-0                         Methanal   50.0  ...   3.46 -104700.0
```

(continues on next page)

(continued from previous page)

```
50-32-8                      Benzo[a]pyrene  298.0  ...  13.44  324000.0
53-70-3                Dibenz[a,h]anthracene  298.0  ...  16.63  375000.0
56-23-5                   Tetrachloromethane  200.0  ...   9.58  -93700.0
56-55-3                    Benz[a]anthracene  298.0  ...  11.45  328000.0
...                                      ...    ...  ...    ...       ...
800000-46-8  2,2,(3RS,4RS)-Tetramethylhexane  200.0  ...  22.45 -188600.0
800000-47-9  2,(3RS,4SR),5-Tetramethylhexane  200.0  ...  22.32  193700.0
800000-48-0  2,(3RS,4RS),5-Tetramethylhexane  200.0  ...  22.14 -194600.0
800000-56-0            1-Methylbutyl radical  200.0  ...  22.25   54600.0
800002-32-8           Propenoic acid (Dimer)   50.0  ...  13.83 -686000.0

[1961 rows x 14 columns]

In [4]: chemicals.heat_capacity.CRC_standard_data
Out[4]:
                              Chemical        Hfs  ...    S0g    Cpg
CAS                                                ...
50-00-0                     Formaldehyde        NaN  ...  218.8   35.4
50-32-8                   Benzo[a]pyrene        NaN  ...    NaN  254.8
50-69-1                        D-Ribose -1047200.0  ...    NaN    NaN
50-78-2        2-(Acetyloxy)benzoic acid  -815600.0  ...    NaN    NaN
50-81-7                  L-Ascorbic acid -1164600.0  ...    NaN    NaN
...                                  ...        ...  ...    ...    ...
92141-86-1             Cesium metaborate  -972000.0  ...    NaN    NaN
99685-96-8        Carbon [fullerene-C60]  2327000.0  ...  544.0  512.0
114489-96-2  Isobutyl 2-chloropropanoate        NaN  ...    NaN    NaN
115383-22-7       Carbon [fullerene-C70]  2555000.0  ...  614.0  585.0
116836-32-9         sec-Butyl pentanoate        NaN  ...    NaN    NaN

[2470 rows x 13 columns]

In [5]: chemicals.heat_capacity.Cp_dict_PerryI['124-38-9'] # gas only
Out[5]:
{'g': {'Formula': 'CO2',
  'Phase': 'g',
  'Subphase': None,
  'Const': 10.34,
  'Lin': 0.00274,
  'Quadinv': -195500.0,
  'Quad': 0,
  'Tmin': 273.0,
  'Tmax': 1200.0,
  'Error': '1a'}}

In [6]: chemicals.heat_capacity.Cp_dict_PerryI['7704-34-9'] # crystal and gas
Out[6]:
{'g': {'Formula': 'H2S',
  'Phase': 'g',
  'Subphase': None,
  'Const': 7.2,
  'Lin': 0.0036,
  'Quadinv': 0,
```

(continues on next page)

(continued from previous page)

```
  'Quad': 0,
  'Tmin': 300.0,
  'Tmax': 600.0,
  'Error': 8.0},
 'c': {'Formula': 'S',
  'Phase': 'c',
  'Subphase': 'monoclinic',
  'Const': 4.38,
  'Lin': 0.0044,
  'Quadinv': 0,
  'Quad': 0,
  'Tmin': 368.0,
  'Tmax': 392.0,
  'Error': 3.0}}

In [7]: chemicals.heat_capacity.Cp_dict_PerryI['7440-57-5'] # crystal and liquid
Out[7]:
{'c': {'Formula': 'Au',
  'Phase': 'c',
  'Subphase': None,
  'Const': 5.61,
  'Lin': 0.00144,
  'Quadinv': 0,
  'Quad': 0,
  'Tmin': 273.0,
  'Tmax': 1336.0,
  'Error': 2.0},
 'l': {'Formula': 'Au',
  'Phase': 'l',
  'Subphase': None,
  'Const': 7.0,
  'Lin': 0,
  'Quadinv': 0,
  'Quad': 0,
  'Tmin': 1336.0,
  'Tmax': 1573.0,
  'Error': 5.0}}

In [8]: chemicals.heat_capacity.zabransky_dicts.keys()
Out[8]: dict_keys(['Zabransky spline, averaged heat capacity', 'Zabransky␣
↪quasipolynomial, averaged heat capacity', 'Zabransky spline, constant-pressure',
↪'Zabransky quasipolynomial, constant-pressure', 'Zabransky spline, saturation',
↪'Zabransky quasipolynomial, saturation'])
```

1.13 IAPWS: International Association for the Properties of Water and Steam (chemicals.iapws)

This module contains the core of the IAPWS-95 and IAPWS-97 standards. The objective of this module is to contain extremely fast functions to calculate several basic properties of water.

The simplest interfaces are `iapws95_rho` for density calculation only and `iapws95_properties` for some basic properties.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.13.1 IAPWS-95 Basic Solvers

chemicals.iapws.**iapws95_rho**(*T*, *P*)

Calculate the density of water according to the IAPWS-95 standard given a temperature *T* and pressure *P*. The phase is determined in this calculation.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water, [kg/m^3]

See also:

`iapws95_rhol_sat`

`iapws95_rhog_sat`

Notes

There is a sudden transition at the saturation pressure between liquid and vapor density, by design.

This solution is iterative due to the nature of the equation. The solution procedure begins with IAPWS-97's explicit equations as an initial guess, extrapolating when out of range. If the temperature is under the critical temperature, the saturation density is calculated, and used to ensure the solver begins in the feasible region. Newton's method converges extremely, normally after 2 or 3 iterations.

Temperatures under 273.15 K are not officially supported by [1], but a solution is still attempted down to 235 K.

References

[1]

Examples

```
>>> iapws95_rho(T=300.0, P=1e6)
996.96002269499
```

1 GPa and 5000 K are suggested as upper limits of [1] although there are no hardcoded limits for temperature and pressure.

```
>>> iapws95_rho(T=5000.0, P=1e9)
326.79451662743
```

chemicals.iapws.**iapws95_P**(*T*, *rho*)

Calculate the pressure of water according to the IAPWS-95 standard given a temperature *T* and mass density *rho*.

Parameters

T [float] Temperature, [K]

rho [float] Mass density of water, [kg/m^3]

Returns

P [float] Pressure, [Pa]

Notes

The IAPWS-95 model is explicit with inputs of temperature and density, so this is a direct calculation with no iteration required.

References

[1]

Examples

```
>>> iapws95_P(330.0, iapws95_rho(T=330.0, P=8e5))
8e5
>>> iapws95_P(823.0, 40.393893559703734)
14e6
```

Not all temperature and density inputs provide a stable solution; for example anything between the vapor and gas saturation curves. In some but not all of these cases a negative pressure is returned:

```
>>> iapws95_P(T=300, rho=300)
-1.526394720e+23
```

chemicals.iapws.**iapws95_T**(*P*, *rho*)

Calculate the temperature of water according to the IAPWS-95 standard given a density *rho* and pressure *P*.

Parameters

P [float] Pressure, [Pa]

rho [float] Mass density of water, [kg/m^3]

Returns

T [float] Temperature, [K]

Notes

This solution is iterative due to the nature of the equation. The solution procedure begins with IAPWS-97's equations as an initial guess, extrapolating when out of range. Newton's method converges extremely, normally after 2 or 3 iterations.

Due to water's unique density curve, there is a temperature region spanning 273.15 K to 280.005 K where there are two solutions. No guarantee is made as to which solution will be returned.

References

[1]

Examples

```
>>> iapws95_T(P=1e6, rho=995.0)
306.461547194
```

1.13.2 IAPWS-97 Basic Solvers

chemicals.iapws.**iapws97_rho**(*T*, *P*, *use_95_boundary=False*)

Calculate the density of water in kg/m^3 according to the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

use_95_boundary [bool, optional] If True, respect the IAPWS-95 vapor pressure curve instead of the IF-97 one, [-]

Returns

rho [float] Mass density of water, [kg/m^3]

Notes

The range of validity of this formulation is as follows:

For $P \leq 100$ MPa:

$$273.15 \text{ K} \leq T \leq 1073.15 \text{ K}$$

For $P \leq 50$ MPa:

$$1073.15 \text{ K} \leq T \leq 2273.15 \text{ K}$$

A ValueError is raised if the temperature or the pressure is out of bounds.

IAPWS is implemented in four regions in the T-P domain: Region 1 (liquid), region 2 (gas and supercritical gas), region 5 (high temperature gas), and region 3 (near-critical). Significant discontinuities exist between the transitions of each regions. In region 3, there are 26 sub-regions and the correlation has the least accuracy.

For many applications, the discontinuities in IF-97 can be problematic and the slower IAPWS-95 must be used. IAPWS-95 also has a wider range of applicability.

References

[1]

Examples

```
>>> iapws97_rho(648.6, 22.5e6)
353.06081088726
>>> iapws97_rho(330.0, 8e5)
985.10498080770
>>> iapws97_rho(823.0, 14e6)
40.39293607288123
>>> iapws97_rho(2000.0, 3e7)
32.11456228328856
```

chemicals.iapws.**iapws97_P**(*T*, *rho*)

Calculate the pressure of water according to the IAPWS-97 standard given a temperature T and mass density *rho*.

Parameters

T [float] Temperature, [K]

rho [float] Mass density of water, [kg/m^3]

Returns

P [float] Pressure, [Pa]

Notes

The range of validity of this formulation is as follows:

For $P \leq 100$ MPa:

$$273.15 \text{ K} \leq T \leq 1073.15 \text{ K}$$

For $P \leq 50$ MPa:

$$1073.15 \text{ K} \leq T \leq 2273.15 \text{ K}$$

A ValueError is raised if the temperature or density is out of bounds.

Newton's method with analytical derivatives is used here to solve these equations. The solver tolerance is as tight as it can be without causing wasted iterations that do not improve the result at all. Pressure changes quickly with density however, and some discrepancy between solvers is to be expected.

For region 3, there are really two formulations present in IAPWS-97. There is a Helmholtz energy equation (Temperature and density dependent), and also 26 separate backwards equations for *rho* which depend on *T* and *P*. The Helmholtz energy equation is much more accurate and does not have discontinuities. The two sets of equations agree closely not not perfectly. By design, *iapws97_rho* implements the 26 T-P equations and this implements the Helmholtz energy equation. This means that in region 3 solutions will not be consistent. For consistency requirements, IAPWS-95 is recommended.

This solver does not have any issues with multiple solutions. The solvers have been checked to achieve a relative solution tolerance of 5e-9 on 100 million points.

References

[1]

Examples

```
>>> iapws97_P(330.0, iapws97_rho(T=330.0, P=8e5))
8e5
>>> iapws97_P(823.0, 40.39293607288123)
14e6
>>> iapws97_P(T=2000.0, rho=32.11456228328856)
3e7
```

Region 3 point - does not implement the same equations as *iapws97_rho*!

```
>>> iapws97_P(648.6, iapws97_rho(T=648.6, P=22.5e6))
22499974.093936257
```

chemicals.iapws.**iapws97_T**(*P*, *rho*)

Calculate the temperature of water according to the IAPWS-97 standard given a pressure *P* and mass density *rho*.

Parameters

P [float] Pressure, [Pa]

rho [float] Mass density of water, [kg/m^3]

Returns

T [float] Temperature, [K]

Notes

The range of validity of this formulation is as follows:

For $P \leq 100$ MPa:

$$273.15 \text{ K} \leq T \leq 1073.15 \text{ K}$$

For $P \leq 50$ MPa:

$$1073.15 \text{ K} \leq T \leq 2273.15 \text{ K}$$

A ValueError is raised if the pressure or density is out of bounds.

Newton's method with analytical derivatives is used here to solve these equations. The solver tolerance is as tight as it can be without causing wasted iterations that do not improve the result at all.

Due to water's unique density curve, there is a temperature region spanning 273.15 K to 280.005 K where there are two solutions. No guarantee is made as to which solution will be returned.

References

[1]

Examples

```
>>> iapws97_T(8e5, iapws97_rho(T=330.0, P=8e5))
330.0
>>> iapws97_T(14e6, 40.39293607288123)
823.0
>>> iapws97_T(P=3e7, rho=32.11456228328856)
2000.0
```

1.13.3 IAPWS-95 Properties

chemicals.iapws.**iapws95_properties**(*T*, *P*)

Calculate some basic properties of water according to the IAPWS-95 standard given a temperature *T* and pressure *P*.

The properties are density *rho*, internal energy *U*, entropy *S*, enthalpy *H*, isochoric heat capacity *Cv*, isobaric heat capacity *Cp*, speed of sound *w*, Joule-Thomson coefficient *JT*, isothermal throttling coefficient *delta_T*, isentropic temperature-pressure coefficient *beta_s*, and the derivative of mass density with respect to pressure at constant temperature *drho_dP*.

This function is intended as a demonstration of how to use the IAPWS-95 equations. For that reason, mass-units are used in all returned variables.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water, [kg/m^3]

U [float] Internal energy of water, [J/(kg)]

S [float] Entropy of water, [J/(kg*K)]

H [float] Enthalpy of water, [J/(kg)]

Cv [float] Isochoric heat capacity, [J/(kg*K)]

Cp [float] Isobaric heat capacity, [J/(kg*K)]

w [float] Speed of sound, [m/s]

JT [float] Joule-Thomson coefficient, [K/Pa]

delta_T [float] Isothermal throttling coefficient, [J/(kg*Pa)]

beta_s [float] Isentropic temperature-pressure coefficient, [K/Pa]

drho_dP [float] Derivative of mass density with respect to pressure at constant temperature, [kg/(m^3*Pa)]

Notes

Hundreds of useful properties can be obtained from the IAPWS-95 model. It is intended for this function to serve as a useful starting point to those. Calculating every property with every set of units is beyond the scope of *chemicals*. The functions like *iapws95_dAr_ddelta* can be used directly in your own implementation - where you can calculate only those properties which are necessary, for maximum speed.

The formulas are as follows:

$$\frac{u(\delta,\tau)}{RT} = \tau\left(\phi_\tau^{\mathrm{o}} + \phi_\tau^{\mathrm{r}}\right)$$

$$\frac{s(\delta,\tau)}{R} = \tau\left(\phi_\tau^{\mathrm{o}} + \phi_\tau^{\mathrm{r}}\right) - \phi^{\mathrm{o}} - \phi^{\mathrm{r}}$$

$$\frac{h(\delta,\tau)}{RT} = 1 + \tau\left(\phi_\tau^{\mathrm{o}} + \phi_\tau^{\mathrm{r}}\right) + \delta\phi_\delta^{\mathrm{r}}$$

$$\frac{c_v(\delta,\tau)}{R} = -\tau^2\left(\phi_{\tau\tau}^{\mathrm{o}} + \phi_{\tau\tau}^{\mathrm{r}}\right)$$

$$\frac{c_p(\delta,\tau)}{R} = -\tau^2\left(\phi^{\mathrm{o}}_{\tau\tau} + \phi^{\mathrm{r}}_{\tau\tau}\right) + \frac{\left(1+\delta\phi^{\mathrm{r}}_{\delta} - \delta\tau\phi^{\mathrm{r}}_{\delta\tau}\right)^2}{1 + 2\delta\phi^{\mathrm{r}}_{\delta} + \delta^2\phi^{\mathrm{r}}_{\delta\delta}}$$

$$\frac{w^2(\delta,\tau)}{RT} = 1 + 2\delta\phi^{\mathrm{r}}_{\delta} + \delta^2\phi^{\mathrm{r}}_{\delta\delta} - \frac{\left(1+\delta\phi^{\mathrm{r}}_{\delta} - \delta\tau\phi^{\mathrm{r}}_{\delta\tau}\right)^2}{\tau^2\left(\phi^{\mathrm{o}}_{\tau\tau} + \phi^{\mathrm{r}}_{\tau\tau}\right)}$$

$$\mu R\rho = \frac{-\left(\delta\phi^{\mathrm{r}}_{\delta} + \delta^2\phi^{\mathrm{r}}_{\delta\delta} + \delta\tau\phi^{\mathrm{r}}_{\delta\tau}\right)}{\left(1+\delta\phi^{\mathrm{r}}_{\delta} - \delta\tau\phi^{\mathrm{r}}_{\delta\tau}\right)^2 - \tau^2\left(\phi^{\mathrm{o}}_{\tau\tau} + \phi^{\mathrm{r}}_{\tau\tau}\right)\left(1 + 2\delta\phi^{\mathrm{r}}_{\delta} + \delta^2\phi^{\mathrm{r}}_{\delta\delta}\right)}$$

$$\delta_T\rho = 1 - \frac{1+\delta\phi^{\mathrm{r}}_{\delta} - \delta\tau\phi^{\mathrm{r}}_{\delta\tau}}{1 + 2\delta\phi^{\mathrm{r}}_{\delta} + \delta^2\phi^{\mathrm{r}}_{\delta\delta}}$$

$$\beta_S\rho R = \frac{1+\delta\phi^{\mathrm{r}}_{\delta} - \delta\tau\phi^{\mathrm{r}}_{\delta\tau}}{\left(1+\delta\phi^{\mathrm{r}}_{\delta} - \delta\tau\phi^{\mathrm{r}}_{\delta\tau}\right)^2 - \tau^2\left(\phi^{\mathrm{o}}_{\tau\tau} + \phi^{\mathrm{r}}_{\tau\tau}\right)\left(1 + 2\delta\phi^{\mathrm{r}}_{\delta} + \delta^2\phi^{\mathrm{r}}_{\delta\delta}\right)}$$

This derivative isn't part of the same table of properties, but it is needed by the transport calculation routines:

$$\left(\frac{\partial\rho}{\partial P}\right)_T = \frac{1}{RT\left(1 + 2\delta\alpha^{\mathrm{r}}_{\delta} + \delta^2\alpha^{\mathrm{r}}_{\delta\delta}\right)}$$

References

[1]

Examples

```
>>> iapws95_properties(T=300.0, P=1e6)
(996.96002269, 112478.998245, 392.813902893, 113482.047492, 4127.21730497, 4178.
↪103605593, 1503.035983829, -2.202166728257e-07, 0.000920088074745, 1.
↪985617879134e-08, 4.48108429028e-07)
```

```
>>> rho, U, S, H, Cv, Cp, w, JT, delta_T, beta_s, drho_dP = iapws95_
↪properties(T=500.0, P=1e5)
>>> w
548.3138393244
```

1.13.4 IAPWS Saturation Pressure/Temperature

chemicals.iapws.**iapws95_Psat**(T)

Compute the saturation pressure of the IAPWS-95 equation using high- fidelity polynomial fits. These have a relative accuracy of under 1e-12, and are generated by solving the saturation equations under the high-precision environment of mpmath. The range of the fit is 235 K to 647.096 K, the critical point.

$$P_{sat} = P_c \exp(\text{polynomial}(a(T-b)))$$

Parameters

T [float] Temperature at which to calculate the saturation condition, [K]

Returns

Psat [float] Saturation vapor pressure, [Pa]

See also:

iapws95_saturation

Notes

This method should be used in preference to *iapws95_saturation*. Although using mpmath generates slightly different results than using plain floating point numbers, the requirement for the saturation curve is to be smooth, and continuous; mpmath makes this easy and the saturation equations were solved extremely high precision, well under a floating point's error.

The polynomial coefficients have been carefully chosen to be able to be evaluated accurately with horner's method, although they are derived as a Chebyshev approximation originally.

Examples

```
>>> iapws95_Psat(400.0)
245769.3455
```

chemicals.iapws.**iapws95_dPsat_dT**(*T*)

Compute the temperature derivative of saturation pressure of the IAPWS-95 equation using high- fidelity polynomial fits. The range of the fit is 235 K to 647.096 K, the critical point.

$$P_{sat} = P_c \exp(\text{polynomial}(a(T-b)))$$

$$\frac{\partial P_{sat}}{\partial T} = aP_c \exp(\text{polynomial}(a(T-b))) \exp\left(\frac{\partial \text{polynomial}(a(T-b))}{\partial T}\right)$$

Parameters

T [float] Temperature at which to calculate the saturation condition and its temperature derivative, [K]

Returns

dPsat_dT [float] First temperature derivative of Saturation vapor pressure, [Pa/K]

Psat [float] Saturation vapor pressure, [Pa]

Notes

Psat must be calculated in the calculation of the derivative, so it is returned as well which may be useful in some applications.

Examples

```
>>> iapws95_dPsat_dT(400.0)
(7483.62075827, 245769.3455657)
```

chemicals.iapws.**iapws92_Psat**(*T*)

Compute the saturation pressure of the IAPWS-92 equation.

$$P_{sat} = P_c \exp\left(\frac{T_c}{T}[a_1\tau + a_2\tau^{1.5} + a_3\tau^3 + a_4\tau^{3.5} a_5\tau^4 + a_6\tau^{7.5}]\right)$$

Parameters

T [float] Temperature at which to calculate the saturation condition and its temperature derivative, [K]

Returns

Psat [float] Saturation vapor pressure, [Pa]

Notes

The coefficients are [-7.85951783, 1.84408259, -11.7866497, 22.6807411, -15.9618719, 1.80122502]

Examples

```
>>> iapws92_Psat(400.0)
245765.2635418
```

chemicals.iapws.**iapws92_dPsat_dT**(T)

Compute the temperature derivative of saturation pressure of the IAPWS-92 equation.

$$P_{sat} = P_c \exp\left(\frac{T_c}{T}[a_1\tau + a_2\tau^{1.5} + a_3\tau^3 + a_4\tau^{3.5}a_5\tau^4 + a_6\tau^{7.5}]\right)$$

Parameters

T [float] Temperature at which to calculate the saturation condition and its temperature derivative, [K]

Returns

dPsat_dT [float] First temperature derivative of saturation vapor pressure, [Pa/K]

Psat [float] Saturation vapor pressure, [Pa]

Notes

The coefficients are [-7.85951783, 1.84408259, -11.7866497, 22.6807411, -15.9618719, 1.80122502]

Examples

```
>>> iapws92_dPsat_dT(400.0)
(7483.47094105, 245765.263541)
```

chemicals.iapws.**iapws95_Tsat**(P)

Compute the saturation temperature of the IAPWS-95 equation. The range of the fit is 235 K to 647.096 K, the critical point.

Parameters

Psat [float] Saturation vapor pressure specified, [Pa]

Returns

T [float] Temperature at which the saturation pressure occurs, [K]

See also:

iapws95_Psat

Tsat_IAPWS

Notes

This method is quite fast and precise because it starts with great initial guesses and the equation is well-bounded. The precision of this calculation should be the same as *iapws95_Psat*.

Examples

```
>>> iapws95_Tsat(iapws95_Psat(400.0))
400.0
```

chemicals.iapws.**iapws95_saturation**(*T*, *xtol=1e-05*, *rhol_guess=None*, *rhog_guess=None*)

Solve the vapor-liquid saturation equations of IAPWS-95 given a specified temperature. With floating point numbers, the achievable tolerance is somewhat low so *xtol* is exposed as a setting - it can be adjusted somewhat. Density guesses may be provided, otherwise they will be estimated.

$$G_{liq}(T, \rho_l) = G_{vap}(T, \rho_g)$$

$$P_{liq}(T, \rho_l) = P_{vap}(T, \rho_g)$$

Parameters

T [float] Temperature at which to solve for saturation condition, [K]

xtol [float] Tolerance for solver, [-]

rhol_guess [float, optional] Liquid density of water at saturation (guess), [kg/m^3]

rhog_guess [float, optional] Vapor density of water at saturation (guess), [kg/m^3]

Returns

Psat [float] Saturation vapor pressure, 3[Pa]

rhol [float] Saturation liquid water density, [kg/m^3]

rhog [float] Saturation vapor water density, [kg/m^3]

Notes

This is not a perfect function.

With *mpmath* multiple precision, the equation can be solved down to 233.6 K and up to 647.095999995 K - within 10 parts in a billion of the critical point exactly.

Reasons for non-convergence include floating point issues as delta becomes 1, and zero division errors in the matrix inverse.

Examples

```
>>> iapws95_saturation(400.0, xtol=1e-6)
(245769.345, 937.4860, 1.3694075)
>>> iapws95_saturation(647.0955, xtol=1e-7)
(22063866.35, 325.70, 318.277)
```

chemicals.iapws.**iapws11_Psub**(T)

Compute the sublimation pressure of the frozen water using the IAPWS-11 equation from the Revised Release on the Pressure along the Melting and Sublimation Curves of Ordinary Water Substance.

$$P_{sub} = P_t \exp\left(\theta^{-1} \sum_{i=1}^{3} a_i \theta^{b_i}\right)$$

$$\theta = \frac{T}{T_t}$$

Parameters

T [float] Temperature at which to calculate the sublimation condition [K]

Returns

Psub [float] Sublimation vapor pressure, [Pa]

Notes

The triple temperature is 273.16 K, and triple pressure 611.657 Pa.

The coefficients are as follows:

ais = [-0.212144006E2, 0.273203819E2, -0.610598130E1]

bis = [0.333333333E-2, 0.120666667E1, 0.170333333E1]

The equation is valid from 50 K to the triple temperature.

Examples

```
>>> iapws11_Psub(230.0)
8.947352740189151
```

1.13.5 IAPWS Saturation Density

chemicals.iapws.**iapws95_rhol_sat**(T)

Compute the saturation liquid density of the IAPWS-95 equation using high- fidelity polynomial fits. These have a relative accuracy of under 1e-13, except near the critical point where it rises to 1e-10, and are generated by solving the saturation equations under the high-precision environment of mpmath. The range of the fit is 235 K to 647.096 K, the critical point.

Parameters

T [float] Temperature at which to calculate the saturation condition, [K]

Returns

rhol [float] Saturation liquid density, [kg/m^3]

See also:

iapws92_rhol_sat

Notes

This method should be used in preference to *iapws92_rhol_sat*.

Examples

```
>>> iapws95_rhol_sat(400.0)
937.48603939
```

chemicals.iapws.**iapws95_rhog_sat**(*T*)

Compute the saturation vapor density of the IAPWS-95 equation using high- fidelity polynomial fits. These have a relative accuracy of under 1e-13, except near the critical point where it rises to 1e-10, and are generated by solving the saturation equations under the high-precision environment of mpmath. The range of the fit is 235 K to 647.096 K, the critical point.

Parameters

T [float] Temperature at which to calculate the saturation condition, [K]

Returns

rhol [float] Saturation vapor density, [kg/m^3]

See also:

iapws92_rhog_sat

Notes

This method should be used in preference to *iapws92_rhog_sat*.

Examples

```
>>> iapws95_rhog_sat(400.0)
1.3694075410
```

chemicals.iapws.**iapws95_drhol_sat_dT**(*T*)

Compute the first temperature derivative of saturation liquid density of the IAPWS-95 equation using high-fidelity polynomial fits. The actual saturated liquid density is returned as well.

The range of the fit is 235 K to 647.096 K, the critical point.

Parameters

T [float] Temperature at which to calculate the saturation condition and its derivative, [K]

Returns

drhol_dT [float] First temperature derivative of saturation liquid density, [kg/(m^3*K)]

rhol [float] Saturation liquid density, [kg/m^3]

Examples

```
>>> iapws95_drhol_sat_dT(400.0)
(-0.835194603380, 937.486039392)
```

chemicals.iapws.**iapws92_rhol_sat**(T)

Calculates saturation liquid mass density of water using the IAPWS SR1-86(1992) [1] [2] explicit equation.

$$\frac{\rho_l^{sat}}{\rho_c} = 1 + b_1\tau^{1/3} + b_2\tau^{2/3} + b_3\tau^{5/3} + b_4\tau^{16/3} + b_5\tau^{43/3} + b_6\tau^{110/3}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of water, [K]

Returns

rhol_sat [float] Saturation liquid mass density of water [kg/m^3]

See also:

iapws95_rhol_sat

Notes

This equation is fit to experimental data to within its accuracy. It does not satisfy the equilibrium conditions for the IAPWS-95 or IAPWS-97 formulations.

The values of the constants are as follows:

b1 = 1.99274064; b2 = 1.09965342; b3 = -0.510839303; b4 = -1.75493479; b5 = -45.5170352; b6 = -6.74694450e5

References

[1], [2]

Examples

```
>>> iapws92_rhol_sat(300.)
996.5089712803
```

chemicals.iapws.**iapws92_rhog_sat**(T)

Calculates saturation vapor mass density of water using the IAPWS SR1-86(1992) [1] [2] explicit equation.

$$\ln\left(\frac{\rho_g^{sat}}{\rho_c}\right) = 1 + c_1\tau^{2/6} + c_2\tau^{4/6} + c_3\tau^{8/6} + c_4\tau^{18/6} + c_5\tau^{37/6} + c_6\tau^{71/6}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of water, [K]

Returns

rhog_sat [float] Saturation vapor mass density of water [kg/m^3]

See also:

iapws95_rhog_sat

Notes

This equation is fit to experimental data to within its accuracy. It does not satisfy the equilibrium conditions for the IAPWS-95 or IAPWS-97 formulations.

The values of the constants are as follows:

c1 = -2.03150240; c2 = -2.68302940; c3 = -5.38626492; c4 = -17.2991605; c5 = -44.7586581; c6 = -63.9201063

References

[1], [2]

Examples

```
>>> iapws92_rhog_sat(300.)
0.0255887212886
```

1.13.6 IAPWS Constants

chemicals.iapws.**iapws95_Tc = 647.096**
Critical temperature of water in K according to IAPWS-95, also used in IAPWS-97

chemicals.iapws.**iapws95_Pc = 22064000.0**
Critical pressure of water in Pa according to IAPWS-95, also used in IAPWS-97

chemicals.iapws.**iapws95_rhoc = 322.0**
Critical density of water in kg/m^3 according to IAPWS-95, also used in IAPWS-97

chemicals.iapws.**iapws95_MW = 18.015268**
Molecular weight of water in g/mol according to IAPWS-95, also used in IAPWS-97

chemicals.iapws.**iapws95_R = 461.51805**
Specific gas constant in J/(kg*K) according to IAPWS-95

chemicals.iapws.**iapws97_R = 461.526**
Specific gas constant in J/(kg*K) according to IAPWS-97

chemicals.iapws.**iapws95_Tt = 273.16**
Triple temperature of water in K according to IAPWS

1.13.7 IAPWS-97 Region 1

chemicals.iapws.**iapws97_G_region1**(*tau*, *pi*)

Calculates the dimensionless Gibbs free energy for water according to the IAPWS-97 standard (for region 1).

$$\gamma = \sum_{i=1}^{34} I_i (7.1 - \pi)^{I_i} (\tau - 1.222)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (1386 K)/T [-]

pi [float] Dimensionless pressure, P/(16.53 MPa), [-]

Returns

G [float] Dimensionless Gibbs energy G/(RT), [-]

Examples

```
>>> iapws97_G_region1(1386/277.15, 101325/16.53E6)
-0.00016341033954414
```

chemicals.iapws.**iapws97_dG_dpi_region1**(*tau*, *pi*)

Calculates the derivative of dimensionless Gibbs free energy with respect to *pi* for water according to the IAPWS-97 standard (for region 1).

$$\frac{\partial \gamma}{\partial \pi} = \sum_{i=1}^{34} -n_i I_i (7.1 - \pi)^{I_i - 1} (\tau - 1.222)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (1386 K)/T [-]

pi [float] Dimensionless pressure, P/(16.53 MPa), [-]

Returns

dG_dpi [float] Derivative of dimensionless Gibbs energy G/(RT) with respect to *pi*, [-]

Notes

Used in density solution. This contains a hand-optimized implementation with a single division, no power operations, 65 multiplications, 16 local variables, and a minimum number of additions.

Examples

```
>>> iapws97_dG_dpi_region1(1386/277.15, 101325/16.53E6)
0.1292327182544
```

chemicals.iapws.**iapws97_d2G_dpi2_region1**(*tau*, *pi*)

Calculates the second derivative of dimensionless Gibbs free energy with respect to *pi* for water according to the IAPWS-97 standard (for region 1).

$$\frac{\partial^2 \gamma}{\partial \pi^2} = \sum_{i=1}^{34} n_i I_i (I_i - 1)(7.1 - \pi)^{I_i - 2} (\tau - 1.222)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (1386 K)/T [-]

pi [float] Dimensionless pressure, P/(16.53 MPa), [-]

Returns

d2G_dpi2 [float] Second Derivative of dimensionless Gibbs energy G/(RT) with respect to *pi*, [-]

Examples

```
>>> iapws97_d2G_dpi2_region1(1386/277.15, 101325/16.53E6)
 -0.0010570100274769
```

chemicals.iapws.**iapws97_dG_dtau_region1**(*tau*, *pi*)

Calculates the derivative of dimensionless Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 1).

$$\frac{\partial \gamma}{\partial \tau} = \sum_{i=1}^{34} n_i (7.1 - \pi)^{I_i} J_i (\tau - 1.222)^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (1386 K)/T [-]

pi [float] Dimensionless pressure, P/(16.53 MPa), [-]

Returns

dG_dtau [float] Derivative of dimensionless Gibbs energy G/(RT) with respect to *tau*, [-]

Examples

```
>>> iapws97_dG_dtau_region1(1386/277.15, 101325/16.53E6)
0.026440334282967
```

chemicals.iapws.**iapws97_d2G_dtau2_region1**(*tau*, *pi*)

Calculates the second derivative of dimensionless Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 1).

$$\frac{\partial^2 \gamma}{\partial \tau^2} = \sum_{i=1}^{34} n_i (7.1 - \pi)^{I_i} J_i (J_i - 1) (\tau - 1.222)^{J_i - 2}$$

Parameters

tau [float] Dimensionless temperature, (1386 K)/T [-]

pi [float] Dimensionless pressure, P/(16.53 MPa), [-]

Returns

d2G_dtau2 [float] Second Derivative of dimensionless Gibbs energy G/(RT) with respect to *tau*, [-]

Examples

```
>>> iapws97_d2G_dtau2_region1(1386/277.15, 101325/16.53E6)
-0.3645169808573
```

chemicals.iapws.**iapws97_d2G_dpidtau_region1**(*tau*, *pi*)

Calculates the second derivative of dimensionless Gibbs free energy with respect to *tau* and *pi* for water according to the IAPWS-97 standard (for region 1).

$$\frac{\partial^2 \gamma}{\partial \tau \partial \pi} = \sum_{i=1}^{34} -n_i I_i (7.1 - \pi)^{I_i} J_i (\tau - 1.222)^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (1386 K)/T [-]

pi [float] Dimensionless pressure, P/(16.53 MPa), [-]

Returns

d2G_dpidtau [float] Second Derivative of dimensionless Gibbs energy G/(RT) with respect to *tau* and *pi*, [-]

Examples

```
>>> iapws97_d2G_dpidtau_region1(1386/277.15, 101325/16.53E6)
0.025837659858819
```

1.13.8 IAPWS-97 Region 2

chemicals.iapws.**iapws97_G0_region2**(*tau*, *pi*)

Calculates the dimensionless ideal gas Gibbs free energy for water according to the IAPWS-97 standard (for region 2).

$$\gamma^\circ = \ln \pi + \sum_{i=1}^{9} n_i^\circ \tau^{J_i^\circ}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

G0 [float] Dimensionless ideal gas Gibbs energy G0/(RT), [-]

Examples

```
>>> iapws97_G0_region2(540/300.0, 101325/1e6)
3.3180953922351
```

chemicals.iapws.**iapws97_dG0_dtau_region2**(*tau*, *pi*)

Calculates the first derivative of dimensionless ideal gas Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial \gamma^\circ}{\partial \tau} = \sum_{i=1}^{9} n_i^\circ J_i^\circ \tau^{J_i^\circ - 1}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

dG0_dtau [float] First derivative of dimensionless ideal gas Gibbs energy G0/(RT) with respect to *tau*, [-]

Notes

This function does not depend on *pi* but it is accepted for consistency.

Examples

```
>>> iapws97_dG0_dtau_region2(540/300.0, 101325/1e6)
10.2374188173906
```

chemicals.iapws.**iapws97_d2G0_dtau2_region2**(*tau*, *pi*)

Calculates the second derivative of dimensionless ideal gas Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial^2 \gamma^\circ}{\partial \tau^2} = \sum_{i=1}^{9} n_i^\circ J_i^\circ (J_i^\circ - 1) \tau^{J_i^\circ - 2}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2G0_dtau2 [float] Second derivative of dimensionless ideal gas Gibbs energy G0/(RT) with respect to *tau*, [-]

Notes

This function does not depend on *pi* but it is accepted for consistency.

Examples

```
>>> iapws97_d2G0_dtau2_region2(540/300.0, 101325/1e6)
-1.2472096479372
```

chemicals.iapws.**iapws97_Gr_region2**(*tau*, *pi*)

Calculates the dimensionless residual Gibbs free energy for water according to the IAPWS-97 standard (for region 2).

$$\gamma^r = \sum_{i=1}^{43} n_i \pi^{I_i} (\tau - 0.5)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

Gr [float] Dimensionless residual Gibbs energy Gr/(RT), [-]

Examples

```
>>> iapws97_Gr_region2(540/300.0, 101325/1e6)
-0.71851548053980
```

chemicals.iapws.**iapws97_dGr_dpi_region2**(*tau*, *pi*)

Calculates the first derivative of dimensionless residual Gibbs free energy with respect to *pi* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial \gamma^r}{\partial \pi} = \sum_{i=1}^{43} n_i I_i \pi^{I_i - 1} (\tau - 0.5)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

dGr_dpi [float] Derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *pi*, [-]

Notes

Used in density solution.

Examples

```
>>> iapws97_dGr_dpi_region2(540/300.0, 101325/1e6)
-27.7714056629532
```

chemicals.iapws.**iapws97_d2Gr_dpi2_region2**(*tau*, *pi*)

Calculates the second derivative of dimensionless residual Gibbs free energy with respect to *pi* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial^2 \gamma^r}{\partial \pi^2} = \sum_{i=1}^{43} n_i I_i (I_i - 1) \pi^{I_i - 2} (\tau - 0.5)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2Gr_dpi2 [float] Second Derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *pi*, [-]

Examples

```
>>> iapws97_d2Gr_dpi2_region2(540/300.0, 101325/1e6)
-983.15187604898
```

chemicals.iapws.**iapws97_dGr_dtau_region2**(*tau*, *pi*)

Calculates the first derivative of dimensionless residual Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial \gamma^r}{\partial \tau} = \sum_{i=1}^{43} n_i \pi^{I_i} J_i (\tau - 0.5)^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

dGr_dtau [float] Derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *tau*, [-]

Examples

```
>>> iapws97_dGr_dtau_region2(540/300.0, 101325/1e6)
-18.1535856049444
```

chemicals.iapws.**iapws97_d2Gr_dtau2_region2**(*tau*, *pi*)

Calculates the second derivative of dimensionless residual Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial^2 \gamma^r}{\partial \tau^2} = \sum_{i=1}^{43} n_i \pi^{I_i} J_i (J_i - 1)(\tau - 0.5)^{J_i - 2}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2Gr_dtau2 [float] Second derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *tau*, [-]

Examples

```
>>> iapws97_d2Gr_dtau2_region2(540/300.0, 101325/1e6)
-470.9302933324787
```

chemicals.iapws.**iapws97_d2Gr_dpidtau_region2**(*tau*, *pi*)

Calculates the second derivative of dimensionless residual Gibbs free energy with respect to *tau* and *pi* for water according to the IAPWS-97 standard (for region 2).

$$\frac{\partial^2 \gamma^r}{\partial \tau \partial \pi} = \sum_{i=1}^{43} n_i I_i \pi^{I_i - 1} J_i (\tau - 0.5)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2Gr_dpidtau_ [float] Second derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *tau* and *pi*, [-]

Examples

```
>>> iapws97_d2Gr_dpidtau_region2(540/300.0, 101325/1e6)
-735.391845360247
```

1.13.9 IAPWS-97 Region 3

chemicals.iapws.**iapws97_A_region3**(*tau*, *delta*)

Calculates the dimensionless Helmholtz free energy for water according to the IAPWS-97 standard (for region 3).

$$\frac{f(\rho,T)}{RT} = \phi(\delta,\tau) = n_1 \ln\delta + \sum_{i=2}^{40} n_i \delta^{I_i} \tau^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

A [float] Helmholtz free energy A/(RT), [-]

Examples

```
>>> iapws97_A_region3(647.096/500.0, 400.0/322.0)
-3.0336402168865
```

chemicals.iapws.**iapws97_dA_ddelta_region3**(*tau*, *delta*)

Calculates the derivative of dimensionless Helmholtz free energy with respect to *delta* for water according to the IAPWS-97 standard (for region 3).

$$\frac{\partial\phi(\delta,\tau)}{\partial\delta} = \frac{n_1}{\delta} + \sum_{i=2}^{40} n_i I_i \delta^{I_i-1} \tau^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

dA_ddelta [float] Derivative of dimensionless Helmholtz free energy with respect to *delta*, [-]

Examples

```
>>> iapws97_dA_ddelta_region3(647.096/500.0, 400.0/322.0)
7.35562435092
```

chemicals.iapws.**iapws97_d2A_ddelta2_region3**(*tau*, *delta*)

Calculates the second derivative of dimensionless Helmholtz free energy with respect to *delta* for water according to the IAPWS-97 standard (for region 3).

$$\frac{\partial^2\phi(\delta,\tau)}{\partial\delta^2} = \frac{-n_1}{\delta^2} + \sum_{i=2}^{40} n_i I_i (I_i - 1) \delta^{I_i-2} \tau^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2A_ddelta2 [float] Second derivative of dimensionless Helmholtz free energy with respect to *delta*, [-]

Examples

```
>>> iapws97_d2A_ddelta2_region3(647.096/500.0, 400.0/322.0)
-2.2858869882497
```

chemicals.iapws.**iapws97_dA_dtau_region3**(*tau*, *delta*)

Calculates the derivative of dimensionless Helmholtz free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 3).

$$\frac{\partial \phi(\delta, \tau)}{\partial \tau} = + \sum_{i=2}^{40} n_i J_i \delta^{I_i} \tau^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

dA_dtau [float] Derivative of dimensionless Helmholtz free energy with respect to *tau*, [-]

Examples

```
>>> iapws97_dA_dtau_region3(647.096/500.0, 400.0/322.0)
-24.9687028688
```

chemicals.iapws.**iapws97_d2A_dtau2_region3**(*tau*, *delta*)

Calculates the second derivative of dimensionless Helmholtz free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 3).

$$\frac{\partial^2 \phi(\delta, \tau)}{\partial \tau^2} = + \sum_{i=2}^{40} n_i J_i (J_i - 1) \delta^{I_i} \tau^{J_i - 2}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2A_dtau2 [float] Second derivative of dimensionless Helmholtz free energy with respect to *tau*, [-]

Examples

```
>>> iapws97_d2A_dtau2_region3(647.096/500.0, 400.0/322.0)
-373.6565823701
```

chemicals.iapws.**iapws97_d2A_ddeltadtau_region3**(*tau*, *delta*)

Calculates the second derivative of dimensionless Helmholtz free energy with respect to *tau* and *delta* for water according to the IAPWS-97 standard (for region 3).

$$\frac{\partial^2 \phi(\delta, \tau)}{\partial \tau \partial \delta} = + \sum_{i=2}^{40} n_i J_i \delta^{I_i - 1} \tau^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2A_ddeltadtau [float] Second derivative of dimensionless Helmholtz free energy with respect to *tau* and *delta*, [-]

Examples

```
>>> iapws97_d2A_ddeltadtau_region3(647.096/500.0, 400.0/322.0)
145.85190014717
```

1.13.10 IAPWS-97 Region 3 PT Backwards Equation Boundaries

chemicals.iapws.**iapws97_boundary_3uv**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3uv(22.3E6)
647.7996121480069
```

chemicals.iapws.**iapws97_boundary_3ef**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3ef(40E6)
713.959399239744
```

chemicals.iapws.**iapws97_boundary_3cd**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3cd(25E6)
649.3659208321279
```

chemicals.iapws.**iapws97_boundary_3gh**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3gh(25E6)
656.69805722612
```

chemicals.iapws.**iapws97_boundary_3ij**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3ij(25E6)
660.7865756716819
```

chemicals.iapws.**iapws97_boundary_3jk**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3jk(25E6)
668.1915358826951
```

chemicals.iapws.**iapws97_boundary_3mn**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3mn(22.8E6)
649.6054132953997
```

chemicals.iapws.**iapws97_boundary_3qu**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3qu(22E6)
645.6355027340121
```

chemicals.iapws.**iapws97_boundary_3rx**(*P*)

Calculates the transition temperature for a region 3 PT backwards equation transition.

Parameters

P [float] Pressure [Pa]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3rx(22E6)
648.26227536701
```

chemicals.iapws.**iapws97_boundary_3wx**(*logP_MPa*, *logP_MPa_inv*)

Calculates the transition temperature for a region 3 PT backwards equation transition (for one of "wx", "ab", or "op"; the others do not use a log fit). The parameters are provided in the specific units for speed savings only.

Parameters

logP_MPa [float] Natural logarithm of pressure in units of MPa [log(MPa)]

logP_MPa_inv [float] Inverse of Natural logarithm of pressure in units of MPa [1/log(MPa)]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3wx(log(22.3), 1/log(22.3))
648.204947950734
```

chemicals.iapws.**iapws97_boundary_3ab**(*logP_MPa*, *logP_MPa_inv*)

Calculates the transition temperature for a region 3 PT backwards equation transition (for one of "wx", "ab", or "op"; the others do not use a log fit). The parameters are provided in the specific units for speed savings only.

Parameters

logP_MPa [float] Natural logarithm of pressure in units of MPa [log(MPa)]

logP_MPa_inv [float] Inverse of Natural logarithm of pressure in units of MPa [1/log(MPa)]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3ab(log(40), 1/log(40))
693.0341408296053
```

chemicals.iapws.**iapws97_boundary_3op**(*logP_MPa*, *logP_MPa_inv*)

Calculates the transition temperature for a region 3 PT backwards equation transition (for one of "wx", "ab", or "op"; the others do not use a log fit). The parameters are provided in the specific units for speed savings only.

Parameters

logP_MPa [float] Natural logarithm of pressure in units of MPa [log(MPa)]

logP_MPa_inv [float] Inverse of Natural logarithm of pressure in units of MPa [1/log(MPa)]

Returns

T_trans [float] Transition temperature [K]

Examples

```
>>> iapws97_boundary_3op(log(22.8), 1/log(22.8))
650.010694314133
```

1.13.11 IAPWS-97 Region 3 PT Backwards Equations

chemicals.iapws.**iapws97_region3_a**(*T*, *P*)

Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_b**(*T*, *P*)

Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_c**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_d**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_e**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_f**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_g**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_h**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_i**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_j**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_k**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_l**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_m**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_n**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_o**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_p**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_q**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_r**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_s**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_t**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_u**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_v**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_w**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_x**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_y**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

chemicals.iapws.**iapws97_region3_z**(*T*, *P*)
Calculate the mass density water in one of the 26 region 3 backwards regions of the IAPWS-97 standard.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

rho [float] Mass density of water in region 3, [kg/m^3]

Notes

Significant discontinuities exist between each region. These functions are automatically generated and are not to be edited directly.

1.13.12 IAPWS-97 Region 5

chemicals.iapws.**iapws97_G0_region5**(*tau*, *pi*)

Calculates the dimensionless ideal gas Gibbs free energy for water according to the IAPWS-97 standard (for region 5).

$$\gamma^\circ = \ln \pi + \sum_{i=1}^{6} n_i^\circ \tau^{J_i^\circ}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

G0 [float] Dimensionless ideal gas Gibbs energy G/(RT), [-]

Examples

```
>>> iapws97_G0_region5(1000.0/1500, 101325/1e6)
-14.9741430290056
```

chemicals.iapws.**iapws97_dG0_dtau_region5**(*tau*, *pi*)

Calculates the first derivative of dimensionless ideal gas Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial \gamma^\circ}{\partial \tau} = \sum_{i=1}^{6} n_i^\circ J_i^\circ \tau^{J_i^\circ - 1}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

dG0_dtau [float] First derivative of dimensionless ideal gas Gibbs energy G/(RT) with respect to *tau*, [-]

Notes

This function does not depend on *pi* but it is accepted for consistency.

Examples

```
>>> iapws97_dG0_dtau_region5(1000.0/1500, 101325/1e6)
11.311766995978
```

chemicals.iapws.**iapws97_d2G0_dtau2_region5**(*tau*, *pi*)

Calculates the second derivative of dimensionless ideal gas Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial^2 \gamma^\circ}{\partial \tau^2} = \sum_{i=1}^{6} n_i^\circ J_i^\circ (J_i^\circ - 1) \tau^{J_i^\circ - 2}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2G0_dtau2 [float] Second derivative of dimensionless ideal gas Gibbs energy G/(RT) with respect to *tau*, [-]

Notes

This function does not depend on *pi* but it is accepted for consistency.

Examples

```
>>> iapws97_d2G0_dtau2_region5(1000.0/1500, 101325/1e6)
-12.744650271463655
```

chemicals.iapws.**iapws97_Gr_region5**(*tau*, *pi*)

Calculates the dimensionless residual Gibbs free energy for water according to the IAPWS-97 standard (for region 5).

$$\gamma^r = \sum_{i=1}^{6} n_i \pi^{I_i} (\tau)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

Gr [float] Dimensionless residual Gibbs energy Gr/(RT), [-]

Examples

```
>>> iapws97_Gr_region5(1000/300.0, 101325/1e6)
-0.0194648291645718
```

chemicals.iapws.**iapws97_dGr_dpi_region5**(*tau*, *pi*)

Calculates the first derivative of dimensionless residual Gibbs free energy with respect to *pi* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial \gamma^r}{\partial \pi} = \sum_{i=1}^{6} n_i I_i \pi^{I_i - 1} (\tau)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

dGr_dpi [float] Derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *pi*, [-]

Notes

Used in density solution.

Examples

```
>>> iapws97_dGr_dpi_region5(1000/300.0, 101325/1e6)
-0.213281155629998
```

chemicals.iapws.**iapws97_d2Gr_dpi2_region5**(*tau*, *pi*)

Calculates the second derivative of dimensionless residual Gibbs free energy with respect to *pi* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial^2 \gamma^r}{\partial \pi^2} = \sum_{i=1}^{6} n_i I_i (I_i - 1) \pi^{I_i - 2} (\tau)^{J_i}$$

Parameters

tau [float] Dimensionless temperature, (540 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2Gr_dpi2 [float] Second derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *pi*, [-]

Examples

```
>>> iapws97_d2Gr_dpi2_region5(1000/300.0, 101325/1e6)
-0.4179905782304291
```

chemicals.iapws.**iapws97_dGr_dtau_region5**(*tau*, *pi*)

Calculates the first derivative of dimensionless residual Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial \gamma^r}{\partial \tau} = \sum_{i=1}^{6} n_i \pi^{I_i} J_i (\tau)^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

dGr_dtau [float] Derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *tau*, [-]

Examples

```
>>> iapws97_dGr_dtau_region5(1000/300.0, 101325/1e6)
-0.02200629869194
```

chemicals.iapws.**iapws97_d2Gr_dtau2_region5**(*tau*, *pi*)

Calculates the second derivative of dimensionless residual Gibbs free energy with respect to *tau* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial^2 \gamma^r}{\partial \tau^2} = \sum_{i=1}^{6} n_i \pi^{I_i} J_i (J_i - 1) (\tau)^{J_i - 2}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2Gr_dtau2 [float] Second derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *tau*, [-]

Examples

```
>>> iapws97_d2Gr_dtau2_region5(1000/300.0, 101325/1e6)
-0.0239165867999155
```

chemicals.iapws.**iapws97_d2Gr_dpidtau_region5**(*tau*, *pi*)

Calculates the second derivative of dimensionless residual Gibbs free energy with respect to *tau* and *pi* for water according to the IAPWS-97 standard (for region 5).

$$\frac{\partial^2 \gamma^r}{\partial \tau \partial \pi} = \sum_{i=1}^{6} n_i I_i \pi^{I_i - 1} J_i (\tau)^{J_i - 1}$$

Parameters

tau [float] Dimensionless temperature, (1000 K)/T [-]

pi [float] Dimensionless pressure, P/(1 MPa), [-]

Returns

d2Gr_dpidtau [float] Second derivative of dimensionless residual Gibbs energy Gr/(RT) with respect to *tau* and *pi*, [-]

Examples

```
>>> iapws97_d2Gr_dpidtau_region5(1000/300.0, 101325/1e6)
-0.27438379131103097
```

1.13.13 IAPWS-95 Ideal Gas Terms

chemicals.iapws.**iapws95_A0**(*tau*, *delta*)

Calculates the ideal gas Helmholtz energy of water according to the IAPWS-95 standard.

$$\phi^\circ = \ln\delta + n_1 + n_2\tau + n_3\ln\tau + \sum_{i=4}^{8} n_i \ln\left[1 - \exp(-\gamma_i\tau)\right]$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

A0 [float] Ideal gas dimensionless Helmholtz energy A/(RT) [-]

Notes

This implementation is checked to have a relative error always under 1e-15.

Examples

```
>>> iapws95_A0(647.096/300.0, 999.0/322)
9.537075529761053
```

chemicals.iapws.**iapws95_dA0_dtau**(*tau*, *delta*)

Calculates the first derivative of ideal gas Helmholtz energy of water with respect to *tau* according to the IAPWS-95 standard.

$$\frac{\partial\phi^\circ}{\partial\tau} = n_2 + \frac{n_3}{\tau} + \sum_{i=4}^{8} n_i\gamma_i\left[\left(1 - \exp(-\gamma_i\tau)\right)^{-1} - 1\right]$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

dA0_dtau [float] First derivative of ideal gas dimensionless Helmholtz energy A/(RT) with respect to *tau* [-]

Notes

This implementation is checked to have a relative error always under 1e-15.

Examples

```
>>> iapws95_dA0_dtau(647.096/300.0, 999.0/322)
8.079705548882
```

chemicals.iapws.**iapws95_d2A0_dtau2**(*tau*, *delta*)

Calculates the second derivative of ideal gas Helmholtz energy of water with respect to *tau* according to the IAPWS-95 standard.

$$\frac{\partial^2 \phi^\circ}{\partial \tau^2} = \frac{n_3}{\tau^2} + \sum_{i=4}^{8} n_i \gamma_i^2 \exp(-\gamma_i \tau) \left[(1 - \exp(-\gamma_i \tau))^{-2}\right]$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2A0_dtau2 [float] Second derivative of ideal gas dimensionless Helmholtz energy A/(RT) with respect to *tau* [-]

Notes

This implementation is checked to have a relative error always under 1e-15.

Examples

```
>>> iapws95_d2A0_dtau2(647.096/300.0, 999.0/322)
-0.653543047751809
```

chemicals.iapws.**iapws95_d3A0_dtau3**(*tau*, *delta*)

Calculates the third derivative of ideal gas Helmholtz energy of water with respect to *tau* according to the IAPWS-95 standard.

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d3A0_dtau3 [float] Third derivative of ideal gas dimensionless Helmholtz energy A/(RT) with respect to *tau* [-]

Notes

This implementation is checked to have a relative error always under 1e-15. This equation is not explicitly in IAPWS-95, but is needed to compute some second derivatives.

Examples

```
>>> iapws95_d3A0_dtau3(647.096/300.0, 999.0/322)
0.6222542507278
```

chemicals.iapws.**iapws95_A0_tau_derivatives**(*tau*, *delta*)

Calculates the ideal gas Helmholtz energy of water and its first three derivatives with respect to *tau* according to the IAPWS-95 standard. As each of those calls spends most of their time computing exponentials which are the same for each function, function offers a time saving.

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

A0 [float] Ideal gas dimensionless Helmholtz energy A/(RT) [-]

dA0_dtau [float] First derivative of ideal gas dimensionless Helmholtz energy A/(RT) with respect to *tau* [-]

d2A0_dtau2 [float] Second derivative of ideal gas dimensionless Helmholtz energy A/(RT) with respect to *tau* [-]

d3A0_dtau3 [float] Third derivative of ideal gas dimensionless Helmholtz energy A/(RT) with respect to *tau* [-]

Notes

The extra cost of calling this function vs *iapws95_A0* alone is ~15% with numba, ~40% with PyPy, and 120% with CPython.

Examples

```
>>> iapws95_A0_tau_derivatives(647.096/300.0, 999.0/322)
(9.53707552976, 8.0797055488, -0.65354304775, 0.62225425072)
```

1.13.14 IAPWS-95 Residual Terms

chemicals.iapws.**iapws95_Ar**(*tau*, *delta*)

Calculates the residual Helmholtz energy of water according to the IAPWS-95 standard.

$$\phi^{\mathrm{r}} = \sum_{i=1}^{7} n_i \delta^{d_i} \tau^{t_i} + \sum_{i=8}^{51} n_i \delta^{d_i} \tau^{t_i} \mathrm{e}^{-\delta^{c_i}} + \sum_{i=52}^{54} n_i \delta^{d_i} \tau^{t_i} \mathrm{e}^{-\alpha_i(\delta-\varepsilon_i)^2-\beta_i(\tau-\gamma_i)^2} + \sum_{i=55}^{56} n_i \Delta^{b_i} \delta\psi$$

$$\Delta = \theta^2 + B_i \left[(\delta-1)^2\right]^{a_i}$$

$$\theta = (1 - \tau) + A_i \left[(\delta - 1)^2\right]^{\frac{1}{2\beta_i}}$$

$$\psi = e^{-C_i(\delta-1)^2 - D_i(\tau-1)^2}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

Ar [float] Residual Helmholtz energy A/(RT) [-]

Notes

This is an optimized implementatation taking 9 exp calls, 4 sqrts, and 3 powers. It was generated using SymPy's CSE functionality, with select polynomial optimizations by hand as well. It is over 10x faster than a naive implementation.

This implementation has been tested against a straightforward implementation with the equations given in IAPWS-95.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 4E6 points were evaluated. The mean relative error was 5.0416E-15, with a maximum relative error of 1.118E-9 and a standard deviation of 5.773e-13.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 90000 points, the mean relative error was 3.14E-15, with a maximum relative error of 3.54e-12 and a standard deviation of 3.017E-14.

This comparison indicates that this implementation is more accurate than the straightforward implementation.

Examples

```
>>> iapws95_Ar(647.096/300.0, 999.0/322)
-9.57577716026768
```

chemicals.iapws.**iapws95_dAr_ddelta**(*tau*, *delta*)

Calculates the first derivative of residual Helmholtz energy of water with respect to *delta* according to the IAPWS-95 standard.

$$\phi_\delta^r = \sum_{i=1}^{7} n_i d_i \delta^{d_i - 1} \tau^{t_i} + \sum_{i=8}^{51} n_i \mathrm{e}^{-\delta^{c_i}} \left[\delta^{d_i - 1} \tau^{t_i} \left(d_i - c_i \delta^{c_i}\right)\right] + \sum_{i=52}^{54} n_i \delta^{d_i} \tau^{t_i} \mathrm{e}^{-\alpha_i(\delta-\varepsilon_i)^2 - \beta_i(\tau-\gamma_i)^2} \left[\frac{d_i}{\delta} - 2\alpha_i\left(\delta - \varepsilon_i\right)\right] + \sum_{i=55}^{56}$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

dAr_ddelta [float] First derivative of residual Helmholtz energy A/(RT) with respect to *delta*, [-]

Notes

This is an optimized implementatation taking 8 exp calls, 4 sqrts, and 2 powers. It was generated using SymPy's CSE functionality, with select polynomial optimizations by hand as well. It is over 10x faster than a naive implementation.

This implementation has been tested against a straightforward implementation with the equations given in IAPWS-95.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 4E6 points were evaluated. The mean relative error was 4.033E-15, with a maximum relative error of 3.8765e-10 and a standard deviation of 3.189e-13.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 90000 points, the mean relative error was 6.046E-15, with a maximum relative error of 3.39E-10 and a standard deviation of 7.056E-13.

There was a singularity at *tau* = *delta* = 1, but the limit is correctly returned.

Examples

```
>>> iapws95_dAr_ddelta(647.096/300.0, 999.0/322)
-0.3093321202374
```

chemicals.iapws.**iapws95_d2Ar_ddelta2**(*tau*, *delta*)

Calculates the second derivative of residual Helmholtz energy of water with respect to *delta* according to the IAPWS-95 standard.

$$\phi^{\mathrm{r}}_{\delta\delta} = \sum_{i=1}^{7} n_i d_i \left(d_i - 1\right) \delta^{d_i - 2} \tau^{t_i} + \sum_{i=8}^{51} n_i \mathrm{e}^{-\delta^6} \left[\delta^{d_i - 2} \tau^{t_i} \left(\left(d_i - c_i \delta^{c_i}\right)\left(d_i - 1 - c_i \delta^{c_i}\right) - c_i^2 \delta^{c_i}\right)\right] + \sum_{i=52}^{54} n_i \tau^{t_i} \mathrm{e}^{-\alpha_i (\delta - \varepsilon_i)^2 - \beta_i (\tau -$$

$$\cdot \left[-2\alpha_i \delta^{d_i} + 4\alpha_i^2 \delta^{d_i} \left(\delta - \varepsilon_i\right)^2 - 4 d_i \alpha_i \delta^{d_i - 1} \left(\delta - \varepsilon_i\right) + d_i \left(d_i - 1\right) \delta^{d_i - 2}\right] + \sum_{i=55}^{56} n_i \left[\Delta^{b_i} \left(2\frac{\partial \psi}{\partial \delta} + \delta \frac{\partial^2 \psi}{\partial \delta^2}\right) + 2\frac{\partial \Delta^{b_i}}{\partial \delta} \left($$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2Ar_ddelta2 [float] Second derivative of residual Helmholtz energy A/(RT) with respect to *delta*, [-]

Notes

This is an optimized implementatation taking 4 exp calls, 4 sqrts, and 2 powers. It was generated using SymPy's CSE functionality, with select polynomial optimizations by hand as well. It is over 10x faster than a naive implementation.

This implementation has been tested against a straightforward implementation with the equations given in IAPWS-95.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 4E6 points were evaluated. The mean relative error was 9.566e-16, with a maximum relative error of 1.0518E-10 and a standard deviation of 6.20265E-14.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 250000 points, the mean relative error was 1.039E-15, with a maximum relative error of 2.431E-11 and a standard deviation of 5.31708E-14.

Examples

```
>>> iapws95_d2Ar_ddelta2(647.096/300.0, 999.0/322)
1.7862535141735987
```

chemicals.iapws.**iapws95_d3Ar_ddelta3**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of water with respect to *delta* according to the IAPWS-95 standard.

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d3Ar_ddelta3 [float] Third derivative of residual Helmholtz energy A/(RT) with respect to *delta*, [-]

Notes

No equation is given for this in IAPWS-95, and the derivative was symbolically computed with SymPy.

This is an optimized implementatation. It was generated using SymPy's CSE functionality.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-4 kg/m^3 to 5000 kg/m^3, 90000 points were evaluated. The mean relative error was 5.41E-13, with a maximum relative error of 6.3957e-11 and a standard deviation of 3.346e-12.

90000 points were also evaluated with mpmath. The mean relative error was 1.41959E-14, with a maximum relative error of 5.8878E-10 and a standard deviation of 1.978E-12.

Evaluating 10000 points in the 1e-10 to 1e-4 range, the mean relative error was 1.2E-16, maximum relative error 1.2e-16, and standard deviation 6.66e-16.

Examples

```
>>> iapws95_d3Ar_ddelta3(647.096/300.0, 999.0/322)
0.33621190578
```

chemicals.iapws.**iapws95_dAr_dtau**(*tau*, *delta*)

Calculates the first derivative of residual Helmholtz energy of water with respect to *tau* according to the IAPWS-95 standard.

$$\phi_\tau^{\mathrm{r}} = \sum_{i=1}^{7} n_i t_i \delta^{d_i} \tau^{t_i - 1} + \sum_{i=8}^{51} n_i t_i \delta^{d_i} \tau^{t_i - 1} \mathrm{e}^{-\delta^{c_i}} + \sum_{i=52}^{54} n_i \delta^{d_i} \tau^{t_i} \mathrm{e}^{-\alpha_i (\delta - \varepsilon_i)^2 - \beta_i (\tau - \gamma_i)^2} \left[\frac{t_i}{\tau} - 2\beta_i (\tau - \gamma_i)\right] + \sum_{i=55}^{56} n_i \delta \left[\frac{\partial \Delta^{b_i}}{\partial \tau} \psi -$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

dAr_dtau [float] Derivative of residual Helmholtz energy A/(RT) with respect to *tau*, [-]

Notes

This is an optimized implementatation taking 9 exp calls, 4 sqrts, and 2 powers. It was generated using SymPy's CSE functionality, with a limited amount of *horner* polynomial optimizations as well. It is over 10x faster than a naive implementation.

This implementation has been tested against a straightforward implementation with the equations given in IAPWS-95.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 250000 points were evaluated. The mean relative error was 5.68E-14, with a maximum relative error of 6.73E-9 and a standard deviation of 1.35E-11.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 90000 points, the mean relative error was 4.66E-14, with a maximum relative error of 4.25E-10 and a standard deviation of 1.77E-12.

The maximum error ocurs in the extremely low density regime, $\rho < 1e-6$.

Examples

```
>>> iapws95_dAr_dtau(647.096/300.0, 999.0/322)
-7.7043336309570
```

chemicals.iapws.**iapws95_d2Ar_dtau2**(*tau*, *delta*)

Calculates the second derivative of residual Helmholtz energy of water with respect to *tau* according to the IAPWS-95 standard.

$$\phi^{\mathrm{r}}_{\tau\tau} = \sum_{i=1}^{7} n_i t_i \left(t_i - 1\right) \delta^{d_i} \tau^{t_i - 2} + \sum_{i=8}^{51} n_i t_i \left(t_i - 1\right) \delta^{d_i} \tau^{t_i - 2} \mathrm{e}^{-\delta^{c_i}} + \sum_{i=52}^{54} n_i \delta^{d_i} \tau^{t_i} \mathrm{e}^{-\alpha_i (\delta - \varepsilon_i)^2 - \beta_i (\tau - \gamma_i)^2} \left[\left(\frac{t_i}{\tau} - 2\beta_i \left(\tau - \gamma_i\right)\right)^{2}\right.$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2Ar_dtau2 [float] Second derivative of residual Helmholtz energy A/(RT) with respect to *tau*, [-]

Notes

This is an optimized implementatation taking 9 exp calls, 4 sqrts, and 2 powers. It was generated using SymPy's CSE functionality, with a limited amount of *horner* polynomial optimizations as well. It is over 10x faster than a naive implementation.

This implementation has been tested against a straightforward implementation with the equations given in IAPWS-95.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 4E6 points were evaluated. The mean relative error was 4.595E-16, with a maximum relative error of 1.835e-10 and a standard deviation of 1.209E-13.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 250000 points, the mean relative error was 2.6026E-16, with a maximum relative error of 2.36E-12 and a standard deviation of 8.055E-15.

This comparison indicates this implementation is more accurate than the straightforward implementation.

Examples

```
>>> iapws95_d2Ar_dtau2(647.096/300.0, 999.0/322)
-1.2616419775539
```

chemicals.iapws.**iapws95_d2Ar_ddeltadtau**(*tau*, *delta*)

Calculates the second derivative of residual Helmholtz energy of water with respect to *tau* and also *delta* according to the IAPWS-95 standard.

$$\phi^{\mathrm{r}}_{\delta\tau} = \sum_{i=1}^{7} n_i d_i t_i \delta^{d_i-1}\tau^{t_i-1} + \sum_{i=8}^{51} n_i t_i \delta^{d_i-1}\tau^{t_i-1}\left(d_i - c_i\delta^{c_i}\right)\mathrm{e}^{-\delta^{c_i}} + \sum_{i=52}^{54} n_i \delta^{d_i}\tau^{t_i}\mathrm{e}^{-\alpha_i(\delta-\varepsilon_i)^2-\beta_i(\tau-\gamma_i)^2}\left[\frac{d_i}{\delta} - 2\alpha_i\left(\delta-\varepsilon_i\right)\right]$$

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d2Ar_ddeltadtau [float] Second derivative of residual Helmholtz energy A/(RT) with respect to *tau* and *delta*, [-]

Notes

This is an optimized implementatation taking 11 exp calls, 4 sqrts, and 3 powers. It was generated using SymPy's CSE functionality, with select polynomial optimizations by hand as well. It is over 10x faster than a naive implementation.

This implementation has been tested against a straightforward implementation with the equations given in IAPWS-95.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 4E6 points were evaluated. The mean relative error was 2.82E-14, with a maximum relative error of 8.404E-9 and a standard deviation of 5.166e-12.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 90000 points, the mean relative error was 6.974E-14, with a maximum relative error of 4.286E-9 and a standard deviation of 4.286E-12.

Examples

```
>>> iapws95_d2Ar_ddeltadtau(647.096/300.0, 999.0/322)
-0.198403562385
```

chemicals.iapws.**iapws95_d3Ar_ddeltadtau2**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of water with respect to *delta* once and *tau* twice according to the IAPWS-95 standard.

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d3Ar_ddeltadtau2 [float] Third derivative of residual Helmholtz energy A/(RT) with respect to *delta* once and *tau* twice, [-]

Notes

This is an optimized implementatation. It was generated using SymPy's CSE functionality.

No equation is given for this in IAPWS-95, and the derivative was symbolically computed with SymPy.

Like many higher-order derivatives of functions with exponentials, this one balloons to use many, many terms.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 250000 points were evaluated. The mean relative error was 7.936e-16, with a maximum relative error of 1.965E-11 and a standard deviation of 4.7938E-14.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 90000 points, the mean relative error was 6.08E-16, with a maximum relative error of 3.537E-12 and a standard deviation of 1.85197E-14.

Examples

```
>>> iapws95_d3Ar_ddeltadtau2(647.096/300.0, 999.0/322)
1.081479970332
```

chemicals.iapws.**iapws95_d3Ar_ddelta2dtau**(*tau*, *delta*)

Calculates the third derivative of residual Helmholtz energy of water with respect to *delta* twice and *tau* one according to the IAPWS-95 standard.

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d3Ar_ddeltadtau2 [float] Third derivative of residual Helmholtz energy A/(RT) with respect to *delta* twice and *tau* once, [-]

Notes

This is an optimized implementatation. It was generated using SymPy's CSE functionality.

No equation is given for this in IAPWS-95, and the derivative was symbolically computed with SymPy.

Like many higher-order derivatives of functions with exponentials, this one balloons to use many, many terms.

Over a linear temperature range of 200 K to 5000 K and a logarithmic density range of 1E-10 kg/m^3 to 5000 kg/m^3, 250000 points were evaluated. The mean relative error was 3.629e-15, with a maximum relative error of 8.38E-11 and a standard deviation of 2.1214E-13.

Over the same range, the model was evaluated to a precision of 50 decimal places with *mpmath*, and on 10000 points, the mean relative error was 2.4e-15, with a maximum relative error of 7.62E-12 and a standard deviation of 7.818E-14.

Examples

```
>>> iapws95_d3Ar_ddelta2dtau(647.096/300.0, 999.0/322)
0.015646982949077
```

chemicals.iapws.**iapws95_d4Ar_ddelta2dtau2**(*tau*, *delta*)

Calculates the fourth derivative of residual Helmholtz energy of water with respect to *tau* twice and *delta* twice according to the IAPWS-95 standard.

Parameters

tau [float] Dimensionless temperature, (647.096 K)/T [-]

delta [float] Dimensionless density, rho/(322 kg/m^3), [-]

Returns

d4Ar_ddelta2dtau2 [float] Fourth derivative of residual Helmholtz energy A/(RT) with respect to *tau* and *delta*, [-]

Examples

```
>>> iapws95_d4Ar_ddelta2dtau2(647.096/300.0, 999.0/322)
-2.656422915480
```

1.14 Chemical Metadata (chemicals.identifiers)

This module contains a database of metadata on ~70000 chemicals from the PubChem datase. It contains comprehensive feature for searching the metadata. It also includes a small database of common mixture compositions.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.14.1 Search Functions

chemicals.identifiers.**CAS_from_any**(*ID*, *autoload=False*, *cache=True*)

Wrapper around *search_chemical* which returns the CAS number of the found chemical directly.

Parameters

ID [str] One of the name formats described by *search_chemical*, [-]

autoload [bool, optional] Whether to load new chemical databanks during the search if a hit is not immediately found, [-]

cache [bool, optional] Whether or not to cache the search for faster lookup in subsequent queries, [-]

Returns

CASRN [str] A three-piece, dash-separated set of numbers

Notes

An exception is raised if the name cannot be identified. The PubChem database includes a wide variety of other synonyms, but these may not be present for all chemcials. See *search_chemical* for more details.

Examples

```
>>> CAS_from_any('water')
'7732-18-5'
>>> CAS_from_any('InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3')
'64-17-5'
>>> CAS_from_any('CCCCCCCCCC')
'124-18-5'
>>> CAS_from_any('InChIKey=LFQSCWFLJHTTHZ-UHFFFAOYSA-N')
'64-17-5'
>>> CAS_from_any('pubchem=702')
'64-17-5'
>>> CAS_from_any('O') # only elements can be specified by symbol
'17778-80-2'
```

chemicals.identifiers.**MW**(*ID*, *autoload=False*, *cache=True*)

Wrapper around *search_chemical* which returns the molecular weight of the found chemical directly.

Parameters

ID [str] One of the name formats described by *search_chemical*

Returns

MW [float] Molecular weight of chemical, [g/mol]

Notes

An exception is raised if the name cannot be identified. The PubChem database includes a wide variety of other synonyms, but these may not be present for all chemcials. See *search_chemical* for more details.

Examples

```
>>> MW('water')
18.01528
>>> MW('InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3')
46.06844
>>> MW('CCCCCCCCCC')
142.286
>>> MW('InChIKey=LFQSCWFLJHTTHZ-UHFFFAOYSA-N')
46.06844
>>> MW('pubchem=702')
46.06844
```

(continues on next page)

(continued from previous page)

```
>>> MW('O') # only elements can be specified by symbol
15.9994
```

chemicals.identifiers.**search_chemical**(*ID*, *autoload=False*, *cache=True*)

Looks up metadata about a chemical by searching and testing for the input string being any of the following types of chemical identifiers:

- Name, in IUPAC form or common form or a synonym registered in PubChem
- InChI name, prefixed by 'InChI=1S/' or 'InChI=1/'
- InChI key, prefixed by 'InChIKey='
- PubChem CID, prefixed by 'PubChem='
- SMILES (prefix with 'SMILES=' to ensure smiles parsing; ex. 'C' will return Carbon as it is an element whereas the SMILES interpretation for 'C' is methane)
- CAS number (obsolete numbers may point to the current number)

If the input is an ID representing an element, the following additional inputs may be specified as

- Atomic symbol (ex 'Na')
- Atomic number (as a string)

Parameters

ID [str] One of the name formats described above

autoload [bool, optional] Whether to load new chemical databanks during the search if a hit is not immediately found, [-]

cache [bool, optional] Whether or not to cache the search for faster lookup in subsequent queries, [-]

Returns

chemical_metadata [ChemicalMetadata] A class containing attributes which describe the chemical's metadata, [-]

Notes

An exception is raised if the name cannot be identified. The PubChem database includes a wide variety of other synonyms, but these may not be present for all chemcials.

Examples

```
>>> search_chemical('water')
<ChemicalMetadata, name=water, formula=H2O, smiles=O, MW=18.0153>
>>> search_chemical('InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3')
<ChemicalMetadata, name=ethanol, formula=C2H6O, smiles=CCO, MW=46.0684>
>>> search_chemical('CCCCCCCCCC')
<ChemicalMetadata, name=DECANE, formula=C10H22, smiles=CCCCCCCCCC, MW=142.286>
>>> search_chemical('InChIKey=LFQSCWFLJHTTHZ-UHFFFAOYSA-N')
<ChemicalMetadata, name=ethanol, formula=C2H6O, smiles=CCO, MW=46.0684>
>>> search_chemical('pubchem=702')
```

(continues on next page)

(continued from previous page)

```
<ChemicalMetadata, name=ethanol, formula=C2H6O, smiles=CCO, MW=46.0684>
>>> search_chemical('O') # only elements can be specified by symbol
<ChemicalMetadata, name=oxygen, formula=O, smiles=[O], MW=15.9994>
```

chemicals.identifiers.**IDs_to_CASs**(*IDs*)

Find the CAS numbers for multiple chemicals names at once. Also supports having a string input which is a common mixture name in the database. An error will be raised if any of the chemicals cannot be found.

Parameters

IDs [list[str] or str] A string or 1-element list containing the name which may represent a mixture.

Returns

CASs [list[str]] CAS numbers of found chemicals, [-]

Notes

White space, '-', and upper case letters are removed in the search.

Examples

```
>>> IDs_to_CASs('R512A')
['811-97-2', '75-37-6']
>>> IDs_to_CASs(['norflurane', '1,1-difluoroethane'])
['811-97-2', '75-37-6']
```

1.14.2 CAS Number Utilities

chemicals.identifiers.**check_CAS**(*CASRN*)

Checks if a CAS number is valid. Returns False if the parser cannot parse the given string.

Parameters

CASRN [str] A three-piece, dash-separated set of numbers

Returns

result [bool] Boolean value if CASRN was valid. If parsing fails, return False also.

Notes

Check method is according to Chemical Abstract Society. However, no lookup to their service is performed; therefore, this function cannot detect false positives.

Function also does not support additional separators, apart from '-'.

CAS numbers up to the series 1 XXX XXX-XX-X are now being issued.

A long can hold CAS numbers up to 2 147 483-64-7

Examples

```
>>> check_CAS('7732-18-5')
True
>>> check_CAS('77332-18-5')
False
```

chemicals.identifiers.**CAS_to_int**(*i*)

Converts CAS number of a compounds from a string to an int. This is helpful when storing large amounts of CAS numbers, as their strings take up more memory than their numerical representational. All CAS numbers fit into 64 bit ints.

Parameters

CASRN [str] CASRN [-]

Returns

CASRN [int] CASRN [-]

Notes

Accomplishes conversion by removing dashes only, and then converting to an int. An incorrect CAS number will change without exception.

Examples

```
>>> CAS_to_int('7704-34-9')
7704349
```

chemicals.identifiers.**int_to_CAS**(*i*)

Converts CAS number of a compounds from an int to an string. This is helpful when dealing with int CAS numbers.

Parameters

CASRN [int] CASRN [-]

Returns

CASRN [str] CASRN [-]

Notes

Handles CAS numbers with an unspecified number of digits. Does not work on floats.

Examples

```
>>> int_to_CAS(7704349)
'7704-34-9'
```

chemicals.identifiers.**sorted_CAS_key**(*CASs*)

Takes a list of CAS numbers as strings, and returns a tuple of the same CAS numbers, sorted from smallest to largest. This is very convenient for obtaining a unique hash of a set of compounds, so as to see if two groups of compounds are the same.

Parameters

CASs [list[str]] CAS numbers as strings [-]

Returns

CASs_sorted [tuple[str]] Sorted CAS numbers from lowest (first) to highest (last) [-]

Notes

Does not check CAS numbers for validity.

Examples

```
>>> sorted_CAS_key(['7732-18-5', '64-17-5', '108-88-3', '98-00-0'])
('64-17-5', '98-00-0', '108-88-3', '7732-18-5')
```

1.14.3 Database Objects

There is an object used to represent a chemical's metadata, an object used to represent a common mixture's composition, and an object used to hold the mixture metadata.

class chemicals.identifiers.**ChemicalMetadata**(*pubchemid*, *CAS*, *formula*, *MW*, *smiles*, *InChI*, *InChI_key*, *iupac_name*, *common_name*, *synonyms*)

Class for storing metadata on chemicals.

Attributes

pubchemid [int] Identification number on pubchem database; access their information online at https://pubchem.ncbi.nlm.nih.gov/compound/<pubchemid> [-]

formula [str] Formula of the compound; in the same format as *chemicals.elements.serialize_formula* generates, [-]

MW [float] Molecular weight of the compound as calculated with the standard atomic abundances; consistent with the element weights in *chemicals.elements.periodic_table*, [g/mol]

smiles [str] SMILES identification string, [-]

InChI [str] InChI identification string as given in pubchem (there can be multiple valid InChI strings for a compound), [-]

InChI_key [str] InChI key identification string (meant to be unique to a compound), [-]

iupac_name [str] IUPAC name as given in pubchem, [-]

common_name [str] Common name as given in pubchem, [-]

synonyms [list[str]] List of synonyms of the compound, [-]

CAS [int] CAS number of the compound; stored as an int for memory efficiency, [-]

class chemicals.identifiers.**CommonMixtureMetadata**(*name*, *CASs*, *N*, *source*, *names*, *ws*, *zs*, *synonyms*)

Class for storing metadata on predefined chemical mixtures.

Attributes

name [str] Name of the mixture, [-]

source [str] Source of the mixture composition, [-]

N [int] Number of chemicals in the mixture, [-]

CASs [list[str]] CAS numbers of the mixture, [-]

ws [list[float]] Mass fractions of chemicals in the mixture, [-]

zs [list[float]] Mole fractions of chemicals in the mixture, [-]

names [list[str]] List of names of the chemicals in the mixture, [-]

synonyms [list[str]] List of synonyms of the mixture which can also be used to look it up, [-]

class chemicals.identifiers.**ChemicalMetadataDB**(*elements=True*, *main_db='/home/docs/checkouts/readthedocs.org/user_builds/chemicals/e packages/chemicals-1.1.4- py3.7.egg/chemicals/Identifiers/chemical identifiers pubchem large.tsv'*, *user_dbs=['/home/docs/checkouts/readthedocs.org/user_builds/chemicals packages/chemicals-1.1.4- py3.7.egg/chemicals/Identifiers/chemical identifiers pubchem small.tsv'*, *'/home/docs/checkouts/readthedocs.org/user_builds/chemicals/envs/stable packages/chemicals-1.1.4- py3.7.egg/chemicals/Identifiers/chemical identifiers example user db.tsv'*, *'/home/docs/checkouts/readthedocs.org/user_builds/chemicals/envs/stable packages/chemicals-1.1.4- py3.7.egg/chemicals/Identifiers/Cation db.tsv'*, *'/home/docs/checkouts/readthedocs.org/user_builds/chemicals/envs/stable packages/chemicals-1.1.4- py3.7.egg/chemicals/Identifiers/Anion db.tsv'*, *'/home/docs/checkouts/readthedocs.org/user_builds/chemicals/envs/stable packages/chemicals-1.1.4- py3.7.egg/chemicals/Identifiers/Inorganic db.tsv']*)

Object which holds the main database of chemical metadata.

Warning: To allow the *chemicals* to grow and improve, the details of this class may change in the future without notice!

Attributes

finished_loading Whether or not the database has loaded the main database.

Methods

autoload_main_db()	Load the main database when needed.
finish_loading()	Complete loading the main database, if it has not been fully loaded.
load(file_name)	Load a particular file into the indexes.
load_elements()	Load elements into the indexes.
search_CAS(CAS[, autoload])	Search for a chemical by its CAS number.
search_InChI(InChI[, autoload])	Search for a chemical by its InChI string.
search_InChI_key(InChI_key[, autoload])	Search for a chemical by its InChI key.
search_formula(formula[, autoload])	Search for a chemical by its serialized formula.
search_name(name[, autoload])	Search for a chemical by its name.
search_pubchem(pubchem[, autoload])	Search for a chemical by its pubchem number.
search_smiles(smiles[, autoload])	Search for a chemical by its smiles string.

chemicals.identifiers.**get_pubchem_db**()

Helper function to delay the creation of the pubchem_db object.

This avoids loading the database when it is not needed.

1.14.4 Chemical Groups

It is convenient to tag some chemicals with labels like "refrigerant", or in a certain database or not. The following chemical groups are available.

chemicals.identifiers.**cryogenics = {'132259-10-0': 'Air', '1333-74-0': 'hydrogen', '630-08-0': 'carbon monoxide', '74-82-8': 'methane', '7439-90-9': 'krypton', '7440-01-9': 'neon', '7440-37-1': 'Argon', '7440-59-7': 'helium', '7440-63-3': 'xenon', '7727-37-9': 'nitrogen', '7782-39-0': 'deuterium', '7782-41-4': 'fluorine', '7782-44-7': 'oxygen'}**

chemicals.identifiers.**inerts = {'10043-92-2': 'radon', '10102-43-9': 'Nitric Oxide', '10102-44-0': 'Nitrogen Dioxide', '124-38-9': 'Carbon Dioxide', '132259-10-0': 'Air', '7439-90-9': 'krypton', '7440-01-9': 'Neon', '7440-37-1': 'Argon', '7440-59-7': 'Helium', '7440-63-3': 'Xenon', '7727-37-9': 'Nitrogen', '7732-18-5': 'water', '7782-41-4': 'fluorine', '7782-44-7': 'Oxygen', '7782-50-5': 'chlorine'}**

chemicals.identifiers.**dippr_compounds**()

Loads and returns a set of compounds known in the DIPPR database. This can be useful for knowing if a chemical is of industrial relevance.

Returns

dippr_compounds [set([str])] A set of CAS numbers from the 2014 edition of the DIPPR database.

1.15 Surface Tension (chemicals.interface)

This module contains various surface tension estimation routines, dataframes of fit coefficients, fitting model equations, mixing rules, and water-hydrocarbon interfacial tension estimation routines.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.15.1 Pure Component Correlations

chemicals.interface.**Brock_Bird**(*T*, *Tb*, *Tc*, *Pc*)

Calculates air-liquid surface tension using the [1] emperical method. Old and tested.

$$\sigma = P_c^{2/3} T_c^{1/3} Q (1 - T_r)^{11/9}$$

$$Q = 0.1196 \left[1 + \frac{T_{br} \ln(P_c/1.01325)}{1 - T_{br}} \right] - 0.279$$

Parameters

T [float] Temperature of fluid [K]

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

sigma [float] Liquid surface tension, N/m

Notes

Numerous arrangements of this equation are available. This is DIPPR Procedure 7A: Method for the Surface Tension of Pure, Nonpolar, Nonhydrocarbon Liquids The exact equation is not in the original paper. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

p-dichloribenzene at 412.15 K, from DIPPR; value differs due to a slight difference in method.

```
>>> Brock_Bird(412.15, 447.3, 685, 3.952E6)
0.022084483251924 95
```

Chlorobenzene from Poling, as compared with a % error value at 293 K.

```
>>> Brock_Bird(293.15, 404.75, 633.0, 4530000.0)
0.032985686413713036
```

chemicals.interface.**Pitzer_sigma**(*T*, *Tc*, *Pc*, *omega*)

Calculates air-liquid surface tension using the correlation derived by [1] from the works of [2] and [3]. Based on critical property CSP methods.

$$\sigma = P_c^{2/3} T_c^{1/3} \frac{1.86 + 1.18\omega}{19.05} \left[\frac{3.75 + 0.91\omega}{0.291 - 0.08\omega}\right]^{2/3} (1 - T_r)^{11/9}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

sigma [float] Liquid surface tension, N/m

Notes

The source of this equation has not been reviewed. Internal units of presure are bar, surface tension of mN/m. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1], [2], [3]

Examples

Chlorobenzene from Poling, as compared with a % error value at 293 K.

```
>>> Pitzer_sigma(293., 633.0, 4530000.0, 0.249)
0.03458453513446388
```

chemicals.interface.**Sastri_Rao**(*T*, *Tb*, *Tc*, *Pc*, *chemicaltype=None*)

Calculates air-liquid surface tension using the correlation derived by [1] based on critical property CSP methods and chemical classes.

$$\sigma = K P_c^x T_b^y T_c^z \left[\frac{1 - T_r}{1 - T_{br}}\right]^m$$

Parameters

T [float] Temperature of fluid [K]

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

sigma [float] Liquid surface tension, N/m

Notes

The source of this equation has not been reviewed. Internal units of presure are bar, surface tension of mN/m. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Chlorobenzene from Poling, as compared with a % error value at 293 K.

```
>>> Sastri_Rao(293.15, 404.75, 633.0, 4530000.0)
0.03234567739694441
```

chemicals.interface.**Zuo_Stenby**(*T*, *Tc*, *Pc*, *omega*)

Calculates air-liquid surface tension using the reference fluids methods of [1].

$$\sigma^{(1)} = 40.520(1 - T_r)^{1.287}$$

$$\sigma^{(2)} = 52.095(1 - T_r)^{1.21548}$$

$$\sigma_r = \sigma_r^{(1)} + \frac{\omega - \omega^{(1)}}{\omega^{(2)} - \omega^{(1)}}(\sigma_r^{(2)} - \sigma_r^{(1)})$$

$$\sigma = T_c^{1/3} P_c^{2/3} [\exp(\sigma_r) - 1]$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

sigma [float] Liquid surface tension, N/m

Notes

Presently untested. Have not personally checked the sources. The reference values for methane and n-octane are from the DIPPR database. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Chlorobenzene

```
>>> Zuo_Stenby(293., 633.0, 4530000.0, 0.249)
0.03345569011871088
```

chemicals.interface.**Hakim_Steinberg_Stiel**(*T*, *Tc*, *Pc*, *omega*, *StielPolar=0.0*)

Calculates air-liquid surface tension using the reference fluids methods of [1].

$$\sigma = 4.60104 \times 10^{-7} P_c^{2/3} T_c^{1/3} Q_p \left(\frac{1 - T_r}{0.4}\right)^m$$

$$Q_p = 0.1574 + 0.359\omega - 1.769\chi - 13.69\chi^2 - 0.51\omega^2 + 1.298\omega\chi$$

$$m = 1.21 + 0.5385\omega - 14.61\chi - 32.07\chi^2 - 1.65\omega^2 + 22.03\omega\chi$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

StielPolar [float, optional] Stiel Polar Factor, [-]

Returns

sigma [float] Liquid surface tension, N/m

Notes

Original equation for m and Q are used. Internal units are atm and mN/m. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

1-butanol, as compared to value in CRC Handbook of 0.02493.

```
>>> Hakim_Steinberg_Stiel(298.15, 563.0, 4414000.0, 0.59, StielPolar=-0.07872)
0.02190790257519
```

chemicals.interface.**Miqueu**(*T*, *Tc*, *Vc*, *omega*)

Calculates air-liquid surface tension using the methods of [1].

$$\sigma = kT_c\left(\frac{N_a}{V_c}\right)^{2/3}(4.35+4.14\omega)t^{1.26}(1+0.19t^{0.5}-0.487t)$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Vc [float] Critical volume of fluid [m^3/mol]

omega [float] Acentric factor for fluid, [-]

Returns

sigma [float] Liquid surface tension, N/m

Notes

Uses Avogadro's constant and the Boltsman constant. Internal units of volume are mL/mol and mN/m. However, either a typo is in the article or author's work, or my value of k is off by 10; this is corrected nonetheless. Created with 31 normal fluids, none polar or hydrogen bonded. Has an AARD of 3.5%. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Bromotrifluoromethane, 2.45 mN/m

```
>>> Miqueu(300., 340.1, 0.000199, 0.1687)
0.003474100774091376
```

chemicals.interface.**Aleem**(*T*, *MW*, *Tb*, *rhol*, *Hvap_Tb*, *Cpl*)

Calculates vapor-liquid surface tension using the correlation derived by [1] based on critical property CSP methods.

$$\sigma = \phi \frac{MW^{1/3}}{6N_A^{1/3}} \rho_l^{2/3} \left[H_{vap} + C_{p,l}(T_b - T)\right]$$

$$\phi = 1 - 0.0047MW + 6.8 \times 10^{-6} MW^2$$

Parameters

T [float] Temperature of fluid [K]

MW [float] Molecular weight [g/mol]

Tb [float] Boiling temperature of the fluid [K]

rhol [float] Liquid density at T and P [kg/m^3]

Hvap_Tb [float] Mass enthalpy of vaporization at the normal boiling point [kg/m^3]

Cpl [float] Liquid heat capacity of the chemical at T [J/kg/K]

Returns

sigma [float] Liquid-vapor surface tension [N/m]

Notes

Internal units of molecuar weight are kg/mol. This model is dimensionally consistent.

This model does not use the critical temperature. After it predicts a surface tension of 0 at a sufficiently high temperature, it returns negative results. The temperature at which this occurs (the "predicted" critical temperature) can be calculated as follows:

$$\sigma = 0 \rightarrow T_{c,predicted} \text{ at } T_b + \frac{H_{vap}}{Cp_l}$$

To handle this case, if Tc is larger than T, 0 is returned as the model would return complex numbers.

Because of its dependence on density, it has the potential to model the effect of pressure on surface tension.

Claims AAD of 4.3%. Developed for normal alkanes. Total of 472 data points. Behaves worse for higher alkanes. Behaves very poorly overall.

References

[1]

Examples

Methane at 90 K

```
>>> Aleem(T=90, MW=16.04246, Tb=111.6, rhol=458.7, Hvap_Tb=510870.,
... Cpl=2465.)
0.01669970230131523
```

chemicals.interface.**Mersmann_Kind_sigma**(*T*, *Tm*, *Tb*, *Tc*, *Pc*, *n_associated=1*)

Estimates the surface tension of organic liquid substances according to the method of [1].

$$\sigma^* = \frac{\sigma n_{ass}^{1/3}}{(kT_c)^{1/3} T_{rm} P_c^{2/3}}$$

$$\sigma^* = \left(\frac{T_b - T_m}{T_m}\right)^{1/3} \left[6.25(1 - T_r) + 31.3(1 - T_r)^{4/3}\right]$$

Parameters

T [float] Temperature of the fluid [K]

Tm [float] Melting temperature [K]

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

n_associated [float] Number of associated molecules in a cluster (2 for alcohols, 1 otherwise), [-]

Returns

sigma [float] Liquid-vapor surface tension [N/m]

Notes

In the equation, all quantities must be in SI units. k is the boltzman constant. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

MTBE at STP (the actual value is 0.0181):

```
>>> Mersmann_Kind_sigma(298.15, 164.15, 328.25, 497.1, 3430000.0)
0.016744311449290426
```

chemicals.interface.**sigma_Gharagheizi_1**(*T*, *Tc*, *MW*, *omega*)

Calculates air-liquid surface tension using the equation 4 derived in [1] by gene expression programming.

$$\sigma = 8.948226 \times 10^{-4} \left[\frac{A^2}{MW}\sqrt{\frac{A\omega}{MW}}\right]^{0.5}$$

$$A = (T_c - T - \omega)$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

MW [float] Molecular weight [g/mol]

omega [float] Acentric factor for fluid, [-]

Returns

sigma [float] Liquid surface tension, N/m

Notes

This equation may fail before the critical point. In this case it returns 0.0 If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Methane at 93 K, point from [1]'s supporting material:

```
>>> sigma_Gharagheizi_1(T=95, Tc=190.564, MW=16.04, omega=0.012)
0.0110389739
```

chemicals.interface.**sigma_Gharagheizi_2**(*T*, *Tb*, *Tc*, *Pc*, *Vc*)

Calculates air-liquid surface tension using the equation 6 derived in [1] by gene expression programming.

$$\frac{\sigma}{\text{N/m}} = 10^{-4}\left(\frac{P_c}{\text{bar}}\right)^{2/3}\left(\frac{T_c}{\text{K}}\right)^{1/3}(1-T_r)^{11/9}\left[7.728729T_{br} + 2.476318\left(T_{br}^3 + \frac{V_c}{\text{m}^3/\text{kmol}}\right)\right]$$

Parameters

T [float] Temperature of fluid [K]

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

MW [float] Molecular weight [g/mol]

Vc [float] Critical volume of fluid [m^3/mol]

Returns

sigma [float] Liquid surface tension, N/m

Notes

This expression gives does converge to 0 at the critical point. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Methane at 93 K, point from [1]'s supporting material:

```
>>> sigma_Gharagheizi_2(T=95, Tb=111.66, Tc=190.564, Pc=45.99e5, Vc=0.0986e-3)
0.01674894057
```

1.15.2 Mixing Rules

chemicals.interface.**Winterfeld_Scriven_Davis**(*xs*, *sigmas*, *rhoms*)

Calculates surface tension of a liquid mixture according to mixing rules in [1] and also in [2].

$$\sigma_M = \sum_i \sum_j \frac{1}{V_L^{L2}} (x_i V_i) (x_j V_j) \sqrt{\sigma_i \cdot \sigma_j}$$

Parameters

xs [array-like] Mole fractions of all components, [-]

sigmas [array-like] Surface tensions of all components, [N/m]

rhoms [array-like] Molar densities of all components, [mol/m^3]

Returns

sigma [float] Air-liquid surface tension of mixture, [N/m]

Notes

DIPPR Procedure 7C: Method for the Surface Tension of Nonaqueous Liquid Mixtures

Becomes less accurate as liquid-liquid critical solution temperature is approached. DIPPR Evaluation: 3-4% AARD, from 107 nonaqueous binary systems, 1284 points. Internally, densities are converted to kmol/m^3. The Amgat function is used to obtain liquid mixture density in this equation.

Raises a ZeroDivisionError if either molar volume are zero, and a ValueError if a surface tensions of a pure component is negative.

References

[1], [2]

Examples

```
>>> Winterfeld_Scriven_Davis([0.1606, 0.8394], [0.01547, 0.02877],
... [8610., 15530.])
0.02496738845043982
```

chemicals.interface.**Weinaug_Katz**(*parachors*, *Vml*, *Vmg*, *xs*, *ys*)

Calculates surface tension of a liquid mixture according to mixing rules in [1] and also in [2]. This is based on the Parachor concept. This is called the Macleod-Sugden model in some places.

$$\sigma_M = \left[\sum_i P_i \left(\frac{x_i}{V_{m,l}} - \frac{y_i}{V_{m,g}}\right)\right]^4$$

Parameters

parachors [list[float]] Parachors of each component, [N^0.25*m^2.75/mol]

Vml [float] Liquid mixture molar volume, [m^3/mol]

Vmg [float] Gas mixture molar volume; this can be set to zero at low pressures, [m^3/mol]

xs [list[float]] Mole fractions of all components in liquid phase, [-]

xs [list[float]] Mole fractions of all components in gas phase, [-]

Returns

sigma [float] Air-liquid surface tension of mixture, [N/m]

Notes

This expression is efficient and does not require pure component surface tensions. Its accuracy is dubious.

References

[1], [2]

Examples

```
>>> Weinaug_Katz([5.1e-5, 7.2e-5], Vml=0.000125, Vmg=0.02011, xs=[.4, .6], ys=[.6, .
↪4])
0.06547479150776776
```

Neglect the vapor phase density by setting *Vmg* to a high value:

```
>>> Weinaug_Katz([5.1e-5, 7.2e-5], Vml=0.000125, Vmg=1e100, xs=[.4, .6], ys=[.6, .
↪4])
0.06701752894095361
```

chemicals.interface.**Diguilio_Teja**(*T*, *xs*, *sigmas_Tb*, *Tbs*, *Tcs*)

Calculates surface tension of a liquid mixture according to mixing rules in [1].

$$\sigma = 1.002855(T^*)^{1.118091}\frac{T}{T_b}\sigma_r$$

$$T^* = \frac{(T_c/T) - 1}{(T_c/T_b) - 1}$$

$$\sigma_r = \sum x_i \sigma_i$$

$$T_b = \sum x_i T_{b,i}$$

$$T_c = \sum x_i T_{c,i}$$

Parameters

T [float] Temperature of fluid [K]

xs [array-like] Mole fractions of all components

sigmas_Tb [array-like] Surface tensions of all components at the boiling point, [N/m]

Tbs [array-like] Boiling temperatures of all components, [K]

Tcs [array-like] Critical temperatures of all components, [K]

Returns

sigma [float] Air-liquid surface tension of mixture, [N/m]

Notes

Simple model, however it has 0 citations. Gives similar results to the *Winterfeld_Scriven_Davis* model.

Raises a ValueError if temperature is greater than the mixture's critical temperature or if the given temperature is negative, or if the mixture's boiling temperature is higher than its critical temperature.

[1] claims a 4.63 percent average absolute error on 21 binary and 4 ternary non-aqueous systems. [1] also considered Van der Waals mixing rules for *Tc*, but found it provided a higher error of 5.58%

References

[1]

Examples

```
>>> Diguilio_Teja(T=298.15, xs=[0.1606, 0.8394],
... sigmas_Tb=[0.01424, 0.02530], Tbs=[309.21, 312.95], Tcs=[469.7, 508.0])
0.025716823875045505
```

1.15.3 Correlations for Specific Substances

chemicals.interface.**sigma_IAPWS**(T)

Calculate the surface tension of pure water as a function of . temperature. Assumes the 2011 IAPWS [1] formulation.

$$\sigma = B\tau^{\mu}(1 + b\tau)$$

$$\tau = 1 - T/T_c$$

$$B = 0.2358\text{N/m}$$

$$b = -0.625$$

$$\mu = 1.256$$

Parameters

T [float] Temperature of liquid [K]

Returns

sigma [float] Air-liquid surface tension, [N/m]

Notes

This function is valid from the triple temperature to the critical temperature. No effects for pressure are included in the formulation. Test values are from IAPWS 2010 book.

The equation is valid from the triple point to the critical point, 647.096 K; but [1] also recommends its use down to -25°C.

If a value larger than the critical temperature is input, 0.0 is returned.

References

[1]

Examples

```
>>> sigma_IAPWS(300.)
0.0716859625271
>>> sigma_IAPWS(450.)
0.0428914991565
>>> sigma_IAPWS(600.)
0.0083756108728
```

1.15.4 Petroleum Correlations

chemicals.interface.**API10A32**(*T*, *Tc*, *K_W*)

Calculates the interfacial tension between a liquid petroleum fraction and air, using the oil's pseudocritical temperature and Watson K Characterization factor.

$$\sigma = \frac{673.7\left[\frac{(T_c - T)}{T_c}\right]^{1.232}}{K_W}$$

Parameters

T [float] Liquid temperature, [K]

Tc [float] Pseudocritical temperature (or critical temperature if using the equation with a pure component), [K]

K_W [float] Watson characterization factor

Returns

sigma [float] Air-liquid surface tension, [N/m]

Notes

[1] cautions that this should not be applied to coal liquids, and that it will give higher errors at pressures above 500 psi. [1] claims this has an average error of 10.7%.

This function converges to zero at *Tc*. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Sample problem in Comments on Procedure 10A3.2.1 of [1];

```
>>> from fluids.core import F2K, R2K
>>> API10A32(T=F2K(60), Tc=R2K(1334), K_W=12.4)
29.577333312096968
```

1.15.5 Oil-Water Interfacial Tension Correlations

chemicals.interface.**Meybodi_Daryasafar_Karimi**(*rho_water*, *rho_oil*, *T*, *Tc*)

Calculates the interfacial tension between water and a hydrocabon liquid according to the correlation of [1].

$$\gamma_{hw} = \left(\frac{A_1 + A_2\Delta\rho + A_3\Delta\rho^2 + A_4\Delta\rho^3}{A_5 + A_6\frac{T^{A_7}}{T_{c,h}} + A_8 T^{A_9}}\right)^{A_{10}}$$

Parameters

rho_water [float] The density of the aqueous phase, [kg/m^3]

rho_oil [float] The density of the hydrocarbon phase, [kg/m^3]

T [float] Temperature of the fluid, [K]

Tc [float] Critical temperature of the hydrocarbon mixture, [K]

Returns

sigma [float] Hydrocarbon-water surface tension [N/m]

Notes

Internal units of the equation are g/mL and mN/m.

References

[1]

Examples

```
>>> Meybodi_Daryasafar_Karimi(980, 760, 580, 914)
0.02893598143089256
```

1.15.6 Fit Correlations

chemicals.interface.**REFPROP_sigma**(*T*, *Tc*, *sigma0*, *n0*, *sigma1=0.0*, *n1=0.0*, *sigma2=0.0*, *n2=0.0*)

Calculates air-liquid surface tension using the REFPROP_sigma [1] regression-based method. Relatively recent, and most accurate.

$$\sigma(T) = \sigma_0\left(1 - \frac{T}{T_c}\right)^{n_0} + \sigma_1\left(1 - \frac{T}{T_c}\right)^{n_1} + \sigma_2\left(1 - \frac{T}{T_c}\right)^{n_2}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

sigma0 [float] First emperical coefficient of a fluid

n0 [float] First emperical exponent of a fluid

sigma1 [float, optional] Second emperical coefficient of a fluid.

n1 [float, optional] Second emperical exponent of a fluid.

sigma1 [float, optional] Third emperical coefficient of a fluid.

n2 [float, optional] Third emperical exponent of a fluid.

Returns

sigma [float] Liquid surface tension, [N/m]

Notes

Function as implemented in [1]. No example necessary; results match literature values perfectly. Form of function returns imaginary results when T > Tc; 0 is returned if this is the case.

When fitting parameters to this function, it is easy to end up with a fit that returns negative surface tension near but not quite at the critical point.

References

[1]

Examples

Parameters for water at 298.15 K

```
>>> REFPROP_sigma(298.15, 647.096, -0.1306, 2.471, 0.2151, 1.233)
0.07205503890847453
```

chemicals.interface.**Somayajulu**(*T*, *Tc*, *A*, *B*, *C*)

Calculates air-liquid surface tension using the [1] emperical (parameter-regressed) method. Well regressed, no recent data.

$$\sigma = aX^{5/4} + bX^{9/4} + cX^{13/4}$$

$$X = (T_c - T)/T_c$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

A [float] Regression parameter

B [float] Regression parameter

C [float] Regression parameter

Returns

sigma [float] Liquid surface tension, N/m

Notes

Presently untested, but matches expected values. Internal units are mN/m. Form of function returns imaginary results when T > Tc; 0.0 is returned if this is the case. Function is claimed valid from the triple to the critical point. Results can be evaluated beneath the triple point.

This function can be accidentally fit to return negative values of surface tension.

References

[1]

Examples

Water at 300 K

```
>>> Somayajulu(300, 647.126, 232.713514, -140.18645, -4.890098)
0.07166386387996758
```

chemicals.interface.**Jasper**(T, a, b)

Calculates surface tension of a fluid given two parameters, a linear fit in Celcius from [1] with data reprinted in [2].

$$\sigma = a - bT$$

Parameters

T [float] Temperature of fluid, [K]

a [float] Parameter for equation. Chemical specific.

b [float] Parameter for equation. Chemical specific.

Returns

sigma [float] Surface tension [N/m]

Notes

Internal units are mN/m, and degrees Celcius. This function has been checked against several references.

As this is a linear model, negative values of surface tension will eventually arise. 0 is returned in these cases.

References

[1], [2]

Examples

```
>>> Jasper(298.15, 24, 0.0773)
0.0220675
```

chemicals.interface.**PPDS14**(*T*, *Tc*, *a0*, *a1*, *a2*)

Calculates air-liquid surface tension using the [1] emperical (parameter-regressed) method, called the PPDS 14 equation for surface tension.

$$\sigma = a_0 \tau^{a_1} (1 + a_2 \tau)$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a0 [float] Regression parameter, [N/m]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

Returns

sigma [float] Liquid surface tension, [N/m]

Notes

If Tc is larger than T, 0 is returned as the model would return complex numbers.

If this model is fit with *a0* and *a2* as positive values, it is guaranteed to predict only positive values of *sigma* right up to the critical point. However, *a2* is often fit to be a negative value.

References

[1], [2]

Examples

Benzene at 280 K from [1]

```
>>> PPDS14(T=280, Tc=562.05, a0=0.0786269, a1=1.28646, a2=-0.112304)
0.030559764256249854
```

chemicals.interface.**Watson_sigma**(*T*, *Tc*, *a1*, *a2*, *a3=0.0*, *a4=0.0*, *a5=0.0*)

Calculates air-liquid surface tension using the Watson [1] emperical (parameter-regressed) method developed by NIST.

$$\sigma = \exp\left[a_1 + \ln(1 - T_r)\left(a_2 + a_3 T_r + a_4 T_r^2 + a_5 T_r^3\right)\right]$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

a5 [float] Regression parameter, [-]

Returns

sigma [float] Liquid surface tension, [N/m]

Notes

This expression is also used for enthalpy of vaporization in [1]. The coefficients from NIST TDE for enthalpy of vaporization are kJ/mol.

This model is coded to return 0 values at Tr >= 1. It is otherwise not possible to evaluate this expression at Tr = 1, as log(0) is undefined (although the limit shows the expression converges to 0).

This equation does not have any term forcing it to become near-zero at the critical point, but it cannot be fit so as to produce negative values.

References

[1]

Examples

Isooctane at 350 K from [1]:

```
>>> Watson_sigma(T=350.0, Tc=543.836, a1=-3.02417, a2=1.21792, a3=-5.26877e-9, a4=5.
↪62659e-9, a5=-2.27553e-9)
0.0138340926605649
```

chemicals.interface.**ISTExpansion**(*T*, *Tc*, *a1*, *a2*, *a3=0.0*, *a4=0.0*, *a5=0.0*)

Calculates air-liquid surface tension using the IST expansion [1] emperical (parameter-regressed) method developed by NIST.

$$\sigma = \sum_i a_i \left(1 - \frac{T}{T_c}\right)^i$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

a5 [float] Regression parameter, [-]

Returns

sigma [float] Liquid surface tension, [N/m]

Notes

This equation hsa a term term forcing it to become zero at the critical point, but it can easily be fit so as to produce negative values at any reduced temperature.

References

[1]

Examples

Diethyl phthalate at 400 K from [1]:

```
>>> ISTExpansion(T=400.0, Tc=776.0, a1=0.037545, a2=0.0363288)
0.02672100905515996
```

1.15.7 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.interface.**sigma_data_Mulero_Cachadina**
Data from [5] with *REFPROP_sigma* coefficients.

chemicals.interface.**sigma_data_Jasper_Lange**
Data as shown in [4] but originally in [3] with *Jasper* coefficients.

chemicals.interface.**sigma_data_Somayajulu**
Data from [1] with *Somayajulu* coefficients.

chemicals.interface.**sigma_data_Somayajulu2**
Data from [2] with *Somayajulu* coefficients. These should be preferred over the original coefficients.

chemicals.interface.**sigma_data_VDI_PPDS_11**
Data from [6] with *chemicals.dippr.EQ106* coefficients.

The structure of each dataframe is shown below:

```
In [1]: import chemicals

In [2]: chemicals.interface.sigma_data_Mulero_Cachadina
Out[2]:
                                                  Fluid  ...    Tmax
CAS                                                      ...
60-29-7                                  Diethyl ether  ...  453.15
64-17-5                                       Ethanol   ...  513.15
67-56-1                                      Methanol   ...  508.15
67-64-1                                       Acetone   ...  353.15
71-43-2                                       Benzene   ...  553.15
...                                               ...  ...     ...
7783-54-2                       Nitrogen trifluoride  ...  206.36
7789-20-0                                        D2O  ...  642.02
10024-97-2                              Nitrous oxide   ...  293.15
22410-44-2       RE245cb2 (Methyl-pentafluoroethyl ether)  ...  353.41
29118-24-9  R1234ze(E) (trans-1,3,3,3-tetrafluoropropene)  ...  373.14
```

(continues on next page)

(continued from previous page)

```
[115 rows x 10 columns]

In [3]: chemicals.interface.sigma_data_Jasper_Lange
Out[3]:
                              Name      a       b    Tmin    Tmax
CAS
55-21-0                  Benzamide   47.26  0.0705  402.15  563.15
55-63-0     Glycerol tris(nitrate)   55.74  0.2504  286.15  433.15
56-23-5       Carbon tetrachloride   29.49  0.1224  250.15  349.85
57-06-7       Allyl isothiocyanate   36.76  0.1074  193.15  425.15
60-29-7               Diethyl ether   18.92  0.0908  157.15  307.75
...                             ...    ...     ...     ...     ...
13952-84-6          sec-Butylamine   23.75  0.1057  169.15  336.15
14901-07-6                -Ionone   35.36  0.0950  401.15  401.15
18854-56-3     1,2-Dipropoxyethane   25.03  0.0972     NaN     NaN
19550-30-2  2,3-Dimethyl-1-butanol   26.22  0.0992  259.15  391.15
40626-78-6          2-Methylhexane   21.22  0.0966  155.15  363.15

[522 rows x 5 columns]

In [4]: chemicals.interface.sigma_data_Somayajulu
Out[4]:
                      Chemical      Tt      Tc         A         B          C
CAS
60-29-7            Ethyl ether   157.00  466.74   61.0417   -6.7908    0.14046
64-17-5                Ethanol   159.00  513.92  111.4452 -146.0229   89.57030
64-19-7            Acetic acid   290.00  592.70   91.9020  -91.7035   77.50720
67-56-1                Methanol  175.59  512.64  122.6257 -199.4044  153.37440
71-23-8              Propanaol   147.00  536.78  107.1238 -133.8128   84.43570
...                        ...     ...     ...       ...       ...        ...
10035-10-6    Hydrogen bromide   187.15  363.20   74.0521   20.1043  -30.25710
10102-43-9        Nitric oxide   112.15  180.00   58.6304   97.8722  -33.67390
13465-07-1  Hydrogen disulfide   185.15  572.00  130.1176  -40.6216    4.77160
17778-80-2              Oxygen    54.35  154.58   38.2261    5.6316   -7.74050
19287-45-7             Diborane   104.15  289.80   38.0417   29.7743  -24.26050

[64 rows x 6 columns]

In [5]: chemicals.interface.sigma_data_Somayajulu2
Out[5]:
                      Chemical      Tt      Tc         A         B          C
CAS
60-29-7            Ethyl ether   157.00  466.74   61.0417   -6.7908    0.14046
64-17-5                Ethanol   159.00  513.92  111.4452 -146.0229   89.57030
64-19-7            Acetic acid   290.00  592.70   91.9020  -91.7035   77.50720
67-56-1                Methanol  175.59  512.64  122.6257 -199.4044  153.37440
71-23-8              Propanaol   147.00  536.78  107.1238 -133.8128   84.43570
...                        ...     ...     ...       ...       ...        ...
10035-10-6    Hydrogen bromide   187.15  363.20   74.0521   20.1043  -30.25710
10102-43-9        Nitric oxide   112.15  180.00   58.6304   97.8722  -33.67390
13465-07-1  Hydrogen disulfide   185.15  572.00  150.6970 -102.9100   56.72580
```

(continues on next page)

(continued from previous page)

```
17778-80-2              Oxygen    54.35  154.58   38.2261    5.6316   -7.74050
19287-45-7            Diborane   104.15  289.80   38.0417   29.7743  -24.26050

[64 rows x 6 columns]

In [6]: chemicals.interface.sigma_data_VDI_PPDS_11
Out[6]:
                        Chemical      Tm      Tc  ...        C        D        E
CAS                                               ...
50-00-0              Formaldehyde  181.15  408.05  ...  0.00000  0.00000  0.00000
56-23-5      Carbon tetrachloride  250.25  556.35  ...  0.00000  0.00000  0.00000
56-81-5                  Glycerol  291.45  850.05  ...  0.00000  0.00000  0.00000
60-29-7              Diethyl ether  156.75  466.63  ...  0.00000  0.00000  0.00000
62-53-3                    Aniline  267.15  699.05  ...  0.00000  0.00000  0.00000
...                            ...     ...     ...  ...      ...      ...      ...
10097-32-2                 Bromine  265.85  584.15  ...  0.00000  0.00000  0.00000
10102-43-9              Nitric oxide  112.15  180.15  ...  0.00000  0.00000  0.00000
10102-44-0         Nitrogen dioxide  261.85  431.15  ...  0.00000  0.00000  0.00000
10544-72-6    Dinitrogentetroxide  261.85  431.10  ...  0.00000  0.00000  0.00000
132259-10-0                   Air   63.05  132.53  ...  0.06889  0.17918 -0.14564

[272 rows x 8 columns]
```

1.16 Lennard-Jones Models (chemicals.lennard_jones)

This module contains lookup functions and estimation methods for the parameters molecular diameter *sigma* and the Stockmayer parameter *epsilon*. These are used for diffusivity calculations. It also contains several methods for computing the collision integral, another parameter used in the Lennard-Jones model.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.16.1 Stockmayer Parameter

chemicals.lennard_jones.**Stockmayer**(*CASRN=''*, *Tm=None*, *Tb=None*, *Tc=None*, *Zc=None*, *omega=None*, *method=None*)

This function handles the retrieval or calculation a chemical's Stockmayer parameter. Values are available from one source with lookup based on CASRNs, or can be estimated from 7 CSP methods. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Preferred sources are 'Magalhães, Lito, Da Silva, and Silva (2013)' for common chemicals which had valies listed in that source, and the CSP method *Tee, Gotoh, and Stewart CSP with Tc, omega (1966)* for chemicals

which don't.

Parameters

CASRN [str, optional] CASRN [-]

Tm [float, optional] Melting temperature of compound [K]

Tb [float, optional] Boiling temperature of compound [K]

Tc [float, optional] Critical temperature of compound, [K]

Zc [float, optional] Critical compressibility of compound, [-]

omega [float, optional] Acentric factor of compound, [-]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in Stockmayer_all_methods

Notes

These values are somewhat rough, as they attempt to pigeonhole a chemical into L-J behavior.

The tabulated data is from [2], for 322 chemicals.

References

[1], [2]

Examples

```
>>> Stockmayer(CASRN='64-17-5')
1291.41
>>> Stockmayer('7727-37-9')
71.4
```

chemicals.lennard_jones.**Stockmayer_methods**(*CASRN=None*, *Tm=None*, *Tb=None*, *Tc=None*, *Zc=None*, *omega=None*)

Return all methods available to obtain the Stockmayer parameter for the desired chemical.

Parameters

CASRN [str, optional] CASRN [-]

Tm [float, optional] Melting temperature of compound [K]

Tb [float, optional] Boiling temperature of compound [K]

Tc [float, optional] Critical temperature of compound, [K]

Zc [float, optional] Critical compressibility of compound, [-]

omega [float, optional] Acentric factor of compound, [-]

Returns

methods [list[str]] Methods which can be used to obtain *Stockmayer* with the given inputs.

See also:

Stockmayer

chemicals.lennard_jones.**Stockmayer_all_methods = ('Magalhães, Lito, Da Silva, and Silva (2013)', 'Poling et al. (2001)', 'Tee, Gotoh, and Stewart CSP with Tc, omega (1966)', 'Stiel and Thodos Tc, Zc (1962)', 'Flynn (1960)', 'Bird, Stewart, and Light (2002) critical relation', 'Tee, Gotoh, and Stewart CSP with Tc (1966)', 'Bird, Stewart, and Light (2002) boiling relation', 'Bird, Stewart, and Light (2002) melting relation')**

Tuple of method name keys. See the *Stockmayer* for the actual references

1.16.2 Stockmayer Parameter Correlations

chemicals.lennard_jones.**epsilon_Flynn**(*Tc*)

Calculates Lennard-Jones depth of potential-energy minimum. Uses critical temperature. CSP method by [1] as reported in [2].

$$\epsilon/k = 1.77 T_c^{5/6}$$

Parameters

Tc [float] Critical temperature of fluid [K]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

References

[1], [2]

Examples

```
>>> epsilon_Flynn(560.1)
345.2984087011443
```

chemicals.lennard_jones.**epsilon_Bird_Stewart_Lightfoot_critical**(*Tc*)

Calculates Lennard-Jones depth of potential-energy minimum. Uses critical temperature. CSP method by [1].

$$\epsilon/k = 0.77 T_c$$

Parameters

Tc [float] Critical temperature of fluid [K]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

References

[1]

Examples

```
>>> epsilon_Bird_Stewart_Lightfoot_critical(560.1)
431.27700000000004
```

chemicals.lennard_jones.**epsilon_Bird_Stewart_Lightfoot_boiling**(*Tb*)
Calculates Lennard-Jones depth of potential-energy minimum. Uses boiling temperature. CSP method by [1].

$$\epsilon/k = 1.15T_b$$

Parameters

Tb [float] Boiling temperature [K]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

References

[1]

Examples

```
>>> epsilon_Bird_Stewart_Lightfoot_boiling(357.85)
411.5275
```

chemicals.lennard_jones.**epsilon_Bird_Stewart_Lightfoot_melting**(*Tm*)
Calculates Lennard-Jones depth of potential-energy minimum. Uses melting temperature. CSP method by [1].

$$\epsilon/k = 1.92T_m$$

Parameters

Tm [float] Melting temperature [K]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

References

[1]

Examples

```
>>> epsilon_Bird_Stewart_Lightfoot_melting(231.15)
443.808
```

chemicals.lennard_jones.**epsilon_Stiel_Thodos**(*Tc*, *Zc*)

Calculates Lennard-Jones depth of potential-energy minimum. Uses Critical temperature and critical compressibility. CSP method by [1].

$$\epsilon/k = 65.3 T_c Z_c^{3.6}$$

Parameters

Tc [float] Critical temperature of fluid [K]

Zc [float] Critical compressibility of fluid, [-]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

References

[1]

Examples

Fluorobenzene

```
>>> epsilon_Stiel_Thodos(358.5, 0.265)
196.3755830305783
```

chemicals.lennard_jones.**epsilon_Tee_Gotoh_Steward_1**(*Tc*)

Calculates Lennard-Jones depth of potential-energy minimum. Uses Critical temperature. CSP method by [1].

$$\epsilon/k = 0.7740 T_c$$

Parameters

Tc [float] Critical temperature of fluid [K]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

Notes

Further regressions with other parameters were performed in [1] but are not included here, except for *epsilon_Tee_Gotoh_Steward_2*.

References

[1]

Examples

```
>>> epsilon_Tee_Gotoh_Steward_1(560.1)
433.5174
```

chemicals.lennard_jones.**epsilon_Tee_Gotoh_Steward_2**(*Tc*, *omega*)

Calculates Lennard-Jones depth of potential-energy minimum. Uses critical temperature and acentric factor. CSP method by [1].

$$\epsilon/k = (0.7915 + 0.1693\omega)T_c$$

Parameters

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor for fluid, [-]

Returns

epsilon_k [float] Lennard-Jones depth of potential-energy minimum over k, [K]

Notes

Further regressions with other parameters were performed in [1] but are not included here, except for *epsilon_Tee_Gotoh_Steward_1*.

References

[1]

Examples

```
>>> epsilon_Tee_Gotoh_Steward_2(560.1, 0.245)
466.55125785
```

1.16.3 Molecular Diameter

chemicals.lennard_jones.**molecular_diameter**(*CASRN=None*, *Tc=None*, *Pc=None*, *Vc=None*, *Zc=None*, *omega=None*, *Vm=None*, *Vb=None*, *method=None*)

This function handles the retrieval or calculation a chemical's L-J molecular diameter. Values are available from one source with lookup based on CASRNs, or can be estimated from 9 CSP methods. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Preferred sources are 'Magalhães, Lito, Da Silva, and Silva (2013)' for common chemicals which had valies listed in that source, and the CSP method *Tee, Gotoh, and Stewart CSP with Tc, Pc, omega (1966)* for chemicals which don't.

Parameters

CASRN [str, optional] CASRN [-]

Tc [float, optional] Critical temperature, [K]

Pc [float, optional] Critical pressure, [Pa]

Vc [float, optional] Critical volume, [m^3/mol]

Zc [float, optional] Critical compressibility, [-]

omega [float, optional] Acentric factor of compound, [-]

Vm [float, optional] Molar volume of liquid at the melting point of the fluid [K]

Vb [float, optional] Molar volume of liquid at the boiling point of the fluid [K]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in molecular_diameter_all_methods

Notes

These values are somewhat rough, as they attempt to pigeonhole a chemical into L-J behavior.

The tabulated data is from [2], for 322 chemicals.

References

[1], [2]

Examples

```
>>> molecular_diameter(CASRN='64-17-5')
4.23738
>>> molecular_diameter('7727-37-9')
3.798
```

chemicals.lennard_jones.**molecular_diameter_methods**(*CASRN=None*, *Tc=None*, *Pc=None*, *Vc=None*, *Zc=None*, *omega=None*, *Vm=None*, *Vb=None*)

Return all methods available to obtain the molecular diameter for the desired chemical.

Parameters

CASRN [str, optional] CASRN [-]

Tc [float, optional] Critical temperature, [K]

Pc [float, optional] Critical pressure, [Pa]

Vc [float, optional] Critical volume, [m^3/mol]

Zc [float, optional] Critical compressibility, [-]

omega [float, optional] Acentric factor of compound, [-]

Vm [float, optional] Molar volume of liquid at the melting point of the fluid [K]

Vb [float, optional] Molar volume of liquid at the boiling point of the fluid [K]

Returns

methods [list[str]] Methods which can be used to obtain *molecular_diameter* with the given inputs.

See also:

molecular_diameter

chemicals.lennard_jones.**molecular_diameter_all_methods = ('Magalhães, Lito, Da Silva, and Silva (2013)', 'Poling et al. (2001)', 'Tee, Gotoh, and Stewart CSP with Tc, Pc, omega (1966)', 'Silva, Liu, and Macedo (1998) critical relation with Tc, Pc', 'Bird, Stewart, and Light (2002) critical relation with Tc, Pc', 'Tee, Gotoh, and Stewart CSP with Tc, Pc (1966)', 'Stiel and Thodos Vc, Zc (1962)', 'Flynn (1960)', 'Bird, Stewart, and Light (2002) critical relation with Vc', 'Bird, Stewart, and Light (2002) boiling relation', 'Bird, Stewart, and Light (2002) melting relation')**

Tuple of method name keys. See the *molecular_diameter* for the actual references

1.16.4 Molecular Diameter Correlations

chemicals.lennard_jones.**sigma_Flynn**(*Vc*)

Calculates Lennard-Jones molecular diameter. Uses critical volume. CSP method by [1] as reported in [2].

$$\sigma = 0.561(V_c^{1/3})^{5/4}$$

Parameters

Vc [float] Critical volume of fluid [m^3/mol]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Vc is originally in units of mL/mol.

References

[1], [2]

Examples

```
>>> sigma_Flynn(0.000268)
5.2506948422196285
```

chemicals.lennard_jones.**sigma_Bird_Stewart_Lightfoot_critical_2**(*Tc*, *Pc*)

Calculates Lennard-Jones molecular diameter. Uses critical temperature and pressure. CSP method by [1].

$$\sigma = 2.44(T_c/P_c)^{1/3}$$

Parameters

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Original units of critical pressure are atmospheres.

References

[1]

Examples

```
>>> sigma_Bird_Stewart_Lightfoot_critical_2(560.1, 4550000)
5.658657684653222
```

chemicals.lennard_jones.**sigma_Bird_Stewart_Lightfoot_critical_1**(*Vc*)

Calculates Lennard-Jones molecular diameter. Uses critical volume. CSP method by [1].

$$\sigma = 0.841 V_c^{1/3}$$

Parameters

Vc [float] Critical volume of fluid [m^3/mol]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Original units of Vc are mL/mol.

References

[1]

Examples

```
>>> sigma_Bird_Stewart_Lightfoot_critical_1(0.000268)
5.422184116631474
```

chemicals.lennard_jones.**sigma_Bird_Stewart_Lightfoot_boiling**(*Vb*)

Calculates Lennard-Jones molecular diameter. Uses molar volume of liquid at boiling. CSP method by [1].

$$\sigma = 1.166 V_{b,liq}^{1/3}$$

Parameters

Vb [float] Boiling molar volume of liquid [m^3/mol]

Returns

sigma [float] Lennard-Jones collision integral, [Angstrom]

Notes

Original units of Vb are mL/mol.

References

[1]

Examples

```
>>> sigma_Bird_Stewart_Lightfoot_boiling(0.0001015)
5.439018856944655
```

chemicals.lennard_jones.**sigma_Bird_Stewart_Lightfoot_melting**(*Vm*)

Calculates Lennard-Jones molecular diameter. Uses molar volume of a liquid at its melting point. CSP method by [1].

$$\sigma = 1.222 V_{m,sol}^{1/3}$$

Parameters

Vm [float] Melting molar volume of a liquid at its melting point [m^3/mol]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Original units of Vm are mL/mol.

References

[1]

Examples

```
>>> sigma_Bird_Stewart_Lightfoot_melting(8.8e-05)
5.435407341351406
```

chemicals.lennard_jones.**sigma_Stiel_Thodos**(*Vc*, *Zc*)

Calculates Lennard-Jones molecular diameter. Uses critical volume and compressibility. CSP method by [1].

$$\sigma = 0.1866 V_c^{1/3} Z_c^{-6/5}$$

Parameters

Vc [float] Critical volume of fluid [m^3/mol]

Zc [float] Critical compressibility of fluid, [-]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Vc is originally in units of mL/mol.

References

[1]

Examples

Monofluorobenzene

```
>>> sigma_Stiel_Thodos(0.000271, 0.265)
5.94300853971033
```

chemicals.lennard_jones.**sigma_Tee_Gotoh_Steward_1**(*Tc*, *Pc*)

Calculates Lennard-Jones molecular diameter. Uses critical temperature and pressure. CSP method by [1].

$$\sigma = 2.3647 \left(\frac{T_c}{P_c}\right)^{1/3}$$

Parameters

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Original units of Pc are atm. Further regressions with other parameters were performed in [1] but are not included here, except for *sigma_Tee_Gotoh_Steward_2*.

References

[1]

Examples

```
>>> sigma_Tee_Gotoh_Steward_1(560.1, 4550000)
5.48402779790962
```

chemicals.lennard_jones.**sigma_Tee_Gotoh_Steward_2**(*Tc*, *Pc*, *omega*)

Calculates Lennard-Jones molecular diameter. Uses critical temperature, pressure, and acentric factor. CSP method by [1].

$$\sigma = (2.3551 - 0.0874\omega)\left(\frac{T_c}{P_c}\right)^{1/3}$$

Parameters

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Original units of Pc are atm. Further regressions with other parameters were performed in [1] but are not included here, except for *sigma_Tee_Gotoh_Steward_1*.

References

[1]

Examples

```
>>> sigma_Tee_Gotoh_Steward_2(560.1, 4550000, 0.245)
5.412104867264477
```

chemicals.lennard_jones.**sigma_Silva_Liu_Macedo**(*Tc*, *Pc*)

Calculates Lennard-Jones molecular diameter. Uses critical temperature and pressure. CSP method by [1].

$$\sigma_{LJ}^3 = 0.17791 + 11.779\left(\frac{T_c}{P_c}\right) - 0.049029\left(\frac{T_c}{P_c}\right)^2$$

Parameters

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

sigma [float] Lennard-Jones molecular diameter, [Angstrom]

Notes

Pc is originally in bar. An excellent paper. None is returned if the polynomial returns a negative number, as in the case of 1029.13 K and 3.83 bar.

References

[1]

Examples

```
>>> sigma_Silva_Liu_Macedo(560.1, 4550000)
5.164483998730177
```

1.16.5 Utility Functions

chemicals.lennard_jones.**T_star**(*T*, *epsilon_k=None*, *epsilon=None*)

This function calculates the parameter *T_star* as needed in performing collision integral calculations.

$$T^* = \frac{kT}{\epsilon}$$

Parameters

epsilon_k [float, optional] Lennard-Jones depth of potential-energy minimum over k, [K]

epsilon [float, optional] Lennard-Jones depth of potential-energy minimum [J]

Returns

T_star [float] Dimentionless temperature for calculating collision integral, [-]

Notes

Tabulated values are normally listed as epsilon/k. k is the Boltzman constant, with units of J/K.

References

[1]

Examples

```
>>> T_star(T=318.2, epsilon_k=308.43)
1.0316765554582887
```

1.17 Miscellaneous Data (chemicals.miscdata)

This module contains several tables which are common to different lookup functions.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.17.1 Temperature Dependent data

chemicals.miscdata.**lookup_VDI_tabular_data**(*CASRN*, *prop*)

This function retrieves the tabular data available for a given chemical and a given property. Lookup is based on CASRNs. Length of data returned varies between chemicals. All data is at saturation condition from [1].

Function has data for 58 chemicals.

Parameters

CASRN [str] CASRN [-]

prop [string] Property [-]

Returns

Ts [list] Temperatures where property data is available, [K]

props [list] Properties at each temperature, [various]

Notes

The available properties are 'P', 'Density (l)', 'Density (g)', 'Hvap', 'Cp (l)', 'Cp (g)', 'Mu (l)', 'Mu (g)', 'K (l)', 'K (g)', 'Pr (l)', 'Pr (g)', 'sigma', 'Beta', 'Volume (l)', and 'Volume (g)'.

Data is available for all properties and all chemicals; surface tension data was missing for mercury, but added as estimated from the a/b coefficients listed in Jasper (1972) to simplify the function.

References

[1]

Examples

```
>>> lookup_VDI_tabular_data('67-56-1', 'Mu (g)')
([337.63, 360.0, 385.0, 410.0, 435.0, 460.0, 500.0], [1.11e-05, 1.18e-05, 1.27e-05,␣
↪1.36e-05, 1.46e-05, 1.59e-05, 2.04e-05])
>>> lookup_VDI_tabular_data('7782-41-4', 'sigma')
([53.49, 64.0, 74.0, 85.04, 92.0, 102.0, 112.0, 122.0, 132.0, 144.41], [0.0227, 0.
↪02, 0.0166, 0.0136, 0.0117, 0.0092, 0.0068, 0.0045, 0.0024, 0.0])
```

1.18 Chemical Geometry (chemicals.molecular_geometry)

This module contains various radius of gyration functions.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.18.1 Lookup Functions

chemicals.molecular_geometry.**RG**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's radius of gyration. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 670 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

RG [float] Radius of gyration, [m]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in RG_methods

Notes

The available sources are as follows:

- 'PSI4_2022A', values computed using the Psi4 version 1.3.2 quantum chemistry software, with initialized positions from rdkit's EmbedMolecule method, the basis set 6-31G** and the method mp2 [1].
- 'CHEMSEP', from the databank included and distributed with the licence notice ChemSep v8.1 pure component data - Copyright (c) Harry Kooijman and Ross Taylor (2018) - http://www.perlfoundation.org/artistic_license_2_0. A small portion of the data is used.

References

[1], [2]

Examples

```
>>> RG(CASRN='64-17-5')
2.225e-10
```

chemicals.molecular_geometry.**RG_methods**(*CASRN*)

Return all methods available to obtain the radius of gyration for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the RG with the given inputs.

See also:

RG

chemicals.molecular_geometry.**RG_all_methods = ('PSI4_2022A', 'CHEMSEP')**
Tuple of method name keys. See the *RG* for the actual references

chemicals.molecular_geometry.**linear**(*CASRN*, *method=None*)
This function handles the retrieval whether or not a chemical is linear. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 300 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

linear [bool] Whether or not the chemical is linear, [-]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in linear_methods

Notes

The available sources are as follows:

- 'PSI4_2022A', values computed using the Psi4 version 1.3.2 quantum chemistry software, with initialized positions from rdkit's EmbedMolecule method, the basis set 6-31G** and the method mp2 [1].

Warning: This function does not yet have a reliable data source.

References

[1]

Examples

```
>>> linear(CASRN='64-17-5')
False
```

chemicals.molecular_geometry.**linear_methods**(*CASRN*)
Return all methods available to obtain whether or not the desired chemical is linear.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the linear with the given inputs.

See also:

linear

chemicals.molecular_geometry.**linear_all_methods = ('PSI4_2022A',)**
Tuple of method name keys. See the *linear* for the actual references

1.19 Support for Numba (chemicals.numba)

Basic module which wraps most of chemicals functions and classes to be compatible with the Numba dynamic Python compiler. Numba is only supported on Python 3, and may require the latest version of Numba. Numba is rapidly evolving, and hopefully in the future it will support more of the functionality of fluids.

Using the numba-accelerated version of *chemicals* is easy; simply call functions and classes from the chemicals.numba namespace. The chemicals.numba module must be imported separately on Python < 3.7; it is not loaded automatically as part of chemicals.

```
>>> import chemicals
>>> import chemicals.numba
>>> chemicals.numba.Antoine(180, A=8.95894, B=510.595, C=-15.95)
702271.05185795
```

There is a delay while the code is compiled when using Numba; the speed is not quite free.

Some parts of *chemicals* are not supported in the Numba interface. Any data lookup function like *Tc()* and *omega_methods()* are not numba compatible. These functions could not be speed up anyway though as they are memory-bound and numba helps with things that are compute-bound.

Today, the list of things known not to work is as follows:

- Everything in *chemicals.identifiers*
- Everything in *chemicals.combustion* (uses dictionaries)
- Everything in *chemicals.elements* (uses dictionaries, periodic_table class, string parsing)
- Everything in *chemicals.critical* except the mixture critical point routines
- In *chemicals.flash_basic*, *flash_Tb_Tc_Pc()* and *flash_ideal()* have not been ported but can be made compatible.
- In *chemicals.dippr*, only *EQ102()* is unsupported as it uses a complex hyp2f1 call.
- In *chemicals.reaction*, *stoichiometric_matrix()* (dictionaries), *balance_stoichiometry()* (scipy functions, fractional numbers).
- In *chemicals.safety*, *LFL_ISO_10156_2017()* (dictionaries)
- The assorted functions collision_integral_Kim_Monroe(), *Henry_pressure_mixture()*, *T_converter()*, *Wilke()*, *Wilke_prefactors()*, *Brokaw() viscosity_converter()*, *CAS_to_int()*, *int_to_CAS()*, *sorted_CAS_key()* may or may not be able to be ported but have not yet been.

All of the regular Numba-compiled functions are built with the *nogil* flag, which means you can use Python's threading mechanism effectively to get the speed of parallel processing.

1.20 Relative Permittivity/Dielectric Constant (chemicals.permittivity)

This module contains various permittivity calculation routines and dataframes of coefficients for correlations.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.20.1 Correlations for Specific Substances

chemicals.permittivity.**permittivity_IAPWS**(*T*, *rho*)

Calculate the relative permittivity of pure water as a function of. temperature and density. Assumes the 1997 IAPWS [1] formulation.

$$\epsilon(\rho, T) = \frac{1 + A + 5B + (9 + 2A + 18B + A^2 + 10AB + 9B^2)^{0.5}}{4(1 - B)}$$

$$A(\rho, T) = \frac{N_A \mu^2 \rho g}{M \epsilon_0 k T}$$

$$B(\rho) = \frac{N_A \alpha \rho}{3 M \epsilon_0}$$

$$g(\delta, \tau) = 1 + \sum_{i=1}^{11} n_i \delta^{I_i} \tau^{J_i} + n_{12} \delta \left(\frac{647.096}{228} \tau^{-1} - 1 \right)^{-1.2}$$

$$\delta = \rho / (322 \text{ kg/m}^3)$$

$$\tau = T / 647.096 \text{K}$$

Parameters

T [float] Temperature of water [K]

rho [float] Mass density of water at T and P [kg/m^3]

Returns

epsilon [float] Relative permittivity of water at T and rho, [-]

Notes

Validity:

273.15 < T < 323.15 K for 0 < P < iceVI melting pressure at T or 1000 MPa, whichever is smaller.

323.15 < T < 873.15 K 0 < p < 600 MPa.

Coefficients and constants (they are optimized away in the function itself):

ih = [1, 1, 1, 2, 3, 3, 4, 5, 6, 7, 10]

jh = [0.25, 1, 2.5, 1.5, 1.5, 2.5, 2, 2, 5, 0.5, 10]

Nh = [0.978224486826, -0.957771379375, 0.237511794148, 0.714692244396, -0.298217036956, -0.108863472196, 0.949327488264E-1, -.980469816509E-2, 0.165167634970E-4, 0.937359795772E-4, -0.12317921872E-9]

polarizability = 1.636E-40

dipole = 6.138E-30

References

[1]

Examples

```
>>> permittivity_IAPWS(373., 958.46)
55.565841872697234
```

```
>>> permittivity_IAPWS(650., 40.31090)
1.2659205723606064
```

chemicals.permittivity.**permittivity_CRC**(*T*, *a*, *b*, *c*, *d*)

Return the relative permittivity (epsilon) of a chemical using a polynomical equation as in [1].

Parameters

a,b,c,d [float] Regressed coefficients.

Notes

The permittivity is given by $\epsilon_r = A + BT + CT^2 + DT^3$

References

[1]

Examples

Calculate the permittivity of 4-Nitroaniline:

```
>>> permittivity_CRC(450., 487, -1.5, 0.00129, 0.)
73.225
```

1.20.2 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.permittivity.**permittivity_data_CRC**

Data from [1] with coefficients fit to a polynomial in terms of temperature in K. $\epsilon_r = A + BT + CT^2 + DT^3$ is the equation, although some chemcials only have a constant value.

```
In [1]: import chemicals

In [2]: chemicals.permittivity.permittivity_data_CRC
Out[2]:
```

(continues on next page)

(continued from previous page)

```
                                                  Chemical  ...   Tmax
CAS                                                         ...
50-70-4                                           D-Glucitol  ...    NaN
50-78-2                              2-(Acetyloxy)benzoic acid  ...  416.0
51-79-6                                        Ethyl carbamate  ...  368.0
54-11-5                                             L-Nicotine  ...  363.0
55-63-0                                       Trinitroglycerol  ...    NaN
...                                                        ...  ...    ...
100295-85-0                          6-Methyl-3-heptanol, ()-  ...  383.0
100296-26-2                          2-Methyl-3-heptanol, ()-  ...  403.0
111675-77-5                          2-Methyl-1-heptanol, ()-  ...  328.0
111767-95-4                          5-Methyl-1-heptanol, ()-  ...  328.0
123434-07-1  2,4,6-Trimethyl-3-heptene (unspecified isomer)  ...    NaN

[1303 rows x 9 columns]
```

1.21 Phase Change Properties (chemicals.phase_change)

This module contains lookup functions for melting and boiling point, heat of fusion, various enthalpy of vaporization estimation routines, and dataframes of fit coefficients.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.21.1 Boiling Point

chemicals.phase_change.**Tb**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's normal boiling point. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available. Function has data for approximately 34000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Tb [float] Boiling temperature, [K]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *Tb_all_methods*.

See also:

Tb_methods

Notes

The available sources are as follows:

- 'CRC_ORG', a compillation of data on organics as published in [1].
- 'CRC_INORG', a compillation of data on inorganic as published in [1].
- 'WEBBOOK', a NIST resource [6] containing mostly experimental and averaged values
- 'WIKIDATA', data from the Wikidata project [3]
- 'COMMON_CHEMISTRY', a project from the CAS [4]
- 'JOBACK', an estimation method for organic substances in [5]
- 'YAWS', a large compillation of data from a variety of sources both experimental and predicted; no data points are sourced in the work of [2].
- 'HEOS', a series of values from the NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids (and other high-precision fundamental equations of state)

References

[1], [2], [3], [4], [5], [6], [7]

Examples

```
>>> Tb('7732-18-5')
373.124
```

chemicals.phase_change.**Tb_methods**(*CASRN*)

Return all methods available to obtain the normal boiling point for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Tb with the given inputs.

See also:

Tb

chemicals.phase_change.**Tb_all_methods = ('HEOS', 'CRC_INORG', 'CRC_ORG', 'COMMON_CHEMISTRY', 'WEBBOOK', 'YAWS', 'WIKIDATA', 'JOBACK')**

Tuple of method name keys. See the *Tbg* for the actual references

1.21.2 Melting Point

chemicals.phase_change.**Tm**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's melting point. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available. Function has data for approximately 83000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Tm [float] Melting temperature, [K]

Other Parameters

method [string, optional] A string for the method name to use, as defined by the vairable *Tm_all_methods*.

See also:

Tm_methods

Notes

The available sources are as follows:

- 'OPEN_NTBKM, a compillation of data on organics as published in [1] as Open Notebook Melting Points; Averaged (median) values were used when multiple points were available. For more information on this invaluable and excellent collection, see http://onswebservices.wikispaces.com/meltingpoint.
- 'CRC_ORG', a compillation of data on organics as published in [2].
- 'CRC_INORG', a compillation of data on inorganic as published in [2].
- 'WEBBOOK', a NIST resource [6] containing mostly experimental and averaged values
- 'WIKIDATA', data from the Wikidata project [3]
- 'COMMON_CHEMISTRY', a project from the CAS [4]
- 'JOBACK', an estimation method for organic substances in [5]

References

[1], [2], [3], [4], [5], [6]

Examples

```
>>> Tm(CASRN='7732-18-5')
273.15
```

chemicals.phase_change.**Tm_methods**(*CASRN*)

Return all methods available to obtain the melting point for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Tm with the given inputs.

See also:

Tm

chemicals.phase_change.**Tm_all_methods = ('OPEN_NTBKM', 'CRC_INORG', 'CRC_ORG', 'COMMON_CHEMISTRY', 'WEBBOOK', 'WIKIDATA', 'JOBACK')**

Tuple of method name keys. See the *Tm* for the actual references

1.21.3 Heat of Fusion

Heat of fusion does not strongly depend on temperature or pressure. This is the standard value, at 1 atm and the normal melting point.

chemicals.phase_change.**Hfus**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's heat of fusion. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 22000 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

Hfus [float] Molar enthalpy of fusion at normal melting point, [J/mol]

Other Parameters

method [string, optional] A string for the method name to use, as defined by the variable, *Hfus_all_methods*.

See also:

Hfus_methods

Notes

The available sources are as follows:

- 'CRC', a compillation of data on organics and inorganics as published in [1].
- 'WEBBOOK', a NIST resource [4] containing mostly experimental and averaged values
- 'WIKIDATA', data from the Wikidata project [2]
- 'JOBACK', an estimation method for organic substances in [3]

References

[1], [2], [3], [4]

Examples

```
>>> Hfus('7732-18-5')
6010.0
```

chemicals.phase_change.**Hfus_methods**(*CASRN*)
Return all methods available to obtain the heat of fusion for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Hfus with the given inputs.

See also:

Hfus

chemicals.phase_change.**Hfus_all_methods = ('CRC', 'WEBBOOK', 'WIKIDATA', 'JOBACK')**
Tuple of method name keys. See the *Hfus* for the actual references

1.21.4 Heat of Vaporization at Tb Correlations

chemicals.phase_change.**Riedel**(*Tb*, *Tc*, *Pc*)
Calculates enthalpy of vaporization at the boiling point, using the Ridel [1] CSP method. Required information are critical temperature and pressure, and boiling point. Equation taken from [2] and [3].

The enthalpy of vaporization is given by:

$$\Delta_{vap}H = 1.093 T_b R \frac{\ln P_c - 1.013}{0.930 - T_{br}}$$

Parameters

Tb [float] Boiling temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

Hvap [float] Enthalpy of vaporization at the normal boiling point, [J/mol]

Notes

This equation has no example calculation in any source. The source has not been verified. It is equation 4-144 in Perry's. Perry's also claims that errors seldom surpass 5%.

[2] is the source of example work here, showing a calculation at 0.0% error.

Internal units of pressure are bar.

References

[1], [2], [3]

Examples

Pyridine, 0.0% err vs. exp: 35090 J/mol; from Poling [2].

```
>>> Riedel(388.4, 620.0, 56.3E5)
35089.80179000598
```

chemicals.phase_change.**Chen**(*Tb*, *Tc*, *Pc*)

Calculates enthalpy of vaporization using the Chen [1] correlation and a chemical's critical temperature, pressure and boiling point.

The enthalpy of vaporization is given by:

$$\Delta H_{vb} = RT_b \frac{3.978T_r - 3.958 + 1.555 \ln P_c}{1.07 - T_r}$$

Parameters

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

The formulation presented in the original article is similar, but uses units of atm and calorie instead. The form in [2] has adjusted for this. A method for estimating enthalpy of vaporization at other conditions has also been developed, but the article is unclear on its implementation. Based on the Pitzer correlation.

Internal units: bar and K

References

[1], [2]

Examples

Same problem as in Perry's examples.

```
>>> Chen(294.0, 466.0, 5.55E6)
26705.902558030946
```

chemicals.phase_change.**Liu**(*Tb*, *Tc*, *Pc*)

Calculates enthalpy of vaporization at the normal boiling point using the Liu [1] correlation, and a chemical's critical temperature, pressure and boiling point.

The enthalpy of vaporization is given by:

$$\Delta H_{vap} = RT_b \left[\frac{T_b}{220}\right]^{0.0627} \frac{(1-T_{br})^{0.38}\ln(P_c/P_A)}{1-T_{br}+0.38T_{br}\ln T_{br}}$$

Parameters

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

This formulation can be adjusted for lower boiling points, due to the use of a rationalized pressure relationship. The formulation is taken from the original article.

A correction for alcohols and organic acids based on carbon number, which only modifies the boiling point, is available but not implemented.

No sample calculations are available in the article.

Internal units: Pa and K

References

[1]

Examples

Same problem as in Perry's examples

```
>>> Liu(294.0, 466.0, 5.55E6)
26378.575260517395
```

chemicals.phase_change.**Vetere**(*Tb*, *Tc*, *Pc*, *F=1.0*)

Calculates enthalpy of vaporization at the boiling point, using the Vetere [1] CSP method. Required information are critical temperature and pressure, and boiling point. Equation taken from [2].

The enthalpy of vaporization is given by:

$$\frac{\Delta H_{vap}}{RT_b} = \frac{\tau_b^{0.38}\left[\ln P_c - 0.513 + \frac{0.5066}{P_c T_{br}^2}\right]}{\tau_b + F(1-\tau_b^{0.38})\ln T_{br}}$$

Parameters

Tb [float] Boiling temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

F [float, optional] Constant for a fluid, [-]

Returns

Hvap [float] Enthalpy of vaporization at the boiling point, [J/mol]

Notes

The equation cannot be found in the original source. It is believed that a second article is its source, or that DIPPR staff have altered the formulation.

Internal units of pressure are bar.

References

[1], [2]

Examples

Example as in [2], p2-487; exp: 25.73

```
>>> Vetere(294.0, 466.0, 5.55E6)
26363.43895706672
```

1.21.5 Heat of Vaporization at T Correlations

chemicals.phase_change.**Pitzer**(*T*, *Tc*, *omega*)

Calculates enthalpy of vaporization at arbitrary temperatures using a fit by [2] to the work of Pitzer [1]; requires a chemical's critical temperature and acentric factor.

The enthalpy of vaporization is given by:

$$\frac{\Delta_{vap}H}{RT_c} = 7.08(1-T_r)^{0.354} + 10.95\omega(1-T_r)^{0.456}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor [-]

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

This equation is listed in [3], page 2-487 as method #2 for estimating Hvap. This cites [2].

The recommended range is 0.6 to 1 Tr. Users should expect up to 5% error. This function converges to zero at *Tc*. If Tc is larger than T, 0 is returned as the model would return complex numbers.

The original article has been reviewed and found to have a set of tabulated values which could be used instead of the fit function to provide additional accuracy.

References

[1], [2], [3]

Examples

Example as in [3], p2-487; exp: 37.51 kJ/mol

```
>>> Pitzer(452, 645.6, 0.35017)
36696.749078320056
```

chemicals.phase_change.**SMK**(*T*, *Tc*, *omega*)

Calculates enthalpy of vaporization at arbitrary temperatures using a the work of [1]; requires a chemical's critical temperature and acentric factor.

The enthalpy of vaporization is given by:

$$\frac{\Delta H_{vap}}{RT_c} = \left(\frac{\Delta H_{vap}}{RT_c}\right)^{(R1)} + \left(\frac{\omega-\omega^{(R1)}}{\omega^{(R2)}-\omega^{(R1)}}\right)\left[\left(\frac{\Delta H_{vap}}{RT_c}\right)^{(R2)} - \left(\frac{\Delta H_{vap}}{RT_c}\right)^{(R1)}\right]$$

$$\left(\frac{\Delta H_{vap}}{RT_c}\right)^{(R1)} = 6.537\tau^{1/3} - 2.467\tau^{5/6} - 77.251\tau^{1.208} + 59.634\tau + 36.009\tau^2 - 14.606\tau^3$$

$$\left(\frac{\Delta H_{vap}}{RT_c}\right)^{(R2)} - \left(\frac{\Delta H_{vap}}{RT_c}\right)^{(R1)} = -0.133\tau^{1/3} - 28.215\tau^{5/6} - 82.958\tau^{1.208} + 99.00\tau + 19.105\tau^2 - 2.796\tau^3$$

$$\tau = 1 - T/T_c$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor [-]

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

The original article has been reviewed and found to have coefficients with slightly more precision. Additionally, the form of the equation is slightly different, but numerically equivalent.

The refence fluids are:

ω_0 = benzene = 0.212

ω_1 = carbazole = 0.461

A sample problem in the article has been verified. The numerical result presented by the author requires high numerical accuracy to obtain.

This function converges to zero at *Tc*. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Problem in [1]:

```
>>> SMK(553.15, 751.35, 0.302)
39866.18999046229
```

chemicals.phase_change.**MK**(*T*, *Tc*, *omega*)

Calculates enthalpy of vaporization at arbitrary temperatures using a the work of [1]; requires a chemical's critical temperature and acentric factor.

The enthalpy of vaporization is given by:

$$\Delta H_{vap} = \Delta H_{vap}^{(0)} + \omega \Delta H_{vap}^{(1)} + \omega^2 \Delta H_{vap}^{(2)}$$

$$\frac{\Delta H_{vap}^{(i)}}{RT_c} = b^{(j)}\tau^{1/3} + b_2^{(j)}\tau^{5/6} + b_3^{(j)}\tau^{1.2083} + b_4^{(j)}\tau + b_5^{(j)}\tau^2 + b_6^{(j)}\tau^3$$

$$\tau = 1 - T/T_c$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor [-]

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

The original article has been reviewed. A total of 18 coefficients are used:

WARNING: The correlation has been implemented as described in the article, but its results seem different and with some error. Its results match with other functions however.

Has poor behavior for low-temperature use. This function converges to zero at *Tc*. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

Problem in article for SMK function.

```
>>> MK(553.15, 751.35, 0.302)
38728.00667307733
```

chemicals.phase_change.**Velasco**(*T*, *Tc*, *omega*)

Calculates enthalpy of vaporization at arbitrary temperatures using a the work of [1]; requires a chemical's critical temperature and acentric factor.

The enthalpy of vaporization is given by:

$$\Delta_{vap}H = RT_c(7.2729 + 10.4962\omega + 0.6061\omega^2)(1 - T_r)^{0.38}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

omega [float] Acentric factor [-]

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

The original article has been reviewed. It is regressed from enthalpy of vaporization values at 0.7Tr, from 121 fluids in REFPROP 9.1. A value in the article was read to be similar, but slightly too low from that calculated here. This function converges to zero at *Tc*. If Tc is larger than T, 0 is returned as the model would return complex numbers.

References

[1]

Examples

From graph, in [1] for perfluoro-n-heptane.

```
>>> Velasco(333.2, 476.0, 0.5559)
33299.428636069264
```

chemicals.phase_change.**Clapeyron**(*T*, *Tc*, *Pc*, *dZ=1*, *Psat=101325*)

Calculates enthalpy of vaporization at arbitrary temperatures using the Clapeyron equation.

The enthalpy of vaporization is given by:

$$\Delta H_{vap} = RT\Delta Z\frac{\ln(P_c/Psat)}{(1-T_r)}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

dZ [float] Change in compressibility factor between liquid and gas, []

Psat [float] Saturation pressure of fluid [Pa], optional

Returns

Hvap [float] Enthalpy of vaporization, [J/mol]

Notes

No original source is available for this equation. [1] claims this equation overpredicts enthalpy by several percent. Under Tr = 0.8, dZ = 1 is a reasonable assumption. This equation is most accurate at the normal boiling point.

Internal units are bar.

WARNING: I believe it possible that the adjustment for pressure may be incorrect

References

[1]

Examples

Problem from Perry's examples.

```
>>> Clapeyron(T=294.0, Tc=466.0, Pc=5.55E6)
26512.36357131963
```

chemicals.phase_change.**Watson**(*T*, *Hvap_ref*, *T_ref*, *Tc*, *exponent=0.38*)

Calculates enthalpy of vaporization of a chemical at a temperature using the known heat of vaporization at another temperature according to the Watson [1] [2] correlation. This is an application of the corresponding-states principle, with an emperical temperature dependence.

$$\frac{\Delta H_{vap}^{T1}}{\Delta H_{vap}^{T2}} = \left(\frac{1 - T_{r,1}}{1 - T_{r,2}}\right)^{0.38}$$

Parameters

T [float] Temperature for which to calculate heat of vaporization, [K]

Hvap_ref [float] Enthalpy of vaporization at the known temperature point, [J/mol]

T_ref [float] Reference temperature; ideally as close to *T* as posible, [K]

Tc [float] Critical temperature of fluid [K]

exponent [float, optional] A fit exponent can optionally be used instead of the Watson 0.38 exponent, [-]

Returns

Hvap [float] Enthalpy of vaporization at *T*, [J/mol]

References

[1], [2]

Examples

Predict the enthalpy of vaporization of water at 320 K from a point at 300 K:

```
>>> Watson(T=320, Hvap_ref=43908, T_ref=300.0, Tc=647.14)
42928.990094915454
```

The error is 0.38% compared to the correct value of 43048 J/mol.

If the provided temperature is above the critical point, zero is returned.

chemicals.phase_change.**Watson_n**(*T1*, *T2*, *Hvap1*, *Hvap2*, *Tc*)

Calculates the Watson heat of vaporizaton extrapolation exponent given two known heats of vaporization.

$$n = \left[\frac{\ln\left(\frac{Hvap_1}{Hvap_2}\right)}{\ln\left(\frac{T_1 - T_c}{T_2 - T_c}\right)}\right]$$

Parameters

T1 [float] Temperature of first heat of vaporization point, [K]

T2 [float] Temperature of second heat of vaporization point, [K]

Hvap1 [float] Enthalpy of vaporization at the first known temperature point, [J/mol]

Hvap2 [float] Enthalpy of vaporization at the second known temperature point, [J/mol]

Tc [float] Critical temperature of fluid [K]

Returns

exponent [float] A fit exponent that can be used instead of the Watson 0.38 exponent, [-]

Notes

This can be useful for extrapolating when a correlation does not reach the critical point.

Examples

```
>>> Watson_n(T1=320, T2=300, Hvap1=42928.990094915454, Hvap2=43908, Tc=647.14)
0.380000000000
```

1.21.6 Heat of Vaporization at T Model Equations

chemicals.phase_change.**Alibakhshi**(*T*, *Tc*, *C*)

Calculates enthalpy of vaporization of a chemical at a temperature using a theoretically-derived single-coefficient fit equation developed in [1]. This model falls apart at ~0.8 Tc.

$$\Delta H_{vap} = (4.5\pi N_A)^{1/3.} 4.2 \times 10^{-7}(T_c - 6) - 0.5RT\ln(T) + CT$$

Parameters

T [float] Temperature for which to calculate heat of vaporization, [K]

Tc [float] Critical temperature of fluid [K]

C [float] Alibakhshi fit coefficient, [J/mol/K]

Returns

Hvap [float] Enthalpy of vaporization at *T*, [J/mol]

Notes

The authors of [1] evaluated their model on 1890 compounds for a temperature range of 50 K under *Tb* to 100 K below *Tc*, and obtained an average absolute relative error of 4.5%.

References

[1]

Examples

Predict the enthalpy of vaporization of water at 320 K:

```
>>> Alibakhshi(T=320.0, Tc=647.14, C=-16.7171)
41961.30490225752
```

The error is 2.5% compared to the correct value of 43048 J/mol.

chemicals.phase_change.**PPDS12**(*T*, *Tc*, *A*, *B*, *C*, *D*, *E*)

Calculate the enthalpy of vaporization of a fluid using the 5-term power fit developed by the PPDS and named PPDS equation 12.

$$H_{vap} = RT_c\left(A\tau^{1/3} + B\tau^{2/3} + C\tau + D\tau^2 + E\tau^6\right)$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [-]

C [float] Coefficient, [-]

D [float] Coefficient, [-]

E [float] Coefficient, [-]

Returns

Hvap [float] Enthalpy of vaporization at *T*, [J/mol]

Notes

Coefficients can be found in [1], but no other source for these coefficients has been found.

References

[1], [2]

Examples

Example from [1]:

```
>>> PPDS12(300.0, 591.75, 4.60584, 13.97224, -10.592315, 2.120205, 4.277128)
37948.76862035925
```

Example from [2] for benzene; note the coefficients from [2] predict enthalpy of vaporization in kJ/mol, so the output must be adjusted. The same effect can be obtained by multiplying each of the coefficients by 1000.

```
>>> 1000.0*PPDS12(300.0, 562.05, 0.00171484, 0.0258604, -0.0243564, 0.00740881, 0.
↪00680068)
33662.4258030
```

1.21.7 Heat of Sublimation

No specific correlation is provided. This value is fairly strongly temperature dependent; the dependency comes almost entirely from the vaporization enthalpy's dependence. To calculate heat of sublimation at any temperature, use the equation $H_{sub} = H_{fus} + H_{vap}$.

1.21.8 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.phase_change.**phase_change_data_Perrys2_150**
: A collection of 344 coefficient sets from the DIPPR database published openly in [1]. Provides temperature limits for all its fluids. See *chemicals.dippr.EQ106* for the model equation.

chemicals.phase_change.**phase_change_data_VDI_PPDS_4**
: Coefficients for a equation form developed by the PPDS, published openly in [2]. Extrapolates poorly at low temperatures. See *PPDS12* for the model equation.

chemicals.phase_change.**phase_change_data_Alibakhshi_Cs**
: One-constant limited temperature range regression coefficients presented in [3], with constants for ~2000 chemicals from the DIPPR database. Valid up to 100 K below the critical point, and 50 K under the boiling point. See *Alibakhshi* for the model equation.

The structure of each dataframe is shown below:

```
In [1]: import chemicals

In [2]: chemicals.phase_change.phase_change_data_Perrys2_150
Out[2]:
                          Chemical      Tc       C1  ...       C4    Tmin    Tmax
CAS                                                  ...
50-00-0               Formaldehyde  408.00  30760.0  ...  0.00000  181.15  408.00
55-21-0                  Benzamide  824.00  87809.0  ... -0.14162  403.00  824.00
56-23-5        Carbon tetrachloride  556.35  43252.0  ...  0.00000  250.33  556.35
57-55-6        1,2-Propylene glycol  626.00  80700.0  ...  0.00000  213.15  626.00
60-29-7              Diethyl ether  466.70  40600.0  ...  0.00000  156.85  466.70
...                            ...     ...      ...  ...      ...     ...     ...
10028-15-6                   Ozone  261.00  18587.0  ...  0.00000   80.15  261.00
10035-10-6         Hydrogen bromide  363.15  24850.0  ...  0.00000  185.15  363.15
10102-43-9             Nitric oxide  180.15  21310.0  ...  0.00000  109.50  180.15
13511-13-2    Propenylcyclohexene    636.00  58866.0  ...  0.00000  199.00  636.00
132259-10-0                   Air    132.45   8474.0  ...  0.00000   59.15  132.45

[344 rows x 8 columns]

In [3]: chemicals.phase_change.phase_change_data_VDI_PPDS_4
Out[3]:
                        Chemical      MW  ...          D           E
CAS                                       ...
50-00-0               Formaldehyde   30.03  ...  -4.856937   11.036836
56-23-5        Carbon tetrachloride  153.82  ...  -0.172679    3.053272
56-81-5                  Glycerol   92.09  ...   2.052518  -13.771300
60-29-7              Diethyl ether   74.12  ...  -0.175016    3.557340
62-53-3                    Aniline   93.13  ...  -1.656520    3.263408
```

(continues on next page)

(continued from previous page)

```
...                            ...     ...  ...        ...         ...
10097-32-2                 Bromine  159.82  ...  -0.025698   -0.197360
10102-43-9             Nitric oxide   30.01  ...  -5.159373   97.203137
10102-44-0        Nitrogen dioxide   46.01  ...  10.653997   68.680656
10544-72-6     Dinitrogentetroxide   92.01  ...  -1.535179  102.679020
132259-10-0                    Air   28.96  ...  -8.064787   14.645081

[272 rows x 8 columns]

In [4]: chemicals.phase_change.phase_change_data_Alibakhshi_Cs
Out[4]:
                         Chemical         C
CAS
50-00-0               formaldehyde  -26.7916
50-21-5                lactic acid   30.5238
50-70-4                   sorbitol   89.1371
50-78-2       acetylsalicylic acid   15.9121
50-81-7               ascorbic acid  102.2858
...                             ...      ...
7642-10-6            cis-3-heptene  -17.8032
7719-09-7         thionyl chloride  -31.2745
7719-12-2  phosphorus trichloride  -27.0024
7783-06-4         hydrogen sulfide  -37.3259
7783-07-5        hydrogen selenide  -38.5320

[1890 rows x 2 columns]
```

1.22 Rachford-Rice Equation Solvers (chemicals.rachford_rice)

This module contains functions for solving the Rachford-Rice Equation. This is used to solve ideal flashes, and is the inner loop of the sequential-substitution flash algorithm. It is not used by full newton-algorithms. The sequential-substitution is normally recommended because it does not suffer from the ~N^3 behavior of solving a matrix.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.22.1 Two Phase - Interface

chemicals.rachford_rice.**flash_inner_loop**(*zs*, *Ks*, *method=None*, *guess=None*, *check=False*)

This function handles the solution of the inner loop of a flash calculation, solving for liquid and gas mole fractions and vapor fraction based on specified overall mole fractions and K values. As K values are weak functions of composition, this should be called repeatedly by an outer loop. Will automatically select an algorithm to use if no method is provided. Should always provide a solution.

The automatic algorithm selection will try an analytical solution, and use the Rachford-Rice method if there are 6 or more components in the mixture.

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

guess [float, optional] Optional initial guess for vapor fraction, [-]

check [bool, optional] Whether or not to check the K values to ensure a positive-composition solution exists, [-]

Returns

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Other Parameters

method [string, optional] The method name to use. Accepted methods are 'Analytical', 'Rachford-Rice (Secant)', 'Rachford-Rice (Newton-Raphson)', 'Rachford-Rice (Halley)', 'Rachford-Rice (NumPy)', 'Leibovici and Nichita 2', 'Rachford-Rice (polynomial)', and 'Li-Johns-Ahmadi'. All valid values are also held in the list *flash_inner_loop_methods*.

Notes

A total of eight methods are available for this function. They are:

- 'Analytical', an exact solution derived with SymPy, applicable only only to mixtures of two, three, or four components
- 'Rachford-Rice (Secant)', 'Rachford-Rice (Newton-Raphson)', 'Rachford-Rice (Halley)', or 'Rachford-Rice (NumPy)', which numerically solves an objective function described in *Rachford_Rice_solution*.
- 'Leibovici and Nichita 2', a transformation of the RR equation described in *Rachford_Rice_solution_LN2*.
- 'Li-Johns-Ahmadi', which numerically solves an objective function described in *Li_Johns_Ahmadi_solution*.
- 'Leibovici and Neoschil', which numerically solves an objective function described in *Rachford_Rice_solution_Leibovici_Neoschil*.

Examples

```
>>> flash_inner_loop(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
(0.6907302627738, [0.3394086969663, 0.3650560590371, 0.29553524399648], [0.
↪571903654388, 0.27087159580558, 0.1572247498061])
```

chemicals.rachford_rice.**flash_inner_loop_methods**(*N*)

Return all methods able to solve the Rachford-Rice equation for the specified number of components.

Parameters

N [int] Number of components, [-]

Returns

methods [list[str]] Methods which can be used to solve the Rachford-rice equation

See also:

flash_inner_loop

chemicals.rachford_rice.**flash_inner_loop_all_methods = ('Analytical', 'Rachford-Rice (Secant)', 'Rachford-Rice (Newton-Raphson)', 'Rachford-Rice (Halley)', 'Rachford-Rice (NumPy)', 'Li-Johns-Ahmadi', 'Rachford-Rice (polynomial)', 'Leibovici and Nichita 2', 'Leibovici and Neoschil')**

Tuple of method name keys. See the *flash_inner_loop* for the actual references

1.22.2 Two Phase - Implementations

chemicals.rachford_rice.**Rachford_Rice_solution**(*zs*, *Ks*, *fprime=False*, *fprime2=False*, *guess=None*)

Solves the objective function of the Rachford-Rice flash equation [1]. Uses the method proposed in [2] to obtain an initial guess.

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

fprime [bool, optional] Whether or not to use the first derivative of the objective function in the solver (Newton-Raphson is used) or not (secant is used), [-]

fprime2 [bool, optional] Whether or not to use the second derivative of the objective function in the solver (parabolic Halley`s method is used if True) or not, [-]

guess [float, optional] Optional initial guess for vapor fraction, [-]

Returns

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

The initial guess is the average of the following, as described in [2].

$$\left(\frac{V}{F}\right)_{min} = \frac{(K_{max} - K_{min})z_{of\ K_{max}} - (1 - K_{min})}{(1 - K_{min})(K_{max} - 1)}$$

$$\left(\frac{V}{F}\right)_{max} = \frac{1}{1 - K_{min}}$$

Another algorithm for determining the range of the correct solution is given in [3]; [2] provides a narrower range however. For both cases, each guess should be limited to be between 0 and 1 as they are often negative or larger than 1.

$$\left(\frac{V}{F}\right)_{min} = \frac{1}{1 - K_{max}}$$

$$\left(\frac{V}{F}\right)_{max} = \frac{1}{1 - K_{min}}$$

If the *newton* method does not converge, a bisection method (brenth) is used instead. However, it is somewhat slower, especially as newton will attempt 50 iterations before giving up.

In all benchmarks attempted, secant method provides better performance than Newton-Raphson or parabolic Halley`s method. This may not be generally true; but it is for Python and SciPy's implementation. They are implemented for benchmarking purposes.

The first and second derivatives are:

$$\frac{d\ \text{obj}}{d\frac{V}{F}} = \sum_i \frac{-z_i(K_i - 1)^2}{(1 + \frac{V}{F}(K_i - 1))^2}$$

$$\frac{d^2\ \text{obj}}{d(\frac{V}{F})^2} = \sum_i \frac{2z_i(K_i - 1)^3}{(1 + \frac{V}{F}(K_i - 1))^3}$$

References

[1], [2], [3]

Examples

```
>>> Rachford_Rice_solution(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
(0.6907302627738544, [0.33940869696634357, 0.3650560590371706, 0.2955352439964858],␣
↪[0.5719036543882889, 0.27087159580558057, 0.15722474980613044])
```

chemicals.rachford_rice.**Rachford_Rice_solution_LN2**(*zs*, *Ks*, *guess=None*)

Solves the a objective function for the Rachford-Rice flash equation according to the Leibovici and Nichita (2010) transformation (method 2). This transformation makes the only zero of the function be the desired one. Consequently, higher-order methods may be used to solve this equation. Halley's (second derivative) method is found to be the best; typically needing ~50% fewer iterations than the RR formulation with Secant method.

$$H(y) = \sum_i^n \frac{z_i}{\lambda - c_i} = 0$$

$$\lambda = c_k + \frac{c_{k+1} - c_k}{1 + e^{-y}}$$

$$c_i = \frac{1}{1-K_i}$$

$$c_k = \left(\frac{V}{F}\right)_{min}$$

$$c_{k+1} = \left(\frac{V}{F}\right)_{max}$$

Note the two different uses of c in the above equation, confusingly given in [1]. *lambda* is the vapor fraction.

Once the equation has been solved for y, the vapor fraction can be calculated outside the solver.

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

guess [float, optional] Optional initial guess for vapor fraction, [-]

Returns

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

The initial guess is the average of the following, as described in [2].

$$\left(\frac{V}{F}\right)_{min} = \frac{(K_{max}-K_{min})z_{of\ K_{max}} - (1-K_{min})}{(1-K_{min})(K_{max}-1)}$$

$$\left(\frac{V}{F}\right)_{max} = \frac{1}{1-K_{min}}$$

The first and second derivatives are derived with sympy as follows:

```
>>> from sympy import *
>>> VF_min, VF_max, ai, ci, y = symbols('VF_min, VF_max, ai, ci, y')
>>> V_over_F = (VF_min + (VF_max - VF_min)/(1 + exp(-y)))
>>> F = ai/(V_over_F - ci)
>>> terms = [F, diff(F, y), diff(F, y, 2)]
>>> cse(terms, optimizations='basic')
```

Some helpful information about this transformation can also be found in [3].

References

[1], [2], [3]

Examples

```
>>> Rachford_Rice_solution_LN2(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
(0.6907302627738, [0.3394086969663, 0.3650560590371, 0.29553524399648], [0.
↪571903654388, 0.27087159580558, 0.1572247498061])
```

chemicals.rachford_rice.**Li_Johns_Ahmadi_solution**(*zs*, *Ks*, *guess=None*)

Solves the objective function of the Li-Johns-Ahmadi flash equation. Uses the method proposed in [1] to obtain an initial guess.

$$0 = 1 + \left(\frac{K_{max} - K_{min}}{K_{min} - 1}\right) x_{max} + \sum_{i=2}^{n-1} \frac{K_i - K_{min}}{K_{min} - 1} \left[\frac{z_i(K_{max} - 1)x_{max}}{(K_i - 1)z_{max} + (K_{max} - K_i)x_{max}}\right]$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

Returns

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

The initial guess is the average of the following, as described in [1]. Each guess should be limited to be between 0 and 1 as they are often negative or larger than 1. *max* refers to the corresponding mole fractions for the species with the largest K value.

$$\left(\frac{1 - K_{min}}{K_{max} - K_{min}}\right) z_{max} \leq x_{max} \leq \left(\frac{1 - K_{min}}{K_{max} - K_{min}}\right)$$

If the *newton* method does not converge, a bisection method (brenth) is used instead. However, it is somewhat slower, especially as newton will attempt 50 iterations before giving up.

This method does not work for problems of only two components. K values are sorted internally. Has not been found to be quicker than the Rachford-Rice equation.

References

[1]

Examples

```
>>> Li_Johns_Ahmadi_solution(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
(0.6907302627738544, [0.33940869696634357, 0.3650560590371706, 0.2955352439964858],␣
↪[0.5719036543882889, 0.27087159580558057, 0.15722474980613044])
```

chemicals.rachford_rice.**Rachford_Rice_solution_Leibovici_Neoschil**(*zs*, *Ks*, *guess=None*)

Solves the objective function of the Rachford-Rice flash equation as modified by Leibovici and Neoschil. This modification helps convergence near the vapor fraction boundaries only; it slows convergence in other regions.

$$\left(\frac{V}{F} - \alpha_L\right)\left(\alpha_R - \frac{V}{F}\right)\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

$$\alpha_L = -\frac{1}{K_{max} - 1}$$

$$\alpha_R = \frac{1}{1 - K_{min}}$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

guess [float, optional] Optional initial guess for vapor fraction, [-]

Returns

L_over_F [float] Liquid fraction solution [-]

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

The initial guess is the average of the following.

$$\left(\frac{V}{F}\right)_{min} = \frac{(K_{max} - K_{min})z_{of\ K_{max}} - (1 - K_{min})}{(1 - K_{min})(K_{max} - 1)}$$

$$\left(\frac{V}{F}\right)_{max} = \frac{1}{1 - K_{min}}$$

References

[1]

Examples

```
>>> Rachford_Rice_solution_Leibovici_Neoschil(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742,␣
↪0.532])
(0.3092697372261, 0.69073026277385, [0.339408696966343, 0.36505605903717, 0.
↪29553524399648], [0.57190365438828, 0.270871595805580, 0.157224749806130])
```

chemicals.rachford_rice.**Rachford_Rice_solution_polynomial**(*zs*, *Ks*)

Solves the Rachford-Rice equation by transforming it into a polynomial, and then either analytically calculating the roots, or, using the known range the correct root is in, numerically solving for the correct polynomial root. The analytical solutions are used for N from 2 to 4.

Uses the method proposed in [2] to obtain an initial guess when solving the polynomial for the root numerically.

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

Warning: : Using this function with more than 20 components is likely to crash Python! This model does not work well with many components!

This method, developed first in [3] and expanded in [1], is clever but of little use for large numbers of components.

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

Returns

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

This approach has mostly been ignored by academia, despite some of its advantages.

The initial guess is the average of the following, as described in [2].

$$\left(\frac{V}{F}\right)_{min} = \frac{(K_{max} - K_{min})z_{of\ K_{max}} - (1 - K_{min})}{(1 - K_{min})(K_{max} - 1)}$$

$$\left(\frac{V}{F}\right)_{max} = \frac{1}{1 - K_{min}}$$

If the *newton* method does not converge, a bisection method (brenth) is used instead. However, it is somewhat slower, especially as newton will attempt 50 iterations before giving up.

This method could be speed up somewhat for N <= 4; the checks for the vapor fraction range are not really needed.

References

[1], [2], [3]

Examples

```
>>> Rachford_Rice_solution_polynomial(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
(0.6907302627738543, [0.33940869696634357, 0.3650560590371706, 0.2955352439964858],␣
↪[0.5719036543882889, 0.27087159580558057, 0.15722474980613044])
```

1.22.3 Two Phase - High-Precision Implementations

chemicals.rachford_rice.**Rachford_Rice_solution_mpmath**(*zs*, *Ks*, *dps=200*, *tol=1e-100*)

Solves the the Rachford-Rice flash equation using numerical root-finding to a high precision using the *mpmath* library.

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

dps [int, optional] Number of decimal places to use in the intermediate values of the calculation, [-]

tol [float, optional] The tolerance of the solver used in *mpmath*, [-]

Returns

L_over_F [float] Liquid fraction solution [-]

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

This function is written solely for development purposes with the aim of returning bit-accurate solutions.

Note that the liquid fraction is also returned; it is insufficient to compute it as $\frac{L}{F} = 1 - \frac{V}{F}$.

Examples

```
>>> Rachford_Rice_solution_mpmath(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
(0.3092697372261456, 0.6907302627738544, [0.33940869696634357, 0.3650560590371706,␣
↪0.29553524399648584], [0.5719036543882889, 0.27087159580558057, 0.
↪15722474980613046])
>>> Rachford_Rice_solution_mpmath(zs=[0.999999999999, 1e-12], Ks=[2.0, 1e-12])
(1e-12, 0.999999999999, [0.49999999999975003, 0.50000000000025], [0.
↪9999999999995001, 5.0000000000025e-13])
```

chemicals.rachford_rice.**Rachford_Rice_solution_binary_dd**(*zs*, *Ks*)

Solves the the Rachford-Rice flash equation for a binary system using double-double math. This increases the range in which the calculation can be performed accurately but does not totally eliminate error.

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

The analytical solution for a binary system is:

$$\frac{V}{F} = \frac{-K_0 z_0 - K_1 z_1 + z_0 + z_1}{K_0 K_1 z_0 + K_0 K_1 z_1 - K_0 z_0 - K_0 z_1 - K_1 z_0 - K_1 z_1 + z_0 + z_1}$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

Returns

L_over_F [float] Liquid fraction solution [-]

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Examples

This system with large volatility difference and a trace of a component shows a correct calculation. Try it out with other solvers for bad results!

```
>>> Rachford_Rice_solution_binary_dd(zs=[1E-27, 1.0], Ks=[1000000000000,0.1])
(1.000000000001, -1.0000000000009988e-12, [9.0000000000009e-13, 0.9999999999991],␣
↪[0.90000000000009, 0.09999999999991001])
```

Note the limitations of this solver can be explored by comparing against *Rachford_Rice_solution_mpmath*. For example, with *z0* of 1e-28 in the above example error creeps back in.

chemicals.rachford_rice.**Rachford_Rice_solution_Leibovici_Neoschil_dd**(*zs*, *Ks*, *guess=None*)

Solves the objective function of the Rachford-Rice flash equation as modified by Leibovici and Neoschil, using double-double precision math for maximum accuracy. For most cases, this function will return bit-for-bit accurate results; but there are pathological inputs where error still occurs.

$$\left(\frac{V}{F} - \alpha_L\right)\left(\alpha_R - \frac{V}{F}\right)\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

$$\alpha_L = -\frac{1}{K_{max} - 1}$$

$$\alpha_R = \frac{1}{1 - K_{min}}$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

guess [float, optional] Optional initial guess for vapor fraction, [-]

Returns

L_over_F [float] Liquid fraction solution [-]

V_over_F [float] Vapor fraction solution [-]

xs [list[float]] Mole fractions of each species in the liquid phase, [-]

ys [list[float]] Mole fractions of each species in the vapor phase, [-]

Notes

The initial guess is the average of the following.

$$\left(\frac{V}{F}\right)_{min} = \frac{(K_{max} - K_{min})z_{of\ K_{max}} - (1 - K_{min})}{(1 - K_{min})(K_{max} - 1)}$$

$$\left(\frac{V}{F}\right)_{max} = \frac{1}{1 - K_{min}}$$

References

[1]

Examples

```
>>> Rachford_Rice_solution_Leibovici_Neoschil_dd(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.
↪742, 0.532])
(0.3092697372261, 0.69073026277385, [0.339408696966343, 0.36505605903717, 0.
↪29553524399648], [0.57190365438828, 0.270871595805580, 0.157224749806130])
```

1.22.4 Three Phase

chemicals.rachford_rice.**Rachford_Rice_solution2**(*ns*, *Ks_y*, *Ks_z*, *beta_y=0.5*, *beta_z=1e-06*)

Solves the two objective functions of the Rachford-Rice flash equation for a three-phase system. Initial guesses are required for both phase fractions, *beta_y* and *beta_z*. The Newton method is used, with an analytical Jacobian.

$$F_0 = \sum_i \frac{z_i(K_y - 1)}{1 + \beta_y(K_y - 1) + \beta_z(K_z - 1)} = 0$$

$$F_1 = \sum_i \frac{z_i(K_z - 1)}{1 + \beta_y(K_y - 1) + \beta_z(K_z - 1)} = 0$$

Parameters

ns [list[float]] Overall mole fractions of all species (would be *zs* except that is conventially used for one of the three phases), [-]

Ks_y [list[float]] Equilibrium K-values of *y* phase to *x* phase, [-]

Ks_z [list[float]] Equilibrium K-values of *z* phase to *x* phase, [-]

beta_y [float, optional] Initial guess for *y* phase (between 0 and 1), [-]

beta_z [float, optional] Initial guess for *z* phase (between 0 and 1), [-]

Returns

beta_y [float] Phase fraction of *y* phase, [-]

beta_z [float] Phase fraction of *z* phase, [-]

xs [list[float]] Mole fractions of each species in the *x* phase, [-]

ys [list[float]] Mole fractions of each species in the *y* phase, [-]

zs [list[float]] Mole fractions of each species in the *z* phase, [-]

Notes

The elements of the Jacobian are calculated as follows:

$$\frac{\partial F_0}{\partial \beta_y} = \sum_i \frac{-z_i(K_y-1)^2}{(1+\beta_y(K_y-1)+\beta_z(K_z-1))^2}$$

$$\frac{\partial F_1}{\partial \beta_z} = \sum_i \frac{-z_i(K_z-1)^2}{(1+\beta_y(K_y-1)+\beta_z(K_z-1))^2}$$

$$\frac{\partial F_1}{\partial \beta_y} = \sum_i \frac{\partial F_0}{\partial \beta_z} = \frac{-z_i(K_z-1)(K_y-1)}{(1+\beta_y(K_y-1)+\beta_z(K_z-1))^2}$$

In general, the solution which Newton's method converges to may not be the desired one, so further constraints are required.

Okuno's method in [1] provides a polygonal region where the correct answer lies. It has not been implemented.

The Leibovici and Neoschil method [4] provides a method to compute/update the damping parameter, which is suposed to ensure convergence. It claims to be able to calculate the maximum damping factor for Newton's method, if it tries to go out of bounds.

A custom region which is believed to be the same as that of Okuno is implemented instead - the region which ensures positive compositions for all compounds in all phases, but does not restrict the phase fractions to be between 0 and 1 or even positive.

With the convergence restraint, it is believed if a solution lies within (0, 1) for both variables, the correct solution will be converged to so long as the initial guesses are within the correct region.

Some helpful information has also been found in [2] and [3].

References

[1], [2], [3], [4]

Examples

```
>>> ns = [0.204322076984, 0.070970999150, 0.267194323384, 0.296291964579, 0.
↪067046080882, 0.062489248292, 0.031685306730]
>>> Ks_y = [1.23466988745, 0.89727701141, 2.29525708098, 1.58954899888, 0.
↪23349348597, 0.02038108640, 1.40715641002]
>>> Ks_z = [1.52713341421, 0.02456487977, 1.46348240453, 1.16090546194, 0.
↪24166289908, 0.14815282572, 14.3128010831]
>>> Rachford_Rice_solution2(ns, Ks_y, Ks_z, beta_y=.1, beta_z=.6)
(0.6868328915094766, 0.06019424397668606, [0.1712804659711611, 0.08150738616425436,␣
↪0.1393433949193188, 0.20945175387703213, 0.15668977784027893, 0.22650123851718007,
↪ 0.015225982711774586], [0.21147483364299702, 0.07313470386530294, 0.
↪31982891387635903, 0.33293382568889657, 0.036586042443791586, 0.
↪004616341311925655, 0.02142533917172731], [0.26156812278601893, 0.
↪00200221914149187, 0.20392660665189805, 0.2431536850887592, 0.03786610596908295,␣
↪0.03355679851539993, 0.21792646184834918])
```

1.22.5 N Phase

chemicals.rachford_rice.**Rachford_Rice_solutionN**(*ns*, *Ks*, *betas*)

Solves the (phases -1) objectives functions of the Rachford-Rice flash equation for an N-phase system. Initial guesses are required for all phase fractions except the last. The Newton method is used, with an analytical Jacobian.

Parameters

ns [list[float]] Overall mole fractions of all species, [-]

Ks [list[list[float]]] Equilibrium K-values of all phases with respect to the *x* (reference) phase, [-]

betas [list[float]] Phase fraction initial guesses only for the first N - 1 phases; each value corresponds to the phase fraction of each set of the K values; if a phase fraction is specified for the last phase as well, it is ignored [-]

Returns

betas [list[float]] Phase fractions of all of the phases; one each for each K value set given, plus the reference phase phase fraction [-]

compositions [list[list[float]]] Mole fractions of each species in each phase; order each phase in the same order as the K values were provided, and then the *x* phase last, which was the reference phase [-]

Notes

This algorithm has been used without issue for 4 and 5 phase flashes.

Some helpful information was found in [1], although this method does not follow it exactly.

References

[1]

Examples

```
>>> ns = [0.204322076984, 0.070970999150, 0.267194323384, 0.296291964579, 0.
↪067046080882, 0.062489248292, 0.031685306730]
>>> Ks_y = [1.23466988745, 0.89727701141, 2.29525708098, 1.58954899888, 0.
↪23349348597, 0.02038108640, 1.40715641002]
>>> Ks_z = [1.52713341421, 0.02456487977, 1.46348240453, 1.16090546194, 0.
↪24166289908, 0.14815282572, 14.3128010831]
>>> Rachford_Rice_solutionN(ns, [Ks_y, Ks_z], [.1, .6])
([0.6868328915094767, 0.06019424397668605, 0.25297286451383727], [[0.
↪2114748336429970 2, 0.07313470386530294, 0.3198289138763589, 0.33293382568889657,␣
↪0.03658604244379159, 0.004616341311925657, 0.02142533917172731], [0.
↪26156812278601893, 0.00200221914149187, 0.203926606651898, 0.2431536850887592, 0.
↪03786610596908296, 0.033556798515399944, 0.21792646184834918], [0.
↪1712804659711611, 0.08150738616425436, 0.13934339491931877, 0.2094517538770321 3,␣
↪0.15668977784027896, 0.22650123851718015, 0.015225982711774586]])
```

1.22.6 Two Phase Utility Functions

chemicals.rachford_rice.**Rachford_Rice_polynomial**(*zs*, *Ks*)

Transforms the Rachford-Rice equation into a polynomial and returns its coefficients. A spelled-out solution is used for N from 2 to 5, derived with SymPy and optimized with the common sub expression approach.

> **Warning:** For large numbers of components (>20) this model performs terribly, though with future optimization it may be possible to have better performance.

$$\sum_{i=1}^{N} z_i C_i \left[\Pi_{j\neq i}^{N}\left(1 + \frac{V}{F}C_j\right)\right] = 0$$

$$C_i = K_i - 1.0$$

Once the above calculation is performed, it must be rearranged into polynomial form.

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

Returns

coeffs [float] Coefficients, with earlier coefficients corresponding to higher powers, [-]

Notes

Explicit calculations for any degree can be obtained with SymPy, changing N as desired:

```
>>> from sympy import *
>>> N = 4
>>> Cs = symbols('C0:' + str(N))
>>> zs = symbols('z0:' + str(N))
>>> alpha = symbols('alpha')
>>> tot = 0
>>> for i in range(N):
...     mult_sum = 1
>>> for j in range(N):
...     if j != i:
...         mult_sum *= (1 + alpha*Cs[j])
...     tot += zs[i]*Cs[i]*mult_sum
```

poly_expr = poly(expand(tot), alpha) coeff_list = poly_expr.all_coeffs() cse(coeff_list, optimizations='basic')

[1] suggests a matrix-math based approach for solving the model, but that has not been performed here. [1] also has explicit equations for up to N = 7 to derive the coefficients.

The general form was derived to be slightly different than that in [1], but is confirmed to also be correct as it matches other methods for solving the Rachford-Rice equation. [2] has similar information to [1].

The first coefficient is always 1.

The approach is also discussed in [3], with one example.

References

[1], [2], [3]

Examples

```
>>> Rachford_Rice_polynomial(zs=[0.5, 0.3, 0.2], Ks=[1.685, 0.742, 0.532])
[1.0, -3.6926529966760824, 2.073518878815093]
```

chemicals.rachford_rice.**Rachford_Rice_flash_error**(*V_over_F*, *zs*, *Ks*)

Calculates the objective function of the Rachford-Rice flash equation. This function should be called by a solver seeking a solution to a flash calculation. The unknown variable is *V_over_F*, for which a solution must be between 0 and 1.

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

Parameters

V_over_F [float] Vapor fraction guess [-]

zs [list[float]] Overall mole fractions of all species, [-]

Ks [list[float]] Equilibrium K-values, [-]

Returns

error [float] Deviation between the objective function at the correct V_over_F and the attempted V_over_F, [-]

Notes

The derivation is as follows:

$$Fz_i = Lx_i + Vy_i$$

$$x_i = \frac{z_i}{1 + \frac{V}{F}(K_i - 1)}$$

$$\sum_i y_i = \sum_i K_i x_i = 1$$

$$\sum_i (y_i - x_i) = 0$$

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)} = 0$$

This objective function was proposed in [1].

References

[1]

Examples

```
>>> Rachford_Rice_flash_error(0.5, zs=[0.5, 0.3, 0.2],
... Ks=[1.685, 0.742, 0.532])
0.04406445591174976
```

1.22.7 Numerical Notes

For the two-phase problem, there are the following ways of computing the vapor and liquid mole fractions once the vapor fraction and liquid fraction has been computed:

The most commonly shown expression is:

$$x_i = \frac{z_i}{1 + \frac{V}{F}(K_i - 1)}$$

This can cause numerical issues when K_i is near 1. It also shows issues near $\frac{V}{F}(K_i - 1) = -1$.

Another expression which avoids the second issue is

$$x_i = \frac{z_i}{\frac{L}{F} + (1 - \frac{L}{F})K_i}$$

Much like the other expression above this numerical issues but at different conditions: $\frac{L}{F} = 1$ and $\frac{L}{F} = -(1 - \frac{L}{F})K_i$.

One more expression using both liquid and vapor fraction is:

$$x_i = \frac{z_i}{K_i \frac{V}{F} + \frac{L}{F}}$$

This expression only has one problematic area: $K_i \frac{V}{F} = \frac{L}{F}$. Preferably, this is computed with a fused-multiply-add operation.

Another expression which flips the K value into the liquid form and swaps the vapor fraction for the liquid fraction in-line is as follows

$$x_i = \frac{\frac{z_i}{K_i}}{\frac{\frac{L}{F}}{K_i} + \frac{V}{F}}$$

This also has numerical problems when $-\frac{\frac{L}{F}}{K_i} = \frac{V}{F}$.

Even when computing a solution with high precision such as with *mpmath*, the resulting compositions and phase fractions may fail basic tests. In the following case, a nasty problem has a low-composition but relatively volatile last component. Mathematically, $1 = \frac{\frac{L}{F}x_i + \frac{V}{F}y_i}{z_i}$. This is true for all components except the last one in this case, where significant error exists.

```
>>> zs = [0.004632150100959984, 0.019748784459594933, 0.0037494212674659875, 0.
↪005049281503364983 5, 7.049818284201636e-05, 0.019252941309184937, 0.022923068733233923,
↪ 0.02751809363371991, 0.044055273670258854, 0.026348159124199914, 0.029384949788372902,
↪ 0.022368938441593926, 0.03876345111451487, 0.03440715821883388, 0.04220510198067186,␣
↪0.04109191458414686, 0.031180945124537895, 0.024703227642798916, 0.010618543295340965,␣
↪0.043262442161003854, 0.006774922650311977, 0.02418090788262392, 0.033168278052077886,␣
↪0.03325881573680989, 0.027794535589044905, 0.00302091746847699, 0.013693571363003955,␣
↪0.043274465132840854, 0.02431371852108292, 0.004119055065872986, 0.03314056562191489,␣
↪0.03926511182895087, 0.0305068048046159, 0.014495317922126952, 0.03603737707409988, 0.
↪04346278949361786, 0.019715052322446934, 0.028565255195219907, 0.023343683279902924, 0.
↪026532427286078915, 2.083372372767433e-06]
>>> Ks = [0.000312001984979, 0.478348350355814, 0.057460349529956, 0.142866526725442, 0.
↪186076915390803, 1.67832923245552, 0.010784509466239, 0.037204384948088, 0.
↪005359146955631, 2.41896552551221, 0.020514598049597, 0.104545054017411, 2.
↪37825397780443, 0.176463709057649, 0.000474240879865, 0.004738042026669, 0.
↪02556030236928, 0.00300089652604, 0.010614774675069, 1.75142303167203, 1.
↪47213647779132, 0.035773024794854, 4.15016401471676, 0.024475125100923, 0.
↪00206952065986, 2.09173484409107, 0.06290795470216, 0.001537212006245, 1.
↪16935817509767, 0.001830422812888, 0.058398776367331, 0.516860928072656, 1.
↪03039372722559, 0.460775800103578, 0.10980302936483, 0.009883724220094, 0.
↪021938589630783, 0.983011657214417, 0.01978995396409, 0.204144939961852, 14.
↪0521979447538]
>>> LF, VF, xs, ys = Rachford_Rice_solution_mpmath(zs=zs, Ks=Ks)
>>> (LF*xs[-1] + VF*ys[-1])/zs[-1]
1.0000000000028162
```

1.23 Chemical Reactions (chemicals.reaction)

This module contains lookup functions enthalpies and standard entropies of formation. Lookup functions are availa for the liquid, solid, and gas states. A compound may be in more than one lookup function.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.23.1 Solid Heat of Formation

chemicals.reaction.**Hfs**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's solid/crystaline standard phase heat of formation. The lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

Hfs [float] Solid standard-state heat of formation, [J/mol]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in Hfs_methods

See also:

Hfs_methods

Notes

Sources are:

- 'CRC', from the CRC handbook (1360 values) [1]
- 'WEBBOOK' (2000 values) [2]

References

[1], [2]

Examples

```
>>> Hfs('101-81-5') # Diphenylmethane
71500.0
```

chemicals.reaction.**Hfs_methods**(*CASRN*)

Return all methods available to obtain the solid-phase heat of formation for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Hfs with the given inputs.

See also:

Hfs

chemicals.reaction.**Hfs_all_methods = ('CRC', 'WEBBOOK')**

Tuple of method name keys. See the *Hfs* for the actual references

1.23.2 Liquid Heat of Formation

chemicals.reaction.**Hfl**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's liquid standard phase heat of formation. The lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

Hfl [float] Liquid standard-state heat of formation, [J/mol]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *Hfl_all_methods*.

See also:

Hfl_methods

Notes

Sources are:

- 'ATCT_L', the Active Thermochemical Tables version 1.112. [1]
- 'CRC', from the CRC handbook (1360 values) [2]
- 'WEBBOOK' (2000 values) [3]

References

[1], [2], [3]

Examples

```
>>> Hfl('67-56-1')
-238400.0
```

chemicals.reaction.**Hfl_methods**(*CASRN*)

Return all methods available to obtain the standard liquid-state heat of formation for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Hfl with the given inputs.

See also:

Hfl

chemicals.reaction.**Hfl_all_methods = ('ATCT_L', 'CRC', 'WEBBOOK', 'JANAF')**
Tuple of method name keys. See the *Hfl* for the actual references

1.23.3 Gas Heat of Formation

chemicals.reaction.**Hfg**(*CASRN*, *method=None*)
This function handles the retrieval of a chemical's gas heat of formation. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

Hfg [float] Ideal gas phase heat of formation, [J/mol]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in Hfg_methods

See also:

Hfg_methods

Notes

Function has data for approximately 8700 chemicals. Sources are:

- 'ATCT_G', the Active Thermochemical Tables version 1.112 (600 values) [1]
- 'TRC', from a 1994 compilation (1750 values) [2]
- 'CRC', from the CRC handbook (1360 values) [3]
- 'WEBBOOK', a NIST resource [6] containing mostly experimental and averaged values
- 'JANAF', the 1998 JANAF values online
- 'JOBACK', an estimation method for organic substances in [5]
- 'YAWS', a large compillation of values, mostly estimated (5000 values) [4]

'TRC' data may have come from computational procedures, for example petane is off by 30%.

References

[1], [2], [3], [4], [5], [6]

Examples

```
>>> Hfg('67-56-1')
-200700.0
>>> Hfg('67-56-1', method='YAWS')
-200900.0
>>> Hfg('67-56-1', method='CRC')
-201000.0
>>> Hfg('67-56-1', method='TRC')
-190100.0
```

chemicals.reaction.**Hfg_methods**(*CASRN*)

Return all methods available to obtain the gas phase heat of formation for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Hfg with the given inputs.

See also:

Hfg

chemicals.reaction.**Hfg_all_methods = ('ATCT_G', 'TRC', 'CRC', 'WEBBOOK', 'JANAF', 'YAWS', 'JOBACK')**

Tuple of method name keys. See the *Hfg* for the actual references

1.23.4 Solid Absolute Entropy

chemicals.reaction.**S0s**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's absolute entropy at a reference temperature of 298.15 K and pressure of 1 bar, in the solid state. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

S0s [float] Ideal gas standard absolute entropy of compound, [J/mol/K]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in *S0s_all_methods*.

See also:

S0s_methods

Notes

Sources are:

- 'CRC' [1] from the CRC handbook (1360 values)
- 'WEBBOOK', a NIST resource [2] containing mostly experimental and averaged values

References

[1], [2]

Examples

```
>>> S0s('7439-93-2') # Lithium
29.1
```

chemicals.reaction.**S0s_methods**(*CASRN*)

Return all methods available to obtain the absolute entropy of the compound in the solid phase for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the S0s with the given inputs.

See also:

S0s

chemicals.reaction.**S0s_all_methods = ('CRC', 'WEBBOOK')**

Tuple of method name keys. See the *S0s* for the actual references

1.23.5 Liquid Absolute Entropy

chemicals.reaction.**S0l**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's absolute entropy at a reference temperature of 298.15 K and pressure of 1 bar, in the liquid state.

Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

S0l [float] Ideal gas standard absolute entropy of compound, [J/mol/K]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *S0l_all_methods*.

See also:

S0l_methods

Notes

Sources are:

- 'CRC', from the CRC handbook

References

[1]

Examples

```
>>> S0l('7439-97-6') # Mercury
75.9
```

chemicals.reaction.**S0l_methods**(*CASRN*)

Return all methods available to obtain the absolute entropy for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the S0l with the given inputs.

See also:

S0l

chemicals.reaction.**S0l_all_methods = ('CRC', 'WEBBOOK', 'JANAF')**

Tuple of method name keys. See the *S0l* for the actual references

1.23.6 Gas Absolute Entropy

chemicals.reaction.**S0g**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's absolute entropy at a reference temperature of 298.15 K and pressure of 1 bar, in the ideal gas state.

Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

S0g [float] Ideal gas standard absolute entropy of compound, [J/mol/K]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *S0g_all_methods*

See also:

S0g_methods

Notes

Function has data for approximately 5400 chemicals. Sources are:

- 'CRC', from the CRC handbook (520 values)
- 'YAWS', a large compillation of values, mostly estimated (4890 values)
- 'WEBBOOK', a NIST resource [3] containing mostly experimental and averaged values

References

[1], [2], [3]

Examples

```
>>> S0g('67-56-1')
239.9
>>> S0g('67-56-1', method='YAWS')
239.88
```

chemicals.reaction.**S0g_methods**(*CASRN*)

Return all methods available to obtain the S0g for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the S0g with the given inputs.

See also:

S0g

chemicals.reaction.**S0g_all_methods = ('CRC', 'WEBBOOK', 'JANAF', 'YAWS')**

Tuple of method name keys. See the *S0g* for the actual references

1.23.7 Utility Functions

chemicals.reaction.**Gibbs_formation**(*dHf*, *S0_abs*, *dHfs_std*, *S0_abs_elements*, *coeffs_elements*, *T_ref=298.15*)

This function calculates the Gibbs free energy of formation of a compound, from its constituent elements.

The calculated value will be for a "standard-state" value if *dHf* and *S0_abs* are provided in the standard state; or it will be in an "ideal gas" basis if they are both for an ideal gas. For compounds which are gases at STP, the two values are the same.

Parameters

dHf [float] Molar enthalpy of formation of the created compound, [J/mol]

S0_abs [float] Absolute molar entropy of the created compound at the reference temperature, [J/mol/K]

dHfs_std [list[float]] List of standard molar enthalpies of formation of all elements used in the formation of the created compound, [J/mol]

S0_abs_elements [list[float]] List of standard absolute molar entropies at the reference temperature of all elements used in the formation of the created compound, [J/mol/K]

coeffs_elements [list[float]] List of coefficients for each compound (i.e. 1 for C, 2 for H2 if the target is methane), in the same order as *dHfs_std* and *S0_abs_elements*, [-]

T_ref [float, optional] The standard state temperature, default 298.15 K; few values are tabulated at other temperatures, [-]

Returns

dGf [float] Gibbs free energy of formation for the created compound, [J/mol]

Notes

Be careful for elements like Bromine - is the tabulated value for Br2 or Br?

References

[1]

Examples

Calculate the standard-state Gibbs free energy of formation for water, using water's standard state heat of formation and absolute entropy at 298.15 K:

```
>>> Gibbs_formation(-285830, 69.91,  [0, 0], [130.571, 205.147], [1, .5])
-237161.633825
```

Calculate the ideal-gas state Gibbs free energy of formation for water, using water's ideal-gas state heat of formation and absolute entropy at 298.15 K as a gas:

```
>>> Gibbs_formation(-241818, 188.825,  [0, 0], [130.571, 205.147], [1, .5])
-228604.141075
```

Calculate the Gibbs free energy of formation for CBrF3 (it is a gas at STP, so its standard-state and ideal-gas state values are the same) at 298.15 K:

```
>>> Gibbs_formation(-648980, 297.713, [0, 0, 0], [5.74, 152.206, 202.789], [1, .5,␣
↪1.5])
-622649.329975
```

Note in the above calculation that the Bromine's *S0* and *Hf* are for Br2; and that the value for Bromine as a liquid, which is its standard state, is used.

chemicals.reaction.**entropy_formation**(*Hf*, *Gf*, *T_ref=298.15*)

This function calculates the entropy of formation of a compound, from its constituent elements.

The calculated value will be for a "standard-state" value if *Hf* and *Gf* are provided in the standard state; or it will be in an "ideal gas" basis if they are both for an ideal gas. For compounds which are gases at STP, the two values are the same.

Parameters

Hf [float] Molar enthalpy of formation of the compound, [J/mol]

Gf [float] Molar Gibbs free energy of formation of the compound, [J/mol]

T_ref [float, optional] The standard state temperature, default 298.15 K; few values are tabulated at other temperatures, [-]

Returns

S0 [float] Entropy of formation of the compound, [J/mol/K]

Examples

Entropy of formation of methane:

```
>>> entropy_formation(Hf=-74520, Gf=-50490)
-80.59701492537314
```

Entropy of formation of water in ideal gas state:

```
>>> entropy_formation(Hf=-241818, Gf=-228572)
-44.427301693778304
```

chemicals.reaction.**Hf_basis_converter**(*Hvapm*, *Hf_liq=None*, *Hf_gas=None*)

This function converts a liquid or gas enthalpy of formation to the other. This is useful, as thermodynamic packages often work with ideal- gas as the reference state and require ideal-gas enthalpies of formation.

Parameters

Hvapm [float] Molar enthalpy of vaporization of compound at 298.15 K or (unlikely) the reference temperature, [J/mol]

Hf_liq [float, optional] Enthalpy of formation of the compound in its liquid state, [J/mol]

Hf_gas [float, optional] Enthalpy of formation of the compound in its ideal-gas state, [J/mol]

Returns

Hf_calc [float, optional] Enthalpy of formation of the compound in the other state to the one provided, [J/mol]

Examples

Calculate the ideal-gas enthalpy of formation for water, from its standard- state (liquid) value:

```
>>> Hf_basis_converter(44018, Hf_liq=-285830)
-241812
```

Calculate the standard-state (liquid) enthalpy of formation for water, from its ideal-gas value:

```
>>> Hf_basis_converter(44018, Hf_gas=-241812)
-285830
```

1.23.8 Chemical Reactions

chemicals.reaction.**balance_stoichiometry**(*matrix*, *rounding=9*, *allow_fractional=False*)

This function balances a chemical reaction.

Parameters

matrix [list[list[float]]]

Chemical reaction matrix for further processing; rows contain element counts of each compound, and the columns represent each chemical, [-]

Returns

coefficients [list[float]] Balanced coefficients; all numbers are positive, [-]

Notes

Balance the reaction 4 NH3 + 5 O2 = 4 NO + 6 H2O, without knowing the coefficients:

```
>>> matrix = stoichiometric_matrix([{'N': 1, 'H': 3}, {'O': 2}, {'N': 1, 'O': 1}, {
↪'H': 2, 'O': 1}], [True, True, False, False])
>>> matrix
[[3, 0, 0, -2], [1, 0, -1, 0], [0, 2, -1, -1]]
>>> balance_stoichiometry(matrix)
[4.0, 5.0, 4.0, 6.0]
>>> balance_stoichiometry(matrix, allow_fractional=True)
[1.0, 1.25, 1.0, 1.5]
```

This algorithm relies on *scipy*. The behavior of this function for inputs which do not have a unique solution is undefined.

This algorithm may suffer from floating point issues. If you believe there is an error in the result, please report your reaction to the developers.

References

[1], [2]

chemicals.reaction.**stoichiometric_matrix**(*atomss*, *reactants*)

This function calculates a stoichiometric matrix of reactants and stoichiometric matrix, as required by a solver to compute the reation coefficients.

Parameters

atomss [list[dict[(str, float)]]] A list of dictionaties of (element, element_count) pairs for each chemical, [-]

reactants [list[bool]] List of booleans indicating whether each chemical is a reactant (True) or a product (False), [-]

Returns

matrix [list[list[float]]]

Chemical reaction matrix for further processing; rows contain element counts of each compound, and the columns represent each chemical, [-]

Notes

The rows of the matrix contain the element counts of each compound, and the columns represent each chemical.

References

[1], [2]

Examples

MgO2 -> Mg + 1/2 O2 (k=1)

```
>>> stoichiometric_matrix([{'Mg': 1, 'O': 1}, {'Mg': 1}, {'O': 2}], [True, False,␣
↪False])
[[1, -1, 0], [1, 0, -2]]
```

Cl2 + propylene -> allyl chloride + HCl

```
>>> stoichiometric_matrix([{'Cl': 2}, {'C': 3, 'H': 6}, {'C': 3, 'Cl': 1, 'H': 5}, {
↪'Cl': 1, 'H': 1}], [True, True, False, False, False])
[[0, 3, -3, 0], [2, 0, -1, -1], [0, 6, -5, -1]]
```

Al + 4HNO3 -> Al(NO3)3 + NO + 2H2O (k=1)

```
>>> stoichiometric_matrix([{'Al': 1}, {'H': 1, 'N': 1, 'O': 3}, {'Al': 1, 'N': 3, 'O
↪': 9}, {'N': 1, 'O': 1}, {'H': 2, 'O': 1}], [True, True, False, False, False])
[[1, 0, -1, 0, 0], [0, 1, 0, 0, -2], [0, 1, -3, -1, 0], [0, 3, -9, -1, -1]]
```

4Fe + 3O2 -> 2(Fe2O3) (k=2)

```
>>> stoichiometric_matrix([{'Fe': 1}, {'O': 2}, {'Fe':2, 'O': 3}], [True, True,␣
↪False])
[[1, 0, -2], [0, 2, -3]]
```

4NH3 + 5O2 -> 4NO + 6(H2O) (k=4)

```
>>> stoichiometric_matrix([{'N': 1, 'H': 3}, {'O': 2}, {'N': 1, 'O': 1}, {'H': 2, 'O
↪': 1}], [True, True, False, False])
[[3, 0, 0, -2], [1, 0, -1, 0], [0, 2, -1, -1]]
```

No unique solution: C2H5NO2 + C3H7NO3 + 2C6H14N4O2 + 3C5H9NO2 + 2C9H11NO2 -> 8H2O + C50H73N15O11

```
>>> stoichiometric_matrix([{'C': 2, 'H': 5, 'N': 1, 'O': 2}, {'C': 3, 'H': 7, 'N':␣
↪1, 'O': 3}, {'C': 6, 'H': 14, 'N': 4, 'O': 2}, {'C': 5, 'H': 9, 'N': 1, 'O': 2}, {
↪'C': 9, 'H': 11, 'N': 1, 'O': 2}, {'H': 2, 'O': 1}, {'C': 50, 'H': 73, 'N': 15, 'O
↪': 11}], [True, True, True, True, True, False, False])
[[2, 3, 6, 5, 9, 0, -50], [5, 7, 14, 9, 11, -2, -73], [1, 1, 4, 1, 1, 0, -15], [2,␣
↪3, 2, 2, 2, -1, -11]]
```

1.24 Refractive Index (chemicals.refractivity)

This module contains various refractive index lookup, calculation, and unit conversion routines and dataframes.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.24.1 Lookup Functions

chemicals.refractivity.**RI**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's refractive index. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Function has data for approximately 4500 chemicals.

Parameters

CASRN [str] CASRN [-]

Returns

RI [float] Refractive Index on the Na D line, [-]

T [float or None] Temperature at which refractive index reading was made; None if not available, [K]

Other Parameters

method [string, optional] A string for the method name to use, as defined by constants in RI_methods

Notes

The available sources are as follows:

- 'CRC', a compillation of Organic RI data in [1].
- 'WIKIDATA', data from the Wikidata project [2]

References

[1], [2]

Examples

```
>>> RI(CASRN='64-17-5')
(1.3611, 293.15)
>>> RI("60-35-5")
(1.4278, None)
>>> RI('100-41-4', method='WIKIDATA')
(1.495, None)
```

chemicals.refractivity.**RI_methods**(*CASRN*)
Return all methods available to obtain the refractive index for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the RI with the given inputs.

See also:

RI

chemicals.refractivity.**RI_all_methods = ('CRC', 'WIKIDATA')**
Tuple of method name keys. See the *RI* for the actual references

1.24.2 Correlations for Specific Substances

chemicals.refractivity.**RI_IAPWS**(*T*, *rho*, *wavelength=5.893e-07*)
Calculates the refractive index of water at a given temperature, density, and wavelength.

$$n(\rho, T, \lambda) = \left(\frac{2A+1}{1-A}\right)^{0.5}$$

$$A(\delta, \theta, \Lambda) = \delta\left(a_0 + a_1\delta + a_2\theta + a_3\Lambda^2\theta + a_4\Lambda^{-2}\frac{a_5}{\Lambda^2 - \Lambda_{UV}^2} + \frac{a_6}{\Lambda^2 - \Lambda_{IR}^2} + a_7\delta^2\right)$$

$$\delta = \rho/(1000\ \text{kg/m}^3)$$

$$\theta = T/273.15\text{K}$$

$$\Lambda = \lambda/0.589\mu m$$

$$\Lambda_{IR} = 5.432937$$

$$\Lambda_{UV} = 0.229202$$

Parameters

T [float] Temperature of the water [K]

rho [float] Density of the water [kg/m^3]

wavelength [float] Wavelength of fluid [meters]

Returns

RI [float] Refractive index of the water, [-]

Notes

This function is valid in the following range: 261.15 K < T < 773.15 K 0 < rho < 1060 kg/m^3 0.2 < wavelength < 1.1 micrometers

Test values are from IAPWS 2010 book.

References

[1]

Examples

```
>>> RI_IAPWS(298.15, 997.047435)
1.3328581926471605
```

1.24.3 Unit Conversions

chemicals.refractivity.**brix_to_RI**(*brix*)

Convert a refractive index measurement on the *brix* scale to a standard refractive index.

Parameters

brix [float] Degrees brix to be converted, [°Bx]

Returns

RI [float] Refractive index, [-]

Notes

The scale is officially defined from 0 to 85; but the data source contains values up to 95. Linear extrapolation outside of the bounds is performed; and a table of 96 values are linearly interpolated.

The ICUMSA (International Committee of Uniform Method of Sugar Analysis) published a document setting out the reference values in 1974; but an original data source has not been found and reviewed.

References

[1]

Examples

```
>>> brix_to_RI(5.8)
1.341452
>>> brix_to_RI(0.0)
1.33299
>>> brix_to_RI(95.0)
1.532
```

chemicals.refractivity.**RI_to_brix**(*RI*)

Convert a standard refractive index measurement to the *brix* scale.

Parameters

RI [float] Refractive index, [-]

Returns

brix [float] Degrees brix to be converted, [°Bx]

Notes

The scale is officially defined from 0 to 85; but the data source contains values up to 95.

Linear extrapolation to values under 0 or above 95 is performed.

The ICUMSA (International Committee of Uniform Method of Sugar Analysis) published a document setting out the reference values in 1974; but an original data source has not been found and reviewed.

References

[1]

Examples

```
>>> RI_to_brix(1.341452)
5.800000000000059
>>> RI_to_brix(1.33299)
0.0
>>> RI_to_brix(1.532)
95.0
```

1.24.4 Utility functions

chemicals.refractivity.**polarizability_from_RI**(*RI*, *Vm*)

Returns the polarizability of a fluid given its molar volume and refractive index.

$$\alpha = \left(\frac{3}{4\pi N_A}\right)\left(\frac{n^2-1}{n^2+2}\right)V_m$$

Parameters

RI [float] Refractive Index on Na D line, [-]

Vm [float] Molar volume of fluid, [m^3/mol]

Returns

alpha [float] Polarizability [m^3]

Notes

This Lorentz-Lorentz-expression is most correct when van der Waals interactions dominate. Alternate conversions have been suggested. This is often expressed in units of cm^3 or Angstrom^3. To convert to these units, multiply by 1E9 or 1E30 respectively.

References

[1]

Examples

```
>>> polarizability_from_RI(1.3611, 5.8676E-5)
5.147658206528923e-30
```

chemicals.refractivity.**molar_refractivity_from_RI**(*RI*, *Vm*)

Returns the molar refractivity of a fluid given its molar volume and refractive index.

$$R_m = \left(\frac{n^2-1}{n^2+2}\right) V_m$$

Parameters

RI [float] Refractive Index on Na D line, [-]

Vm [float] Molar volume of fluid, [m^3/mol]

Returns

Rm [float] Molar refractivity [m^3/mol]

References

[1]

Examples

```
>>> molar_refractivity_from_RI(1.3611, 5.8676E-5)
1.2985217089649597e-05
```

chemicals.refractivity.**RI_from_molar_refractivity**(*Rm*, *Vm*)

Returns the refractive index of a fluid given its molar volume and molar refractivity.

$$RI = \sqrt{\frac{-2R_m - V_m}{R_m - V_m}}$$

Parameters

Rm [float] Molar refractivity [m^3/mol]

Vm [float] Molar volume of fluid, [m^3/mol]

Returns

RI [float] Refractive Index on Na D line, [-]

References

[1]

Examples

```
>>> RI_from_molar_refractivity(1.2985e-5, 5.8676E-5)
1.3610932757685672
```

1.24.5 Pure Component Liquid Fit Correlations

chemicals.refractivity.**TDE_RIXExpansion**(*T*, *Bs*, *Cs*, *wavelength=5.8926e-07*)

Calculates the refractive index of a pure liquid at a given temperature, and wavelength, using the NIST TDE RIXExpansion formula [1].

$$n(T, \lambda) = \sum_{i=0}^{i} B_i t^i + \sum_j C_j w^j$$

$$t = T - 298.15$$

$$w = WL \times 10^9 - 589.26$$

Parameters

T [float] Temperature of the fluid [K]

Bs [list[float]] Polynomial temperature expansion coefficients, in reverse order to the polynomial (as needed for efficient computation with horner's method'), [-]

Cs [list[float]] Polynomial wavelength expansion coefficients, in reverse order to the polynomial (as needed for efficient computation with horner's method'), [-]

wavelength [float] Wavelength of fluid [meters]

Returns

RI [float] Refractive index of the pure fluid, [-]

References

[1]

Examples

```
>>> TDE_RIXExpansion(330.0, Bs=[-0.000125041, 1.33245], Cs=[1.20771e-7, -3.56795e-5,
↪ 0.0], wavelength=589.26e-9*.7)
1.33854894426073
```

1.25 Health, Safety, and Flammability Properties (chemicals.safety)

This module contains functions for lookup the following properties for a chemical:

- Short-term Exposure Limit (STEL)
- Time-Weighted Average Exposure Limit (TWA)
- Celing limit for working exposure
- Whether a chemicals is absorbed thorough human skin
- Whether a chemical is a carcinogen, suspected of being a carcinogen, or has been identified as unlikely to be a carcinogen
- Flash point
- Auto ignition point
- Lower flammability limit
- Upper flammability limit

In addition, several estimation methods for chemicals without flammability limits are provided and for calculating the flammability limits of mixtures.

This module also contains several utility functions.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.25.1 Short-term Exposure Limit

chemicals.safety.**STEL**(*CASRN*, *method=None*)

This function handles the retrieval of Short-term Exposure Limit (STEL) on worker exposure to dangerous chemicals.

Parameters

CASRN [str] CASRN, [-]

method [str] Name of method to use, [-]

Returns

STEL [float] Short-term Exposure Limit, [ppm or mg/m^3]

units [str] One of ppm or mg/m^3, [-]

Notes

The ppm value is preferentially returned if both are available. While they can be converted in specific cases, it is better to work with the specified units of the original source.

Examples

```
>>> STEL('67-64-1')
(750.0, 'ppm')
>>> STEL('7664-38-2')
(0.7489774978301237, 'ppm')
>>> STEL('55720-99-5')
(2.0, 'mg/m^3')
```

chemicals.safety.**STEL_methods**(*CASRN*)

Return all methods available to obtain STEL for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain STEL with the given inputs.

See also:

STEL

chemicals.safety.**STEL_all_methods = ('Ontario Limits',)**

Tuple of method name keys. See the *STEL* for the actual references

1.25.2 Time-Weighted Average Exposure Limit

chemicals.safety.**TWA**(*CASRN*, *method=None*)

Return the Time-Weighted Average exposure limits (TWA) for the desired chemical if it is available.

Parameters

CASRN [str] CASRN, [-]

method [str] Name of method to use, [-]

Returns

TWA [float] Time-Weighted Average exposure, [ppm or mg/m^3]

units [str] One of ppm or mg/m^3, [-]

Notes

The ppm value is preferentially returned if both are available. While they can be converted in specific cases, it is better to work with the specified units of the original source.

Examples

```
>>> TWA('98-00-0')
(10.0, 'ppm')
>>> TWA('1303-00-0')
(5.0742430905659505e-05, 'ppm')
```

chemicals.safety.**TWA_methods**(*CASRN*)

Return all methods available to obtain the Time-Weighted Average exposure limits (TWA) for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain TWA with the given inputs.

See also:

TWA

Examples

```
>>> TWA_methods('71-43-2')
['Ontario Limits']
```

chemicals.safety.**TWA_all_methods = ('Ontario Limits',)**

Tuple of method name keys. See the *TWA* for the actual references

1.25.3 Ceiling Limit

chemicals.safety.**Ceiling**(*CASRN*, *method=None*)

This function handles the retrieval of ceiling limits on worker exposure to dangerous chemicals. Ceiling limits are not to be exceeded at any time.

Parameters

CASRN [str] CASRN, [-]

method [str] Name of method to use, [-]

Returns

Ceiling [float] Ceiling Limit, [ppm or mg/m^3]

units [str] One of ppm or mg/m^3, [-]

Examples

```
>>> Ceiling('75-07-0')
(25.0, 'ppm')
>>> Ceiling('1395-21-7')
(6e-05, 'mg/m^3')
```

chemicals.safety.**Ceiling_methods**(*CASRN*)

Return all methods available to obtain Ceiling limits for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain Ceiling limits with the given inputs.

See also:

Ceiling

chemicals.safety.**Ceiling_all_methods = ('Ontario Limits',)**

Tuple of method name keys. See the *Ceiling* for the actual references

1.25.4 Skin Absorbance

chemicals.safety.**Skin**(*CASRN*, *method=None*)

This function handles the retrieval of whether or not a chemical can be absorbed through the skin, relevant to chemical safety calculations.

Parameters

CASRN [str] CASRN, [-]

method [str] Name of method to use, [-]

Returns

skin [bool] Whether or not the substance is absorbed through human skin, [-]

Examples

```
>>> Skin('108-94-1')
True
>>> Skin('1395-21-7')
False
```

chemicals.safety.**Skin_methods**(*CASRN*)
Return all methods available to obtain whether or not a chemical can be absorbed through the skin.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain whether or not a chemical can be absorbed through the skin.

See also:

Skin

chemicals.safety.**Skin_all_methods = ('Ontario Limits',)**
Tuple of method name keys. See the *Skin* for the actual references

1.25.5 Carcinogenicity

chemicals.safety.**Carcinogen**(*CASRN*, *method=None*)
Looks up if a chemical is listed as a carcinogen or not according to either a specifc method or with all methods. Returns either the status as a string for a specified method, or the status of the chemical in all available data sources, in the format {source: status}.

Parameters

CASRN [str] CASRN [-]

Returns

status [str or dict] Carcinogen status information [-].

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *Carcinogen_all_methods*.

Notes

Supported methods are:

- **IARC**: International Agency for Research on Cancer, [1]. As extracted with a last update of February 22, 2016. Has listing information of 863 chemicals with CAS numbers. Chemicals without CAS numbers not included here. If two listings for the same CAS were available, the harshest rating was used. If two listings were available published at different times, the latest value was used. All else equal, the most pessimistic value was used.
- **NTP**: National Toxicology Program, [2]. Has data on 228 chemicals.

References

[1], [2]

Examples

```
>>> Carcinogen('61-82-5')
{'International Agency for Research on Cancer': 'Not classifiable as to its␣
↪carcinogenicity to humans (3)', 'National Toxicology Program 13th Report on␣
↪Carcinogens': 'Reasonably Anticipated'}
```

chemicals.safety.**Carcinogen_methods**(*CASRN*)

Return all methods available to obtain Carcinogen listings for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain Carcinogen listings with the given inputs.

See also:

Carcinogen

chemicals.safety.**Carcinogen_all_methods = ('International Agency for Research on Cancer', 'National Toxicology Program 13th Report on Carcinogens')**

Tuple of method name keys. See the *Carcinogen* for the actual references

1.25.6 Flash Point

chemicals.safety.**T_flash**(*CASRN*, *method=None*)

This function handles the retrieval or calculation of a chemical's flash point. Lookup is based on CASRNs. No predictive methods are currently implemented. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

T_flash [float] Flash point of the chemical, [K]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *T_flash_all_methods*,

See also:

T_flash_methods

Notes

Preferred source is 'IEC 60079-20-1 (2010)' [1], with the secondary source 'NFPA 497 (2008)' [2] having very similar data. A third source 'Serat DIPPR (2017)' [3] provides third hand experimental but evaluated data from the DIPPR database, version unspecified, for 870 compounds.

The predicted values from the DIPPR databank are also available in the supporting material in [3], but are not included.

References

[1], [2], [3], [4]

Examples

```
>>> T_flash(CASRN='64-17-5')
285.15
>>> T_flash('111-69-3', method='WIKIDATA')
365.92778
```

chemicals.safety.**T_flash_methods**(*CASRN*)

Return all methods available to obtain T_flash for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain T_flash with the given inputs.

See also:

T_flash

chemicals.safety.**T_flash_all_methods = ('IEC 60079-20-1 (2010)', 'NFPA 497 (2008)', 'Serat DIPPR (2017)', 'WIKIDATA')**

Tuple of method name keys. See the *T_flash* for the actual references

1.25.7 Autoignition Point

chemicals.safety.**T_autoignition**(*CASRN*, *method=None*)

This function handles the retrieval or calculation of a chemical's autoifnition temperature. Lookup is based on CASRNs. No predictive methods are currently implemented. Will automatically select a data source to use if no Method is provided; returns None if the data is not available.

Parameters

CASRN [str] CASRN [-]

Returns

Tautoignition [float] Autoignition point of the chemical, [K].

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *T_autoignition_all_methods*.

See also:

T_autoignition_methods

Notes

Preferred source is 'IEC 60079-20-1 (2010)' [1], with the secondary source 'NFPA 497 (2008)' [2] having very similar data.

References

[1], [2], [3]

Examples

```
>>> T_autoignition(CASRN='71-43-2')
771.15
>>> T_autoignition('111-69-3', method='WIKIDATA')
823.15
```

chemicals.safety.**T_autoignition_methods**(*CASRN*)

Return all methods available to obtain T_autoignition for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain T_autoignition with the given inputs.

See also:

T_autoignition

chemicals.safety.**T_autoignition_all_methods = ('IEC 60079-20-1 (2010)', 'NFPA 497 (2008)', 'WIKIDATA')**

Tuple of method name keys. See the *T_autoignition* for the actual references

1.25.8 Lower Flammability Limit

chemicals.safety.**LFL**(*Hc=None*, *atoms=None*, *CASRN=''*, *method=None*)

This function handles the retrieval or calculation of a chemical's Lower Flammability Limit. Lookup is based on CASRNs. Will automatically select a data source to use if no Method is provided; returns None if the data is not available.

Parameters

Hc [float, optional] Heat of combustion of gas [J/mol].

atoms [dict, optional] Dictionary of atoms and atom counts.

CASRN [str, optional] CASRN, [-]

Returns

LFL [float] Lower flammability limit of the gas in an atmosphere at STP, [mole fraction].

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *LFL_all_methods*.

Notes

Preferred source is 'IEC 60079-20-1 (2010)' [1], with the secondary source 'NFPA 497 (2008)' [2] having very similar data. If the heat of combustion is provided, the estimation method *Suzuki_LFL* can be used. If the atoms of the molecule are available, the method *Crowl_Louvar_LFL* can be used.

References

[1], [2], [3]

Examples

```
>>> LFL(CASRN='71-43-2')
0.012
>>> LFL(Hc=-890590.0, atoms={'C': 1, 'H': 4}, CASRN='74-82-8')
0.044
>>> LFL(CASRN='111-69-3', method='WIKIDATA')
0.017
```

chemicals.safety.**LFL_methods**(*Hc=None*, *atoms=None*, *CASRN=''*)

Return all methods available to obtain LFL for the desired chemical.

Parameters

Hc [float, optional] Heat of combustion of gas [J/mol].

atoms [dict, optional] Dictionary of atoms and atom counts.

CASRN [str, optional] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain LFL with the given inputs.

See also:

LFL

Examples

Methane

```
>>> LFL_methods(Hc=-890590.0, atoms={'C': 1, 'H': 4}, CASRN='74-82-8')
['IEC 60079-20-1 (2010)', 'NFPA 497 (2008)', 'Suzuki (1994)', 'Crowl and Louvar␣
↪(2001)']
```

chemicals.safety.**LFL_all_methods = ('IEC 60079-20-1 (2010)', 'NFPA 497 (2008)', 'WIKIDATA', 'Suzuki (1994)', 'Crowl and Louvar (2001)')**

Tuple of method name keys. See the *LFL* for the actual references

chemicals.safety.**Suzuki_LFL**(*Hc*)

Calculates lower flammability limit, using the Suzuki [1] correlation. Uses heat of combustion only.

The lower flammability limit of a gas is air is:

$$\text{LFL} = \frac{-3.42}{\Delta H_c^\circ} + 0.569$$

$$\Delta H_c^\circ + 0.0538 \Delta H_c^{\circ 2} + 1.80$$

Parameters

Hc [float] Heat of combustion of gas [J/mol]

Returns

LFL [float] Lower flammability limit, mole fraction [-]

Notes

Fit performed with 112 compounds, r^2 was 0.977. LFL in percent volume in air. Hc is at standard conditions, in MJ/mol. 11 compounds left out as they were outliers. Equation does not apply for molecules with halogen atoms, only hydrocarbons with oxygen or nitrogen or sulfur. No sample calculation provided with the article. However, the equation is straightforward. Limits of equations's validity are -6135596 J where it predicts a LFL of 0, and -48322129 J where it predicts a LFL of 1.

References

[1]

Examples

Pentane, 1.5 % LFL in literature

```
>>> Suzuki_LFL(-3536600)
0.014276107095811815
```

chemicals.safety.**Crowl_Louvar_LFL**(*atoms*)

Calculates lower flammability limit, using the Crowl-Louvar [1] correlation. Uses molecular formula only. The lower flammability limit of a gas is air is:

$$C_m H_x O_y + z O_2 \rightarrow m CO_2 + \frac{x}{2} H_2O$$

$$\text{LFL} = \frac{0.55}{4.76m + 1.19x - 2.38y + 1}$$

Parameters

atoms [dict] Dictionary of atoms and atom counts

Returns

LFL [float] Lower flammability limit, mole fraction

Notes

Coefficient of 0.55 taken from [2]

References

[1], [2]

Examples

Hexane, example from [1], lit. 1.2 %

```
>>> Crowl_Louvar_LFL({'H': 14, 'C': 6})
0.011899610558199915
```

chemicals.safety.**LFL_ISO_10156_2017**(*zs*, *LFLs*, *CASs*)

Calculate the lower flammability limit of a mixture of combustible gases and inert gases according to ISO 10156 (2017) [1].

$$\text{LFL} = \frac{1}{\sum_{i=1}^{n_{combustible}} \frac{A_i}{\text{LFL}'_i}}$$

$$\text{LFL}'_i = \frac{1 - \text{LFL}'_m - (1-K)\frac{\sum_j^{n_{inert}} B_j}{\sum_j^{n_{combustible}} A_j}\text{LFL}'_m}{100 - \text{LFL}'_m}\text{LFL}_i$$

$$K = \sum_i^{n_{inert}} z_i K_k$$

The *B* sum is the total mole fraction of all inert gas compounds; and the *A* sum is the total mole fraction of all combustible compounds. K_k are the looked up inert gas coefficients. LFL'_m is calculated as the Le Chatelier's lower flammability limit if there were no inert gases in the mixture.

Parameters

zs [list[float]] Mole fractions of all components in a gas including inerts, [-]

LFLs [list[float]] Lower or upper flammability limits for each flammable component in a gas, [-]

CASs [list[str]] CAS numbers of each compound; required to look up inert gas factors, [-]

Returns

LFL [float] Lower or flammability limit of a gas mixture, [-]

Notes

Inert gas parameters are available for O2, N2, CO2, He, Ar, Ne, Kr, Xe, SO2, SF6, CF4, C3F8, and C2HF5.

References

[1]

Examples

All the sample problems from [1] have been implemented as tests.

```
>>> zs = [.15, .15, .3, .35+.05*.79, .05*.21]
>>> LFLs = [.04, .044, None, None, None]
>>> CASs = ['1333-74-0', '74-82-8', '124-38-9', '7727-37-9', '7782-44-7']
>>> LFL_ISO_10156_2017(zs, LFLs, CASs)
0.1427372274
```

1.25.9 Upper Flammability Limit

chemicals.safety.**UFL**(*Hc=None*, *atoms=None*, *CASRN=''*, *method=None*)

This function handles the retrieval or calculation of a chemical's Upper Flammability Limit. Lookup is based on CASRNs. Two predictive methods are currently implemented. Will automatically select a data source to use if no Method is provided; returns None if the data is not available.

Parameters

Hc [float, optional] Heat of combustion of gas [J/mol]

atoms [dict, optional] Dictionary of atoms and atom counts

CASRN [str, optional] CASRN [-]

Returns

UFL [float] Upper flammability limit of the gas in an atmosphere at STP, [mole fraction]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *UFL_all_methods*.

Notes

Preferred source is 'IEC 60079-20-1 (2010)' [1], with the secondary source 'NFPA 497 (2008)' [2] having very similar data. If the heat of combustion is provided, the estimation method *Suzuki_UFL* can be used. If the atoms of the molecule are available, the method *Crowl_Louvar_UFL* can be used.

References

[1], [2], [3]

Examples

```
>>> UFL(CASRN='71-43-2')
0.086
```

Methane

```
>>> UFL(Hc=-890590.0, atoms={'C': 1, 'H': 4}, CASRN='74-82-8')
0.17
>>> UFL(CASRN='111-69-3', method='WIKIDATA')
0.05
```

chemicals.safety.**UFL_methods**(*Hc=None*, *atoms=None*, *CASRN=''*)
Return all methods available to obtain UFL for the desired chemical.

Parameters

Hc [float, optional] Heat of combustion of gas [J/mol].

atoms [dict, optional] Dictionary of atoms and atom counts.

CASRN [str, optional] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain UFL with the given inputs.

See also:

UFL

Examples

Methane

```
>>> UFL_methods(Hc=-890590.0, atoms={'C': 1, 'H': 4}, CASRN='74-82-8')
['IEC 60079-20-1 (2010)', 'NFPA 497 (2008)', 'Suzuki (1994)', 'Crowl and Louvar␣
↪(2001)']
```

chemicals.safety.**UFL_all_methods = ('IEC 60079-20-1 (2010)', 'NFPA 497 (2008)', 'WIKIDATA', 'Suzuki (1994)', 'Crowl and Louvar (2001)')**
Tuple of method name keys. See the *UFL* for the actual references

chemicals.safety.**Suzuki_UFL**(*Hc*)
Calculates upper flammability limit, using the Suzuki [1] correlation. Uses heat of combustion only. The upper flammability limit of a gas is air is:

$$\text{UFL} = 6.3\Delta H_c^\circ + 0.567\Delta H_c^{\circ 2} + 23.5$$

Parameters

Hc [float] Heat of combustion of gas [J/mol]

Returns

UFL [float] Upper flammability limit, mole fraction

Notes

UFL in percent volume in air according to original equation. Hc is at standard conditions in the equation, in units of MJ/mol. AAPD = 1.2% for 95 compounds used in fit. Somewhat better results than the High and Danner method. 4.9% < UFL < 23.0% -890.3 kJ/mol < dHc < -6380 kJ/mol r^2 = 0.989 Sample calculations provided for all chemicals, both this method and High and Danner. Examples are from the article. Predicts a UFL of 1 at 7320190 J and a UFL of 0 at -5554160 J.

References

[1]

Examples

Pentane, literature 7.8% UFL

```
>>> Suzuki_UFL(-3536600)
0.0831119493052
```

chemicals.safety.**Crowl_Louvar_UFL**(*atoms*)

Calculates upper flammability limit, using the Crowl-Louvar [1] correlation. Uses molecular formula only. The upper flammability limit of a gas is air is:

$$C_m H_x O_y + z O_2 \rightarrow m CO_2 + \frac{x}{2} H_2O$$

$$\text{UFL} = \frac{3.5}{4.76m + 1.19x - 2.38y + 1}$$

Parameters

atoms [dict] Dictionary of atoms and atom counts

Returns

UFL [float] Upper flammability limit, mole fraction

Notes

Coefficient of 3.5 taken from [2]

References

[1], [2]

Examples

Hexane, example from [1], lit. 7.5 %

```
>>> Crowl_Louvar_UFL({'H': 14, 'C': 6})
0.07572479446127219
```

1.25.10 Mixture Flammability Limit

chemicals.safety.**fire_mixing**(*ys*, *FLs*)

Le Chatelier's mixing rule for lower and upper flammability limits of mixtures of gases.

Parameters

ys [list[float]] Normalized mole fractions of all flammable components in a gas, [-]

FLs [list[float]] Lower or upper flammability limits for each flammable component in a gas, [-]

Returns

FL [float] Lower or upper flammability limit of a gas, [-]

Notes

This equation has a higher accuracy for lower flammability limits than upper flammability limits. Some sources recommend not using it for upper flammability limits.

References

[1]

Examples

Sample problems from [1] for the lower and upper flammability limit.

```
>>> fire_mixing(ys=normalize([0.0024, 0.0061, 0.0015]), FLs=[.012, .053, .031])
0.02751172136637642
```

```
>>> fire_mixing(ys=normalize([0.0024, 0.0061, 0.0015]), FLs=[.075, .15, .32])
0.12927551844869378
```

1.25.11 Utility Methods

chemicals.safety.**ppmv_to_mgm3**(*ppmv*, *MW*, *T=298.15*, *P=101325.0*)

Converts a concentration in ppmv to units of mg/m^3. Used in industrial toxicology.

$$\frac{mg}{m^3} = \frac{ppmv \cdot P}{RT} \cdot \frac{MW}{1000}$$

Parameters

ppmv [float] Concentration of a component in a gas mixure [parts per million, volumetric]

MW [float] Molecular weight of the trace gas [g/mol]

T [float, optional] Temperature of the gas at which the ppmv is reported, [K]

P [float, optional] Pressure of the gas at which the ppmv is reported, [Pa]

Returns

mgm3 [float] Concentration of a substance in an ideal gas mixture [mg/m^3]

Notes

The term P/(RT)/1000 converts to 0.040874 at STP. Its inverse is reported as 24.45 in [1].

References

[1]

Examples

```
>>> ppmv_to_mgm3(1.0, 40.0)
1.6349617809430446
```

chemicals.safety.**mgm3_to_ppmv**(*mgm3*, *MW*, *T=298.15*, *P=101325.0*)

Converts a concentration in mg/m^3 to units of ppmv. Used in industrial toxicology.

$$ppmv = \frac{1000RT}{MW \cdot P} \cdot \frac{mg}{m^3}$$

Parameters

mgm3 [float] Concentration of a substance in an ideal gas mixture [mg/m^3]

MW [float] Molecular weight of the trace gas [g/mol]

T [float, optional] Temperature of the gas at which the ppmv is reported, [K]

P [float, optional] Pressure of the gas at which the ppmv is reported, [Pa]

Returns

ppmv [float] Concentration of a component in a gas mixure [parts per million, volumetric]

Notes

The term P/(RT)/1000 converts to 0.040874 at STP. Its inverse is reported as 24.45 in [1].

References

[1]

Examples

```
>>> mgm3_to_ppmv(1.635, 40.0)
1.0000233761164334
```

chemicals.safety.**NFPA_30_classification**(*T_flash*, *Tb=None*, *Psat_100F=None*)

Classify a chemical's flammability/combustibility according to the NFPA 30 standard Flammable and Combustible Liquids Code.

Class IA: Flash Point < 73°F; Boiling Point < 100°F Class IB: Flash Point < 73°F; 100°F <= Boiling Point Class IC: 73°F <= Flash Point < 100°F Class II: 100°F <= Flash Point < 140°F Class IIIA: 140°F <= Flash Point < 200°F Class IIIB: 200°F <= Flash Point

Class I liquids are designated as flammable; class II and II liquids are designated as combustible.

Parameters

T_flash [float] Flash point (closed-cup method, adjusted for sea level), [K]

Tb [float, optional] Normal boiling point (needed to classify IA and IB liquids), [K]

Psat_100F [float, optional] Vapor pressure at 100°F (needed to classify IA and IB liquids), [K]

Returns

classification [str] One of 'IA', 'IB', 'IC', 'II', 'IIIA', 'IIIB', [-]

Notes

Only one of *Tb* or *Psat_100F* is needed.

Class 'IA' also includes unstable liquids.

References

[1]

Examples

Ethylene oxide

```
>>> NFPA_30_classification(253.15, 283.55)
'IA'
```

Butyl alcohol

```
>>> NFPA_30_classification(308.15)
'IC'
```

1.26 Solubility (chemicals.solubility)

This module contains various solubility calculation routines and a Henry's law coefficient converter.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.26.1 Henry's Law

chemicals.solubility.**Henry_pressure**(*T*, *A*, *B=0.0*, *C=0.0*, *D=0.0*, *E=0.0*, *F=0.0*)

Calculates Henry's law constant as a function of temperature according to the SI units of *Pa* and using a common temperature dependence as used in many process simulation applications.

Only the *A* parameter is required - which has no temperature dependence when used by itself. As the model is exponential, a sufficiently high temperature may cause an OverflowError. A negative temperature (or just low, if fit poorly) may cause a math domain error.

$$H_{12} = \exp\left(A_{12} + \frac{B_{12}}{T} + C_{12}\ln(T) + D_{12}T + \frac{E_{12}}{T^2}\right)$$

Parameters

T [float] Temperature, [K]

A-F [float] Parameter for the equation; chemical and property specific [-]

Returns

H12 [float] Henry's constant [Pa]

Notes

Add 11.51292 to the *A* constant if it is said to provide units of *bar*, so that it provides units of *Pa* instead.

The *F* parameter is not often included in models. It is rare to fit all parameters.

References

[1]

Examples

Random test example.

```
>>> Henry_pressure(300.0, A=15.0, B=300.0, C=.04, D=1e-3, E=1e2, F=1e-5)
37105004.47898146
```

chemicals.solubility.**Henry_pressure_mixture**(*Hs*, *weights=None*, *zs=None*)

Mixing rule for Henry's law components. Applies a logarithmic average to all solvent components and mole fractions. Optionally, weight factors can be provided instead of using mole fractions - only specify one of them.

A common weight factor is using volume fractions of powers of them, or using critical volumes.

Parameters

Hs [list[float or None]] Henry's law constant between each gas and the solvent (None for other solvents of gases without parameters available), [Pa]

weights [list[float], optional] Weight factors, [-]

zs [list[float]] Mole fractions of all species in phase, [-]

Returns

H [value] Henry's law constant for the gas in the liquid phase, [-]

Notes

The default weight factor formulation is from [1].

References

[1]

Examples

```
>>> Henry_pressure_mixture([1072330.36341, 744479.751106, None], zs=[.48, .48, .04])
893492.1611602883
```

chemicals.solubility.**Henry_converter**(*val*, *old_scale*, *new_scale*, *rhom=None*, *MW=None*)

Converts Henry's law constant for a gas with respect to a solvent from one scale to another.

There are many scales, but it is recommemed to operate in the scale of *SI* - which returns a value with units *Pa*, and directly gets used in place of vapor pressure inside a flash calculation. This removes the complexity of Henry's law, avoiding possible simplication in favor of use with other thermodynamic models.

Only some scales require the molecular weight and the molar density of the solvent. Values for water, the most common solute, are 55344.59 mol/m^3 at STP and 18.01528 g/mol.

Parameters

val [float] Henry's law constant, various units

old_scale [str] String representing the scale that *val* is in originally.

new_scale [str] String representing the scale that *val* should be converted to.

Returns

result [float] Input *val* converted from *old_scale* to *new_scale*, various units

Notes

The valid scales for this function are any of the following:

('Hcp', 'mol/(m^3*Pa)', 'M/atm', 'Hcc', 'mol/(kg*Pa)', 'Hbp', 'mol/(kg*atm)', 'Hxp', '1/atm', 'alpha', 'bunsen coefficient', 'KHpx', 'atm', 'm^3*Pa/mol', 'KHpc', 'm^3*atm/mol', 'KHcc', 'SI')

References

[1]

Examples

```
>>> Henry_converter(1.2e-5, old_scale='Hcp', new_scale='SI', rhom=55344.59,
... MW=18.01528)
4612049166.666666
```

```
>>> Henry_converter(0.0297475, old_scale='Hcc', new_scale='KHcc',
... rhom=55344.59, MW=18.01528)
33.61627027481301
```

chemicals.solubility.**Henry_constants**(*lnHenry_matrix*, *zs*, *henry_components*, *skip_zero=True*, *Hs=None*)

Calculate the Henry's law constants for a list of components, only some of which are henry's law following components (solutes) and the rest that are solvents. The empirical mixing rule from [1] is used as follows:

$$H_i = \exp\left(\frac{\sum_{\text{j=solvent}} z_j \ln H_{i,j}}{\sum_{\text{j=solvent}} z_j}\right)$$

Parameters

lnHenry_matrix [list[list[float]]] Henry's law constants between every species; 0.0 for non-applicable solvents, [log(Pa)]

zs [list[float]] Mole fractions of all species in phase; this can be mass or volume fractions as well, [-]

henry_components [list[bool]] Whether or not each component is a henry's law solvent or not, [-]

skip_zero [bool] If true, if parameters are missing from a solvent-solute pair, that pair will not be counted as part of the solvent fraction. If false, the calculation proceeds and the solubility is underestimated. Missing parameters are assumed from the value of *lnHenry_matrix* being 0, [-]

Hs [list[float], optional] Henry's law constants for each component; 0 for non-henry components (input array), [Pa]

Returns

Hs [list[float]] Henry's law constants for each component; 0 for non-henry components, [Pa]

References

[1]

Examples

Oxygen and methane in water:

```
>>> lnHenry_matrix = [[0.0, 0.0, 0.0], [22.13581843104147, 0.0, 0.0], [22.
↪239038459475733, 0.0, 0.0]]
>>> Henry_constants(lnHenry_matrix, [0.8, 0.15, 0.05], [False, True, True], True)
[0.0, 4106424071.093, 4552937470.331]
```

chemicals.solubility.**dHenry_constants_dT**(*lnHenry_matrix*, *dlnHenry_matrix_dT*, *zs*, *henry_components*, *skip_zero=True*, *dH_dTs=None*)

Calculate the first temperature derivative of Henry's law constants for a list of components, only some of which are henry's law following components (solutes) and the rest that are solvents. The empirical mixing rule from [1] is used as follows:

Parameters

lnHenry_matrix [list[list[float]]] Henry's law constants between every species; 0.0 for non-applicable solvents, [log(Pa)]

dlnHenry_matrix_dT [list[list[float]]] First temperature derivative of Henry's law constants between every species; 0.0 for non-applicable solvents, [log(Pa)/K]

zs [list[float]] Mole fractions of all species in phase; this can be mass or volume fractions as well, [-]

henry_components [list[bool]] Whether or not each component is a henry's law solvent or not, [-]

skip_zero [bool] If true, if parameters are missing from a solvent-solute pair, that pair will not be counted as part of the solvent fraction. If false, the calculation proceeds and the solubility is underestimated. Missing parameters are assumed from the value of *lnHenry_matrix* being 0, [-]

dH_dTs [list[float], optional] First temperature derivative of Henry's law constants for each component; 0 for non-henry components (input array), [Pa/K]

Returns

dH_dTs [list[float]] First temperature derivative of Henry's law constants for each component; 0 for non-henry components, [Pa/K]

References

[1]

Examples

Oxygen and methane in water:

```
>>> lnHenry_matrix = [[0.0, 0.0, 0.0], [22.13581843104147, 0.0, 0.0], [22.
↪239038459475733, 0.0, 0.0]]
>>> dlnHenry_matrix_dT = [[0.0, 0.0, 0.0], [0.017113988888888904, 0.0, 0.0], [0.
↪015461911111111101, 0.0, 0.0]]
>>> dHenry_constants_dT(lnHenry_matrix, dlnHenry_matrix_dT, [0.8, 0.15, 0.05],␣
↪[False, True, True], True)
[0.0, 70277295.92576516, 70397114.46071726]
```

chemicals.solubility.**d2Henry_constants_dT2**(*lnHenry_matrix*, *dlnHenry_matrix_dT*, *d2lnHenry_matrix_dT2*, *zs*, *henry_components*, *skip_zero=True*, *d2H_dT2s=None*)

Calculate the second temperature derivative of Henry's law constants for a list of components, only some of which are henry's law following components (solutes) and the rest that are solvents. The empirical mixing rule from [1] is used as follows:

Parameters

lnHenry_matrix [list[list[float]]] Henry's law constants between every species; 0.0 for non-applicable solvents, [log(Pa)]

dlnHenry_matrix_dT [list[list[float]]] First temperature derivative of Henry's law constants between every species; 0.0 for non-applicable solvents, [log(Pa)/K]

d2lnHenry_matrix_dT2 [list[list[float]]] Second temperature derivative of Henry's law constants between every species; 0.0 for non-applicable solvents, [log(Pa)/K^2]

zs [list[float]] Mole fractions of all species in phase; this can be mass or volume fractions as well, [-]

henry_components [list[bool]] Whether or not each component is a henry's law solvent or not, [-]

skip_zero [bool] If true, if parameters are missing from a solvent-solute pair, that pair will not be counted as part of the solvent fraction. If false, the calculation proceeds and the solubility is underestimated. Missing parameters are assumed from the value of *lnHenry_matrix* being 0, [-]

d2H_dT2s [list[float], optional] Second temperature derivative of Henry's law constants for each component; 0 for non-henry components (input array), [Pa/K^2]

Returns

d2H_dT2s [list[float]] Second temperature derivative of Henry's law constants for each component; 0 for non-henry components, [Pa/K^2]

References

[1]

Examples

Oxygen and methane in water:

```
>>> lnHenry_matrix = [[0.0, 0.0, 0.0], [22.13581843104147, 0.0, 0.0], [22.
↪239038459475733, 0.0, 0.0]]
>>> dlnHenry_matrix_dT = [[0.0, 0.0, 0.0], [0.017113988888888904, 0.0, 0.0], [0.
↪015461911111111101, 0.0, 0.0]]
>>> d2lnHenry_matrix_dT2 = [[0.0, 0.0, 0.0], [-0.0004070325925925928, 0.0, 0.0], [-
↪0.00034016518518518524, 0.0, 0.0]]
>>> d2Henry_constants_dT2(lnHenry_matrix, dlnHenry_matrix_dT, d2lnHenry_matrix_dT2,␣
↪[0.8, 0.15, 0.05], [False, True, True], True)
[0.0, -468723.574327235, -460276.89146166]
```

1.26.2 Utility functions

chemicals.solubility.**solubility_eutectic**(*T*, *Tm*, *Hm*, *Cpl=0*, *Cps=0*, *gamma=1*)

Returns the maximum solubility of a solute in a solvent.

$$\ln x_i^L \gamma_i^L = \frac{\Delta H_{m,i}}{RT}\left(1 - \frac{T}{T_{m,i}}\right) - \frac{\Delta C_{p,i}(T_{m,i} - T)}{RT} + \frac{\Delta C_{p,i}}{R}\ln\frac{T_m}{T}$$

$$\Delta C_{p,i} = C_{p,i}^L - C_{p,i}^S$$

Parameters

T [float] Temperature of the system [K]

Tm [float] Melting temperature of the solute [K]

Hm [float] Heat of melting at the melting temperature of the solute [J/mol]

Cpl [float, optional] Molar heat capacity of the solute as a liquid [J/mol/K]

Cps: float, optional Molar heat capacity of the solute as a solid [J/mol/K]

gamma [float, optional] Activity coefficient of the solute as a liquid [-]

Returns

x [float] Mole fraction of solute at maximum solubility [-]

Notes

gamma is of the solute in liquid phase

References

[1]

Examples

From [1], matching example

```
>>> solubility_eutectic(T=260., Tm=278.68, Hm=9952., Cpl=0, Cps=0, gamma=3.0176)
0.243400713
```

chemicals.solubility.**solubility_parameter**(*T*, *Hvapm*, *Vml*)

This function handles the calculation of a chemical's solubility parameter. Calculation is a function of temperature, but is not always presented as such. *Hvapm*, *Vml*, *T* are required.

$$\delta = \sqrt{\frac{\Delta H_{vap} - RT}{V_m}}$$

Parameters

T [float] Temperature of the fluid [k]

Hvapm [float] Heat of vaporization [J/mol/K]

Vml [float] Specific volume of the liquid [m^3/mol]

Returns

delta [float] Solubility parameter, [Pa^0.5]

Notes

Undefined past the critical point. For convenience, if Hvap is not defined, an error is not raised; None is returned instead. Also for convenience, if Hvapm is less than RT, None is returned to avoid taking the root of a negative number.

This parameter is often given in units of cal/ml, which is 2045.48 times smaller than the value returned here.

References

[1]

Examples

Pentane at STP

```
>>> solubility_parameter(T=298.2, Hvapm=26403.3, Vml=0.000116055)
14357.68128600315
```

chemicals.solubility.**Tm_depression_eutectic**(*Tm*, *Hm*, *x=None*, *M=None*, *MW=None*)

Returns the freezing point depression caused by a solute in a solvent. Can use either the mole fraction of the solute or its molality and the molecular weight of the solvent. Assumes ideal system behavior.

$$\Delta T_m = \frac{RT_m^2 x}{\Delta H_m}$$

$$\Delta T_m = \frac{RT_m^2(MW)M}{1000\Delta H_m}$$

Parameters

Tm [float] Melting temperature of the solute [K]

Hm [float] Heat of melting at the melting temperature of the solute [J/mol]

x [float, optional] Mole fraction of the solute [-]

M [float, optional] Molality [mol/kg]

MW: float, optional Molecular weight of the solvent [g/mol]

Returns

dTm [float] Freezing point depression [K]

Notes

MW is the molecular weight of the solvent. M is the molality of the solute.

References

[1]

Examples

From [1], matching example.

```
>>> Tm_depression_eutectic(353.35, 19110, .02)
1.0864598583150
```

1.27 ITS Temperature Scales (chemicals.temperature)

This module contains functionality for converting between the temperature scales ITS-90, ITS-76, ITS-68, ITS-48, and ITS-27. These historical temperature scales can deviate quite a bit from modern temperature measurements! It is important to convert old measurements of temperature to their modern equivalent.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

- *Conversion functions*

1.27.1 Conversion functions

chemicals.temperature.**T_converter**(*T*, *current*, *desired*)

Converts the a temperature reading made in any of the scales 'ITS-90', 'ITS-68','ITS-48', 'ITS-76', or 'ITS-27' to any of the other scales. Not all temperature ranges can be converted to other ranges; for instance, 'ITS-76' is purely for low temperatures, and 5 K on it has no conversion to 'ITS-90' or any other scale. Both a conversion to ITS-90 and to the desired scale must be possible for the conversion to occur. The conversion uses cubic spline interpolation.

ITS-68 conversion is valid from 14 K to 4300 K. ITS-48 conversion is valid from 93.15 K to 4273.15 K ITS-76 conversion is valid from 5 K to 27 K. ITS-27 is valid from 903.15 K to 4273.15 k.

Parameters

T [float] Temperature, on *current* scale [K]

current [str] String representing the scale T is in, 'ITS-90', 'ITS-68', 'ITS-48', 'ITS-76', or 'ITS-27'.

desired [str] String representing the scale T will be returned in, 'ITS-90', 'ITS-68', 'ITS-48', 'ITS-76', or 'ITS-27'.

Returns

T [float] Temperature, on scale *desired* [K]

Notes

Because the conversion is performed by spline functions, a re-conversion of a value will not yield exactly the original value. However, it is quite close.

The use of splines is quite quick (20 micro seconds/calculation). While just a spline for one-way conversion could be used, a numerical solver would have to be used to obtain an exact result for the reverse conversion. This was found to take approximately 1 ms/calculation, depending on the region.

References

[1], [2]

Examples

```
>>> T_converter(500, 'ITS-68', 'ITS-48')
499.9470092992346
```

chemicals.temperature.**ITS90_68_difference**(*T*)

Calculates the difference between ITS-90 and ITS-68 scales using a series of models listed in [1], [2], and [3].

The temperature difference is given by the following equations:

From 13.8 K to 73.15 K:

$$T_{90} - T_{68} = a_0 + \sum_{i=1}^{12} a_i[(T_{90}/K - 40)/40]^i$$

From 83.8 K to 903.75 K:

$$T_{90} - T_{68} = \sum_{i=1}^{8} b_i[(T_{90}/K - 273.15)/630]^i$$

From 903.75 K to 1337.33 K:

$$T_{90} - T_{68} = \sum_{i=0}^{5} c_i [T_{90}/^\circ C]^i$$

Above 1337.33 K:

$$T_{90} - T_{68} = -1.398 \cdot 10^{-7} \left(\frac{T_{90}}{K}\right)^2$$

Parameters

T [float] Temperature, ITS-90, or approximately ITS-68 [K]

Returns

dT [float] Temperature, difference between ITS-90 and ITS-68 at T [K]

Notes

The conversion is straightforward when T90 is known. Theoretically, the model should be solved numerically to convert the reverse way. However, according to [4], the difference is under 0.05 mK from 73.15 K to 903.15 K, and under 0.26 mK up to 1337.33 K.

For temperatures under 13.8 K, no conversion is performed.

The first set of coefficients is:

-0.005903, 0.008174, -0.061924, -0.193388, 1.490793, 1.252347, -9.835868, 1.411912, 25.277595, -19.183815, -18.437089, 27.000895, -8.716324.

The second set of coefficients is:

0, -0.148759, -0.267408, 1.08076, 1.269056, -4.089591, -1.871251, 7.438081, -3.536296.

The third set of coefficients is:

7.8687209E1, -4.7135991E-1, 1.0954715E-3, -1.2357884E-6, 6.7736583E-10, -1.4458081E-13.

These last coefficients use the temperature in degrees Celcius. A slightly older model used the following coefficients but a different equation over the same range:

-0.00317, -0.97737, 1.2559, 2.03295, -5.91887, -3.23561, 7.23364, 5.04151.

The model for these coefficients was:

$$T_{90} - T_{68} = c_0 + \sum_{i=1}^{7} c_i [(T_{90}/K - 1173.15)/300]^i$$

For temperatures larger than several thousand K, the differences have no meaning and grows quadratically.

References

[1], [2], [3], [4]

Examples

```
>>> ITS90_68_difference(1000.)
0.01231818956580355
```

1.28 Thermal Conductivity (chemicals.thermal_conductivity)

This module contains various thermal conductivity estimation routines, dataframes of fit coefficients, and mixing rules.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.28.1 Pure Low Pressure Liquid Correlations

chemicals.thermal_conductivity.**Sheffy_Johnson**(*T*, *MW*, *Tm*)

Calculate the thermal conductivity of a liquid as a function of temperature using the Sheffy-Johnson (1961) method. Requires Temperature, molecular weight, and melting point.

$$k = 1.951\frac{1 - 0.00126(T - T_m)}{T_m^{0.216} MW^{0.3}}$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Tm [float] Melting point of the fluid [K]

Returns

kl [float] Thermal conductivity of the fluid, W/m/k

Notes

The origin of this equation has been challenging to trace. It is presently unknown, and untested.

References

[1]

Examples

```
>>> Sheffy_Johnson(300, 47, 280)
0.17740150413112193
```

chemicals.thermal_conductivity.**Sato_Riedel**(*T*, *MW*, *Tb*, *Tc*)

Calculate the thermal conductivity of a liquid as a function of temperature using the CSP method of Sato-Riedel [1], [2], published in Reid [3]. Requires temperature, molecular weight, and boiling and critical temperatures.

$$k = \frac{1.1053}{\sqrt{MW}} \frac{3 + 20(1 - T_r)^{2/3}}{3 + 20(1 - T_{br})^{2/3}}$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

This equation has a complicated history. It is proposed by Reid [3]. Limited accuracy should be expected. Uncheecked.

References

[1], [2], [3]

Examples

```
>>> Sato_Riedel(300, 47, 390, 520)
0.21037692461337687
```

chemicals.thermal_conductivity.**Lakshmi_Prasad**(*T*, *MW*)

Estimates thermal conductivity of pure liquids as a function of temperature using a reference fluid approach. Low accuracy but quick. Developed using several organic fluids.

$$\lambda = 0.0655 - 0.0005T + \frac{1.3855 - 0.00197T}{MW^{0.5}}$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

This equation returns negative numbers at high T sometimes. This equation is one of those implemented by DDBST. If this results in a negative thermal conductivity, no value is returned.

References

[1]

Examples

```
>>> Lakshmi_Prasad(273.15, 100)
0.013664450
```

chemicals.thermal_conductivity.**Gharagheizi_liquid**(*T*, *MW*, *Tb*, *Pc*, *omega*)

Estimates the thermal conductivity of a liquid as a function of temperature using the CSP method of Gharagheizi [1]. A convoluted method claiming high-accuracy and using only statistically significant variable following analalysis.

Requires temperature, molecular weight, boiling temperature and critical pressure and acentric factor.

$$k = 10^{-4}\left[10\omega + 2P_c - 2T + 4 + 1.908(T_b + \frac{1.009B^2}{MW^2}) + \frac{3.9287MW^4}{B^4} + \frac{A}{B^8}\right]$$

$$A = 3.8588MW^8(1.0045B + 6.5152MW - 8.9756)$$

$$B = 16.0407MW + 2T_b - 27.9074$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Tb [float] Boiling temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor of the fluid [-]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

Pressure is internally converted into bar, as used in the original equation.

This equation was derived with 19000 points representing 1640 unique compounds.

References

[1]

Examples

```
>>> Gharagheizi_liquid(300, 40, 350, 1E6, 0.27)
0.2171113029534838
```

chemicals.thermal_conductivity.**Nicola_original**(*T*, *MW*, *Tc*, *omega*, *Hfus*)

Estimates the thermal conductivity of a liquid as a function of temperature using the CSP method of Nicola [1]. A simpler but long method claiming high-accuracy and using only statistically significant variable following analalysis.

Requires temperature, molecular weight, critical temperature, acentric factor and the heat of vaporization.

$$\frac{\lambda}{1\text{Wm/K}} = -0.5694 - 0.1436T_r + 5.4893\times 10^{-10}\frac{\Delta_{fus}H}{\text{kmol/J}} + 0.0508\omega + \left(\frac{1\text{kg/kmol}}{MW}\right)^{0.0622}$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Tc [float] Critical temperature of the fluid [K]

omega [float] Acentric factor of the fluid [-]

Hfus [float] Heat of fusion of the fluid [J/mol]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

A weird statistical correlation. Recent and yet to be reviewed. This correlation has been superceded by the author's later work. Hfus is internally converted to be in J/kmol.

References

[1]

Examples

```
>>> Nicola_original(300, 142.3, 611.7, 0.49, 201853)
0.2305018632230984
```

chemicals.thermal_conductivity.**Nicola**(*T*, *MW*, *Tc*, *Pc*, *omega*)

Estimates the thermal conductivity of a liquid as a function of temperature using the CSP method of [1]. A statistically derived equation using any correlated terms.

Requires temperature, molecular weight, critical temperature and pressure, and acentric factor.

$$\frac{\lambda}{0.5147 W/m/K} = -0.2537 T_r + \frac{0.0017 Pc}{\text{bar}} + 0.1501\omega + \left(\frac{1}{MW}\right)^{-0.2999}$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor of the fluid [-]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

A statistical correlation. A revision of an original correlation.

References

[1]

Examples

```
>>> Nicola(300, 142.3, 611.7, 2110000.0, 0.49)
0.10863821554584034
```

chemicals.thermal_conductivity.**Bahadori_liquid**(*T*, *MW*)

Estimates the thermal conductivity of parafin liquid hydrocarbons. Fits their data well, and is useful as only MW is required. X is the Molecular weight, and Y the temperature.

$$K = a + bY + CY^2 + dY^3$$

$$a = A_1 + B_1 X + C_1 X^2 + D_1 X^3$$

$$b = A_2 + B_2 X + C_2 X^2 + D_2 X^3$$

$$c = A_3 + B_3 X + C_3 X^2 + D_3 X^3$$

$$d = A_4 + B_4 X + C_4 X^2 + D_4 X^3$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

The accuracy of this equation has not been reviewed.

References

[1]

Examples

Data point from [1].

```
>>> Bahadori_liquid(273.15, 170)
0.1427427810827268
```

chemicals.thermal_conductivity.**kl_Mersmann_Kind**(*T*, *MW*, *Tc*, *Vc*, *na*)

Estimates the thermal conductivity of organic liquid substances according to the method of [1].

$$\lambda^* = \frac{\lambda \cdot V_c^{2/3} \cdot T_c \cdot \mathrm{MW}^{0.5}}{(k \cdot T_c)^{1.5} \cdot N_A^{7/6}}$$

$$\lambda^* = \frac{2}{3}\left(n_a + 40\sqrt{1 - T_r}\right)$$

Parameters

T [float] Temperature of the fluid [K]

MW [float] Molecular weight of the fluid [g/mol]

Tc [float] Critical temperature of the fluid [K]

Vc [float] Critical volume of the fluid [m^3/mol]

na [float] Number of atoms in the molecule, [-]

Returns

kl [float] Estimated liquid thermal conductivity [W/m/k]

Notes

In the equation, all quantities must be in SI units but N_A is in a kmol basis and Vc is in units of (m^3/kmol); this is converted internally.

References

[1]

Examples

Dodecane at 400 K:

```
>>> kl_Mersmann_Kind(400, 170.33484, 658.0,
... 0.000754, 38)
0.0895271829899285
```

1.28.2 Pure High Pressure Liquid Correlations

chemicals.thermal_conductivity.**DIPPR9G**(*T*, *P*, *Tc*, *Pc*, *kl*)

Adjustes for pressure the thermal conductivity of a liquid using an emperical formula based on [1], but as given in [2].

$$k = k^* \left[0.98 + 0.0079 P_r T_r^{1.4} + 0.63 T_r^{1.2} \left(\frac{P_r}{30 + P_r}\right)\right]$$

Parameters

T [float] Temperature of fluid [K]

P [float] Pressure of fluid [Pa]

Tc: float Critical point of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

kl [float] Thermal conductivity of liquid at 1 atm or saturation, [W/m/K]

Returns

kl_dense [float] Thermal conductivity of liquid at P, [W/m/K]

Notes

This equation is entrely dimensionless; all dimensions cancel. The original source has not been reviewed.

This is DIPPR Procedure 9G: Method for the Thermal Conductivity of Pure Nonhydrocarbon Liquids at High Pressures

References

[1], [2]

Examples

From [2], for butyl acetate.

```
>>> DIPPR9G(515.05, 3.92E7, 579.15, 3.212E6, 7.085E-2)
0.0864419738671184
```

chemicals.thermal_conductivity.**Missenard**(*T*, *P*, *Tc*, *Pc*, *kl*)

Adjustes for pressure the thermal conductivity of a liquid using an emperical formula based on [1], but as given in [2].

$$\frac{k}{k^*} = 1 + QP_r^{0.7}$$

Parameters

T [float] Temperature of fluid [K]

P [float] Pressure of fluid [Pa]

Tc: float Critical point of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

kl [float] Thermal conductivity of liquid at 1 atm or saturation, [W/m/K]

Returns

kl_dense [float] Thermal conductivity of liquid at P, [W/m/K]

Notes

This equation is entirely dimensionless; all dimensions cancel. An interpolation routine is used here from tabulated values of Q. The original source has not been reviewed.

References

[1], [2]

Examples

Example from [2], toluene; matches.

```
>>> Missenard(304., 6330E5, 591.8, 41E5, 0.129)
0.2198375777069657
```

1.28.3 Liquid Mixing Rules

chemicals.thermal_conductivity.**DIPPR9H**(*ws*, *ks*)

Calculates thermal conductivity of a liquid mixture according to mixing rules in [1] and also in [2].

$$\lambda_m = \left(\sum_i w_i \lambda_i^{-2} \right)^{-1/2}$$

This is also called the Vredeveld (1973) equation. A review in [3] finds this the best model on average. However, they did caution that in some cases a linear mole-fraction mixing rule performs better. This equation according to Poling [1] should not be used if some components have thermal conductivities more than twice other components. They also say this should not be used with water.

Parameters

ws [float] Mass fractions of components

ks [float] Liquid thermal conductivites of all components, [W/m/K]

Returns

kl [float] Thermal conductivity of liquid mixture, [W/m/K]

Notes

This equation is entirely dimensionless; all dimensions cancel. The example is from [2]; all results agree. The original source has not been reviewed.

DIPPR Procedure 9H: Method for the Thermal Conductivity of Nonaqueous Liquid Mixtures

Average deviations of 3%. for 118 nonaqueous systems with 817 data points. Max deviation 20%. According to DIPPR.

In some sources, this equation is given with the molecular weights included:

$$\lambda_m^{-2} = \frac{\sum_i z_i MW_i \lambda_i^{-2}}{\sum_i z_i MW_i}$$

References

[1], [2], [3]

Examples

```
>>> DIPPR9H([0.258, 0.742], [0.1692, 0.1528])
0.15657104706719646
```

chemicals.thermal_conductivity.**DIPPR9I**(*zs*, *Vms*, *ks*)

Calculates thermal conductivity of a liquid mixture according to mixing rules in [1]. This is recommended in [2] for aqueous and nonaqueous systems.

$$k_{mix} = \sum_i \sum_j \phi_i \phi_j k_{i,j}$$

$$k_{i,j} = \frac{2}{\frac{1}{k_i} + \frac{1}{k_j}}$$

$$\phi_i = \frac{z_i V_{m,i}}{\sum_j^n z_j V_{m,j}}$$

Parameters

zs [list[float]] Mole fractions of components, [-]

Vms [list[float]] Molar volumes of each component, [m^3/mol]

ks [float] Liquid thermal conductivites of all components, [W/m/K]

Returns

kl [float] Thermal conductivity of liquid mixture, [W/m/K]

Notes

This equation is entirely dimensionless; all dimensions cancel. The example is from [2]; all results agree.

[2] found average deviations of 4-6% for 118 nonaqueous systems and 15 aqueous systems at atmospheric pressure, with a maximum deviation of 33%.

The computational complexity here is N^2, with a division present in the inner loop.

References

[1], [2]

Examples

```
>>> DIPPR9I(zs=[.682, .318], Vms=[1.723e-2, 7.338e-2], ks=[.6037, .1628])
0.25397430656658937
```

chemicals.thermal_conductivity.**Filippov**(*ws*, *ks*)

Calculates thermal conductivity of a binary liquid mixture according to mixing rules in [2] as found in [1].

$$\lambda_m = w_1\lambda_1 + w_2\lambda_2 - 0.72 w_1 w_2(\lambda_2 - \lambda_1)$$

Parameters

ws [float] Mass fractions of components

ks [float] Liquid thermal conductivites of all components, [W/m/K]

Returns

kl [float] Thermal conductivity of liquid mixture, [W/m/K]

Notes

This equation is entirely dimensionless; all dimensions cancel. The original source has not been reviewed. Only useful for binary mixtures.

References

[1], [2]

Examples

```
>>> Filippov([0.258, 0.742], [0.1692, 0.1528])
0.15929167628799998
```

1.28.4 Pure Low Pressure Gas Correlations

chemicals.thermal_conductivity.**Eucken**(*MW*, *Cvm*, *mu*)

Estimates the thermal conductivity of a gas as a function of temperature using the CSP method of Eucken [1].

$$\frac{\lambda MW}{\eta C_v} = 1 + \frac{9/4}{C_v/R}$$

Parameters

MW [float] Molecular weight of the gas [g/mol]

Cvm [float] Molar contant volume heat capacity of the gas [J/mol/K]

mu [float] Gas viscosity [Pa*s]

Returns

kg [float] Estimated gas thermal conductivity [W/m/k]

Notes

Temperature dependence is introduced via heat capacity and viscosity. A theoretical equation. No original author located. MW internally converted to kg/g-mol.

References

[1]

Examples

2-methylbutane at low pressure, 373.15 K. Mathes calculation in [1].

```
>>> Eucken(MW=72.151, Cvm=135.9, mu=8.77E-6)
0.018792645058456698
```

chemicals.thermal_conductivity.**Eucken_modified**(*MW*, *Cvm*, *mu*)

Estimates the thermal conductivity of a gas as a function of temperature using the Modified CSP method of Eucken [1].

$$\frac{\lambda MW}{\eta C_v} = 1.32 + \frac{1.77}{C_v/R}$$

Parameters

MW [float] Molecular weight of the gas [g/mol]

Cvm [float] Molar contant volume heat capacity of the gas [J/mol/K]

mu [float] Gas viscosity [Pa*s]

Returns

kg [float] Estimated gas thermal conductivity [W/m/k]

Notes

Temperature dependence is introduced via heat capacity and viscosity. A theoretical equation. No original author located. MW internally converted to kg/g-mol.

References

[1]

Examples

2-methylbutane at low pressure, 373.15 K. Mathes calculation in [1].

```
>>> Eucken_modified(MW=72.151, Cvm=135.9, mu=8.77E-6)
0.02359353760551249
```

chemicals.thermal_conductivity.**DIPPR9B**(*T*, *MW*, *Cvm*, *mu*, *Tc=None*, *chemtype=None*)

Calculates the thermal conductivity of a gas using one of several emperical equations developed in [1], [2], and presented in [3].

For monoatomic gases:

$$k = 2.5\frac{\eta C_v}{MW}$$

For linear molecules:

$$k = \frac{\eta}{MW}\left(1.30C_v + 14644.00 - \frac{2928.80}{T_r}\right)$$

For nonlinear molecules:

$$k = \frac{\eta}{MW}(1.15C_v + 16903.36)$$

Parameters

T [float] Temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

MW [float] Molwcular weight of fluid [g/mol]

Cvm [float] Molar heat capacity at constant volume of fluid, [J/mol/K]

mu [float] Viscosity of gas, [Pa*s]

Returns

k_g [float] Thermal conductivity of gas, [W/m/k]

Notes

Tested with DIPPR values. Cvm is internally converted to J/kmol/K.

References

[1], [2], [3]

Examples

CO:

```
>>> DIPPR9B(200., 28.01, 20.826, 1.277E-5, 132.92, chemtype='linear')
0.01813208676438415
```

chemicals.thermal_conductivity.**Chung**(*T*, *MW*, *Tc*, *omega*, *Cvm*, *mu*)

Estimates the thermal conductivity of a gas as a function of temperature using the CSP method of Chung [1].

$$\frac{\lambda MW}{\eta C_v} = \frac{3.75\Psi}{C_v/R}$$

$$\Psi = 1 + \alpha\left\{[0.215 + 0.28288\alpha - 1.061\beta + 0.26665Z]/[0.6366 + \beta Z + 1.061\alpha\beta]\right\}$$

$$\alpha = \frac{C_v}{R} - 1.5$$

$$\beta = 0.7862 - 0.7109\omega + 1.3168\omega^2$$

$$Z = 2 + 10.5T_r^2$$

Parameters

T [float] Temperature of the gas [K]

MW [float] Molecular weight of the gas [g/mol]

Tc [float] Critical temperature of the gas [K]

omega [float] Acentric factor of the gas [-]

Cvm [float] Molar contant volume heat capacity of the gas [J/mol/K]

mu [float] Gas viscosity [Pa*s]

Returns

kg [float] Estimated gas thermal conductivity [W/m/k]

Notes

MW internally converted to kg/g-mol.

References

[1], [2]

Examples

2-methylbutane at low pressure, 373.15 K. Mathes calculation in [2].

```
>>> Chung(T=373.15, MW=72.151, Tc=460.4, omega=0.227, Cvm=135.9, mu=8.77E-6)
0.023015653797111124
```

chemicals.thermal_conductivity.**Eli_Hanley**(*T*, *MW*, *Tc*, *Vc*, *Zc*, *omega*, *Cvm*)

Estimates the thermal conductivity of a gas as a function of temperature using the reference fluid method of Eli and Hanley [1] as shown in [2].

$$\lambda = \lambda^* + \frac{\eta^*}{MW}(1.32)\left(C_v - \frac{3R}{2}\right)$$

$$Tr = \min(Tr, 2)$$

$$\theta = 1 + (\omega - 0.011)\left(0.56553 - 0.86276\ln Tr - \frac{0.69852}{Tr}\right)$$

$$\psi = [1 + (\omega - 0.011)(0.38560 - 1.1617\ln Tr)]\frac{0.288}{Z_c}$$

$$f = \frac{T_c}{190.4}\theta$$

$$h = \frac{V_c}{9.92E-5}\psi$$

$$T_0 = T/f$$

$$\eta_0^*(T_0) = \sum_{n=1}^{9} C_n T_0^{(n-4)/3}$$

$$\theta_0 = 1944\eta_0$$

$$\lambda^* = \lambda_0 H$$

$$\eta^* = \eta_0^* H \frac{MW}{16.04}$$

$$H = \left(\frac{16.04}{MW}\right)^{0.5} f^{0.5}/h^{2/3}$$

Parameters

T [float] Temperature of the gas [K]

MW [float] Molecular weight of the gas [g/mol]

Tc [float] Critical temperature of the gas [K]

Vc [float] Critical volume of the gas [m^3/mol]

Zc [float] Critical compressibility of the gas []

omega [float] Acentric factor of the gas [-]

Cvm [float] Molar contant volume heat capacity of the gas [J/mol/K]

Returns

kg [float] Estimated gas thermal conductivity [W/m/k]

Notes

Reference fluid is Methane. MW internally converted to kg/g-mol.

References

[1], [2]

Examples

2-methylbutane at low pressure, 373.15 K. Matches calculation in [2].

```
>>> Eli_Hanley(T=373.15, MW=72.151, Tc=460.4, Vc=3.06E-4, Zc=0.267,
... omega=0.227, Cvm=135.9)
0.02247951724513664
```

chemicals.thermal_conductivity.**Gharagheizi_gas**(*T*, *MW*, *Tb*, *Pc*, *omega*)

Estimates the thermal conductivity of a gas as a function of temperature using the CSP method of Gharagheizi [1]. A convoluted method claiming high-accuracy and using only statistically significant variable following analalysis.

Requires temperature, molecular weight, boiling temperature and critical pressure and acentric factor.

$$k = 7.9505 \times 10^{-4} + 3.989 \times 10^{-5} T - 5.419 \times 10^{-}5 MW + 3.989 \times 10^{-5} A$$

$$A = \frac{\left(2\omega + T - \frac{(2\omega + 3.2825)T}{T_b} + 3.2825\right)}{0.1 M P_c T} \times (3.9752\omega + 0.1 P_c + 1.9876 B + 6.5243)^2$$

Parameters

T [float] Temperature of the fluid [K]

MW: float Molecular weight of the fluid [g/mol]

Tb [float] Boiling temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor of the fluid [-]

Returns

kg [float] Estimated gas thermal conductivity [W/m/k]

Notes

Pressure is internally converted into 10*kPa but author used correlation with kPa; overall, errors have been corrected in the presentation of the formula.

This equation was derived with 15927 points and 1574 compounds. Example value from [1] is the first point in the supportinf info, for CH4.

References

[1]

Examples

```
>>> Gharagheizi_gas(580., 16.04246, 111.66, 4599000.0, 0.0115478000)
0.09594861261873211
```

chemicals.thermal_conductivity.**Bahadori_gas**(*T*, *MW*)

Estimates the thermal conductivity of hydrocarbons gases at low P. Fits their data well, and is useful as only MW is required. Y is the Molecular weight, and X the temperature.

$$K = a + bY + CY^2 + dY^3$$

$$a = A_1 + B_1 X + C_1 X^2 + D_1 X^3$$

$$b = A_2 + B_2 X + C_2 X^2 + D_2 X^3$$

$$c = A_3 + B_3 X + C_3 X^2 + D_3 X^3$$

$$d = A_4 + B_4 X + C_4 X^2 + D_4 X^3$$

Parameters

T [float] Temperature of the gas [K]

MW [float] Molecular weight of the gas [g/mol]

Returns

kg [float] Estimated gas thermal conductivity [W/m/k]

Notes

The accuracy of this equation has not been reviewed.

References

[1]

Examples

```
>>> Bahadori_gas(40+273.15, 20.0) # Point from article
0.03196816533787329
```

1.28.5 Pure High Pressure Gas Correlations

chemicals.thermal_conductivity.**Stiel_Thodos_dense**(*T*, *MW*, *Tc*, *Pc*, *Vc*, *Zc*, *Vm*, *kg*)

Estimates the thermal conductivity of a gas at high pressure as a function of temperature using difference method of Stiel and Thodos [1] as shown in [2].

if $\rho_r < 0.5$:

$$(\lambda - \lambda^\circ)\Gamma Z_c^5 = 1.22 \times 10^{-2}[\exp(0.535\rho_r) - 1]$$

if $0.5 < \rho_r < 2.0$:

$$(\lambda - \lambda^\circ)\Gamma Z_c^5 = 1.22 \times 10^{-2}[\exp(0.535\rho_r) - 1]$$

if $2 < \rho_r < 2.8$:

$$(\lambda - \lambda^\circ)\Gamma Z_c^5 = 1.22 \times 10^{-2}[\exp(0.535\rho_r) - 1]$$

$$\Gamma = 210\left(\frac{T_c MW^3}{P_c^4}\right)^{1/6}$$

Parameters

T [float] Temperature of the gas [K]

MW [float] Molecular weight of the gas [g/mol]

Tc [float] Critical temperature of the gas [K]

Pc [float] Critical pressure of the gas [Pa]

Vc [float] Critical volume of the gas [m^3/mol]

Zc [float] Critical compressibility of the gas [-]

Vm [float] Molar volume of the gas at T and P [m^3/mol]

kg [float] Low-pressure gas thermal conductivity [W/m/k]

Returns

kg [float] Estimated dense gas thermal conductivity [W/m/k]

Notes

Pc is internally converted to bar.

References

[1], [2]

Examples

```
>>> Stiel_Thodos_dense(T=378.15, MW=44.013, Tc=309.6, Pc=72.4E5,
... Vc=97.4E-6, Zc=0.274, Vm=144E-6, kg=2.34E-2)
0.041245574404863684
```

chemicals.thermal_conductivity.**Eli_Hanley_dense**(*T*, *MW*, *Tc*, *Vc*, *Zc*, *omega*, *Cvm*, *Vm*)

Estimates the thermal conductivity of a gas at high pressure as a function of temperature using the reference fluid method of Eli and Hanley [1] as shown in [2].

$$Tr = min(Tr, 2)$$

$$Vr = min(Vr, 2)$$

$$f = \frac{T_c}{190.4}\theta$$

$$h = \frac{V_c}{9.92E-5}\psi$$

$$T_0 = T/f$$

$$\rho_0 = \frac{16.04}{V}h$$

$$\theta = 1 + (\omega - 0.011)\left(0.09057 - 0.86276\ln Tr + \left(0.31664 - \frac{0.46568}{Tr}\right)(V_r - 0.5)\right)$$

$$\psi = [1 + (\omega - 0.011)(0.39490(V_r - 1.02355) - 0.93281(V_r - 0.75464)\ln T_r]\frac{0.288}{Z_c}$$

$$\lambda_1 = 1944\eta_0$$

$$\lambda_2 = \left\{b_1 + b_2\left[b_3 - \ln\left(\frac{T_0}{b_4}\right)\right]^2\right\}\rho_0$$

$$\lambda_3 = \exp\left(a_1 + \frac{a_2}{T_0}\right)\left\{\exp[(a_3 + \frac{a_4}{T_0^{1.5}})\rho_0^{0.1} + (\frac{\rho_0}{0.1617} - 1)\rho_0^{0.5}(a_5 + \frac{a_6}{T_0} + \frac{a_7}{T_0^2})] - 1\right\}$$

$$\lambda^{**} = [\lambda_1 + \lambda_2 + \lambda_3]H$$

$$H = \left(\frac{16.04}{MW}\right)^{0.5} f^{0.5}/h^{2/3}$$

$$X = \left\{\left[1 - \frac{T}{f}\left(\frac{df}{dT}\right)_v\right]\frac{0.288}{Z_c}\right\}^{1.5}$$

$$\left(\frac{df}{dT}\right)_v = \frac{T_c}{190.4}\left(\frac{d\theta}{dT}\right)_v$$

$$\left(\frac{d\theta}{dT}\right)_v = (\omega - 0.011)\left[\frac{-0.86276}{T} + (V_r - 0.5)\frac{0.46568T_c}{T^2}\right]$$

Parameters

T [float] Temperature of the gas [K]

MW [float] Molecular weight of the gas [g/mol]

Tc [float] Critical temperature of the gas [K]

Vc [float] Critical volume of the gas [m^3/mol]

Zc [float] Critical compressibility of the gas [-]

omega [float] Acentric factor of the gas [-]

Cvm [float] Molar contant volume heat capacity of the gas [J/mol/K]

Vm [float] Volume of the gas at T and P [m^3/mol]

Returns

kg [float] Estimated dense gas thermal conductivity [W/m/k]

Notes

Reference fluid is Methane. MW internally converted to kg/g-mol.

References

[1], [2]

Examples

```
>>> Eli_Hanley_dense(T=473., MW=42.081, Tc=364.9, Vc=1.81E-4, Zc=0.274,
... omega=0.144, Cvm=82.70, Vm=1.721E-4)
0.06038475754109959
```

chemicals.thermal_conductivity.**Chung_dense**(*T*, *MW*, *Tc*, *Vc*, *omega*, *Cvm*, *Vm*, *mu*, *dipole*, *association=0.0*)

Estimates the thermal conductivity of a gas at high pressure as a function of temperature using the reference fluid method of Chung [1] as shown in [2].

$$\lambda = \frac{31.2\eta^\circ\Psi}{M'}(G_2^{-1} + B_6 y) + qB_7 y^2 T_r^{1/2} G_2$$

$$\Psi = 1 + \alpha\left\{[0.215 + 0.28288\alpha - 1.061\beta + 0.26665Z]/[0.6366 + \beta Z + 1.061\alpha\beta]\right\}$$

$$\alpha = \frac{C_v}{R} - 1.5$$

$$\beta = 0.7862 - 0.7109\omega + 1.3168\omega^2$$

$$Z = 2 + 10.5T_r^2$$

$$q = 3.586 \times 10^{-3}(T_c/M')^{1/2}/V_c^{2/3}$$

$$y = \frac{V_c}{6V}$$

$$G_1 = \frac{1 - 0.5y}{(1-y)^3}$$

$$G_2 = \frac{(B_1/y)[1 - \exp(-B_4 y)] + B_2 G_1 \exp(B_5 y) + B_3 G_1}{B_1 B_4 + B_2 + B_3}$$

$$B_i = a_i + b_i\omega + c_i\mu_r^4 + d_i\kappa$$

Parameters

T [float] Temperature of the gas [K]

MW [float] Molecular weight of the gas [g/mol]

Tc [float] Critical temperature of the gas [K]

Vc [float] Critical volume of the gas [m^3/mol]

omega [float] Acentric factor of the gas [-]

Cvm [float] Molar contant volume heat capacity of the gas [J/mol/K]

Vm [float] Molar volume of the gas at T and P [m^3/mol]

mu [float] Low-pressure gas viscosity [Pa*s]

dipole [float] Dipole moment [debye]

association [float, optional] Association factor [-]

Returns

kg [float] Estimated dense gas thermal conductivity [W/m/k]

Notes

MW internally converted to kg/g-mol. Vm internally converted to mL/mol. [1] is not the latest form as presented in [1]. Association factor is assumed 0. Relates to the polarity of the gas.

Coefficients as follows:

ais = [2.4166E+0, -5.0924E-1, 6.6107E+0, 1.4543E+1, 7.9274E-1, -5.8634E+0, 9.1089E+1]

bis = [7.4824E-1, -1.5094E+0, 5.6207E+0, -8.9139E+0, 8.2019E-1, 1.2801E+1, 1.2811E+2]

cis = [-9.1858E-1, -4.9991E+1, 6.4760E+1, -5.6379E+0, -6.9369E-1, 9.5893E+0, -5.4217E+1]

dis = [1.2172E+2, 6.9983E+1, 2.7039E+1, 7.4344E+1, 6.3173E+0, 6.5529E+1, 5.2381E+2]

References

[1], [2]

Examples

```
>>> Chung_dense(T=473., MW=42.081, Tc=364.9, Vc=184.6E-6, omega=0.142,
... Cvm=82.67, Vm=172.1E-6, mu=134E-7, dipole=0.4)
0.06160569232570781
```

1.28.6 Gas Mixing Rules

chemicals.thermal_conductivity.**Lindsay_Bromley**(*T*, *ys*, *ks*, *mus*, *Tbs*, *MWs*)

Calculates thermal conductivity of a gas mixture according to mixing rules in [1] and also in [2]. It is significantly more complicated than other kinetic theory models.

$$k = \sum_i \frac{y_i k_i}{\sum_j y_i A_{ij}}$$

$$A_{ij} = \frac{1}{4}\left\{1 + \left[\frac{\eta_i}{\eta_j}\left(\frac{MW_j}{MW_i}\right)^{0.75}\left(\frac{T+S_i}{T+S_j}\right)\right]^{0.5}\right\}^2\left(\frac{T+S_{ij}}{T+S_i}\right)$$

$$S_{ij} = S_{ji} = (S_i S_j)^{0.5}$$

$$S_i = 1.5T_b$$

Parameters

T [float] Temperature of gas [K]

ys [float] Mole fractions of gas components

ks [float] Gas thermal conductivites of all components, [W/m/K]

mus [float] Gas viscosities of all components, [Pa*s]

Tbs [float] Boiling points of all components, [K]

MWs [float] Molecular weights of all components, [g/mol]

Returns

kg [float] Thermal conductivity of gas mixture, [W/m/K]

Notes

This equation is entirely dimensionless; all dimensions cancel. The example is from [2]; all results agree. The original source has not been reviewed.

DIPPR Procedure 9D: Method for the Thermal Conductivity of Gas Mixtures

Average deviations of 4-5% for 77 binary mixtures reviewed in [2], from 1342 points; also six ternary mixtures (70 points); max deviation observed was 40%. (DIPPR)

References

[1], [2], [3]

Examples

```
>>> Lindsay_Bromley(323.15, [0.23, 0.77], [1.939E-2, 1.231E-2], [1.002E-5, 1.015E-
↪5], [248.31, 248.93], [46.07, 50.49])
0.013902644179693132
```

chemicals.thermal_conductivity.**Wassiljewa_Herning_Zipperer**(*zs*, *ks*, *MWs*, *MW_roots=None*)

Calculates thermal conductivity of a gas mixture according to the kinetic theory expression of Wassiljewa with the interaction term from the Herning-Zipperer expression. This is also used for the prediction of gas mixture viscosity.

$$k = \sum \frac{y_i k_i}{\sum y_i A_{ij}}$$

$$A_{ij} = \left(\frac{MW_j}{MW_i}\right)^{0.5}$$

Parameters

zs [float] Mole fractions of gas components, [-]

ks [float] gas thermal conductivites of all components, [W/m/K]

MWs [float] Molecular weights of all components, [g/mol]

MW_roots [float, optional] Square roots of molecular weights of all components; speeds up the calculation if provided, [g^0.5/mol^0.5]

Returns

kg [float] Thermal conductivity of gas mixture, [W/m/K]

Notes

This equation is entirely dimensionless; all dimensions cancel.

References

[1]

Examples

```
>>> Wassiljewa_Herning_Zipperer(zs=[.1, .4, .5], ks=[1.002E-5, 1.15E-5, 2e-5],␣
↪MWs=[40.0, 50.0, 60.0])
1.5861181979916883e-05
```

1.28.7 Correlations for Specific Substances

chemicals.thermal_conductivity.**k_IAPWS**(*T*, *rho*, *Cp=None*, *Cv=None*, *mu=None*, *drho_dP=None*, *drho_dP_Tr=None*)

Calculate the thermal conductivity of water or steam according to the 2011 IAPWS [1] formulation. Critical enhancement is ignored unless parameters for it are provided.

$$\bar{\lambda} = \bar{\lambda}_0 \times \bar{\lambda}_1(\bar{T}, \bar{\rho}) + \bar{\lambda}_2(\bar{T}, \bar{\rho})$$

$$\bar{\lambda}_0 = \frac{\sqrt{\bar{T}}}{\sum_{k=0}^{4} \frac{L_k}{\bar{T}^k}}$$

$$\bar{\lambda}_1(\bar{T}, \bar{\rho}) = \exp\left[\bar{\rho} \sum_{i=0}^{4} \left(\left(\frac{1}{\bar{T}} - 1\right)^i \sum_{j=0}^{5} L_{ij} (\bar{\rho} - 1)^j\right)\right]$$

$$\bar{\lambda}_2 = \Gamma \frac{\bar{\rho} \bar{c}_p \bar{T}}{\bar{\mu}} Z(y)$$

$$Z(y) = \frac{2}{\pi y}\left\{\left[(1 - \kappa^{-1}) \arctan(y) + \kappa^{-1} y\right] - \left[1 - \exp\left(\frac{-1}{y^{-1} + y^{-2}/3\bar{\rho}^2}\right)\right]\right\}$$

$$y = \bar{q}_D \xi(\bar{T}, \bar{\rho})$$

$$\xi = \xi_0 \left(\frac{\Delta\bar{\chi}}{\Gamma_0}\right)^{\nu/\gamma}$$

$$\Delta\bar{\chi}(\bar{T},\bar{\rho}) = \bar{\rho}\left[\zeta(\bar{T},\bar{\rho}) - \zeta(\bar{T}_R,\bar{\rho})\frac{\bar{T}_R}{\bar{T}}\right]$$

$$\zeta = \left(\frac{\partial\bar{\rho}}{\partial\bar{p}}\right)_{\bar{T}}$$

Parameters

T [float] Temperature water [K]

rho [float] Density of water [kg/m^3]

Cp [float, optional] Constant pressure heat capacity of water, [J/kg/K]

Cv [float, optional] Constant volume heat capacity of water, [J/kg/K]

mu [float, optional] Viscosity of water, [Pa*s]

drho_dP [float, optional] Partial derivative of density with respect to pressure at constant temperature, [kg/m^3/Pa]

drho_dP_Tr [float, optional] Partial derivative of density with respect to pressure at constant temperature (at the reference temperature (970.644 K) and the actual density of water); will be calculated from the industrial formulation fit if omitted, [kg/m^3/Pa]

Returns

k [float] Thermal condiuctivity, [W/m/K]

Notes

Gamma = 177.8514;

qd = 0.4E-9;

nu = 0.630;

gamma = 1.239;

zeta0 = 0.13E-9;

Gamma0 = 0.06;

TRC = 1.5

The formulation uses the industrial variant of the critical enhancement. It matches to 5E-6 relative tolerance at the check temperature, and should match even closer outside it.

References

[1]

Examples

```
>>> k_IAPWS(647.35, 750.)
0.5976194153179502
```

Region 1, test 1, from MPEI, exact match:

```
>>> k_IAPWS(T=620., rho=613.227777440324, Cp=7634.337046792,
... Cv=3037.934412104, mu=70.905106751524E-6, drho_dP=5.209378197916E-6)
0.48148519510200044
```

Full scientific calculation:

```
>>> from chemicals.iapws import iapws95_properties, iapws95_P, iapws95_Tc
>>> from chemicals.viscosity import mu_IAPWS
>>> T, P = 298.15, 1e5
>>> rho, _, _, _, Cv, Cp, _, _, _, _, drho_dP = iapws95_properties(T, P)
>>> P_ref = iapws95_P(1.5*iapws95_Tc, rho)
>>> _, _, _, _, _, _, _, _, _, _, drho_dP_Tr = iapws95_properties(1.5*iapws95_Tc, P_
↪ref)
>>> mu = mu_IAPWS(T, rho, drho_dP, drho_dP_Tr)
>>> k_IAPWS(T, rho, Cp, Cv, mu, drho_dP, drho_dP_Tr)
0.60651532815
```

chemicals.thermal_conductivity.**k_air_lemmon**(*T*, *rho*, *Cp=None*, *Cv=None*, *drho_dP=None*, *drho_dP_Tr=None*, *mu=None*)

Calculate the thermal conductivity of air using the Lemmon and Jacobsen (2004) [1] formulation. The critical enhancement term is ignored unless all the rquired parameters for it are provided.

$$\lambda = \lambda^0(T) + \lambda^r(\tau, \delta) + \lambda^c(\tau, \delta)$$

$$\lambda^0 = N_1 \left[\frac{\eta^0(T)}{1\mu\text{Pa}\cdot\text{s}}\right] + N_2\tau^{t_2} + N_3\tau^{t_3}$$

$$\lambda^r = \sum_{i=4}^{n} N_i \tau^{t_i} \delta^{d_i} \exp(-\gamma_i \delta^{l_i})$$

$$\lambda^c = \rho C_p \frac{kR_0T}{6\pi\xi\cdot\eta(T,\rho)}\left(\tilde{\Omega} - \tilde{\Omega}_0\right)$$

$$\tilde{\Omega} = \frac{2}{\pi}\left[\left(\frac{C_p - C_v}{C_p}\right)\tan^{-1}(\xi/q_D) + \frac{C_v}{C_p}(\xi/q_D)\right]$$

$$\tilde{\Omega}_0 = \frac{2}{\pi}\left\{1 - \exp\left[\frac{-1}{q_D/\xi + 1/3(\xi/q_D)^2(\rho_c/\rho)^2}\right]\right\}$$

$$\xi = \xi_0\left[\frac{\tilde{\chi}(T,\rho) - \tilde{\chi}(T_{ref},\rho)\frac{T_{ref}}{T}}{\Gamma}\right]^{\nu/\gamma}$$

$$\tilde{\chi}(T,\rho) = \frac{P_c\rho}{\rho_c^2}\left(\frac{\partial\rho}{\partial P}\right)_T$$

Parameters

T [float] Temperature air [K]

rho [float] Molar density of air [mol/m^3]

Cp [float, optional] Molar constant pressure heat capacity of air, [J/mol/K]

Cv [float, optional] Molar constant volume heat capacity of air, [J/mol/K]

mu [float, optional] Viscosity of air, [Pa*s]

drho_dP [float, optional] Partial derivative of density with respect to pressure at constant temperature, [mol/m^3/Pa]

drho_dP_Tr [float, optional] Partial derivative of density with respect to pressure at constant temperature (at the reference temperature (265.262 K) and the actual density of air), [mol/m^3/Pa]

Returns

k [float] Thermal condiuctivity of air, [W/m/K]

Notes

The constnts are as follows:

Ni = [1.308, 1.405, -1.036, 8.743, 14.76, -16.62, 3.793, -6.142, -0.3778]

ti = [None, -1.1, -0.3, 0.1, 0.0, 0.5, 2.7, 0.3, 1.3]

di = [None, None, None, 1, 2, 3, 7, 7, 11]

li = [None, None, None, 0, 0, 2, 2, 2, 2]

gammai = [None, None, None, 0, 0, 1, 1, 1, 1]

R0 = 1.01; Pc = 3.78502E6 Pa; xi0 = 0.11E-9 nm; qd = 0.31E-9 nm; Tc = 132.6312 K (actually the maxcondentherm); T_ref = 265.262 (2Tc rounded differently); rhoc = 10447.7 mol/m^3 (actually the maxcondentherm); k = 1.380658E-23 J/K; nu = 0.63 and gamma = 1.2415, sigma = 0.36, MW = 28.9586 g/mol.

References

[1]

Examples

Basic calculation at 300 K and approximately 1 bar:

```
>>> k_air_lemmon(300, 40.0)
0.0263839695044
```

Calculation near critical point:

```
>>> k_air_lemmon(132.64, 10400, 2137.078854678728, 35.24316159996235, 0.
↪07417878614315769, 0.00035919027241528256, 1.7762253265868595e-05)
0.07562307234760
```

1.28.8 Fit Correlations

chemicals.thermal_conductivity.**PPDS8**(*T*, *Tc*, *a0*, *a1*, *a2*, *a3*)

Calculate the thermal conductivity of a liquid using the 4-term *tau* polynomial developed by the PPDS and named PPDS equation 8.

$$k_l = a_0 \left(1 + \sum_i^3 a_i \tau^{i/3}\right)$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a0 [float] Coefficient, [-]

a1 [float] Coefficient, [-]

a2 [float] Coefficient, [-]

a3 [float] Coefficient, [-]

Returns

k [float] Low pressure liquid thermal conductivity, [W/(m*K)]

References

[1]

Examples

Sample coefficients for benzene in [1], at 500 K:

```
>>> PPDS8(T=500.0, Tc=562.05, a0=0.0641126, a1=0.61057, a2=-1.72442, a3=3.94394)
0.08536381765218425
```

chemicals.thermal_conductivity.**PPDS3**(*T*, *Tc*, *a1*, *a2*, *a3*)

Calculate the thermal conductivity of a low-pressure gas using the 3-term *Tr* polynomial developed by the PPDS and named PPDS equation 3.

$$k_g = \sqrt{T_r} \left(\sum_{i=1}^3 \frac{a_i}{T_r^i}\right)^{-1}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a1 [float] Coefficient, [-]

a2 [float] Coefficient, [-]

a3 [float] Coefficient, [-]

Returns

k [float] Low pressure gas thermal conductivity, [W/(m*K)]

References

[1]

Examples

Sample coefficients for pentane in [1], at 400 K:

```
>>> PPDS3(T=400.0, Tc=470.008, a1=11.6366, a2=25.1191, a3=-7.21674)
0.0251734811601927
```

chemicals.thermal_conductivity.**Chemsep_16**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculate the thermal conductivity of a low-pressure liquid using the 5-term T exponential polynomial found in ChemSep.

$$k_l = A + \exp\left(\frac{B}{T} + C + DT + ET^2\right)$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [W/(m*K)]

B [float] Coefficient, [K]

C [float] Coefficient, [-]

D [float] Coefficient, [1/K]

E [float] Coefficient, [1/K^2]

Returns

k [float] Low pressure liquid thermal conductivity, [W/(m*K)]

References

[1]

Examples

Sample coefficients for liquid thermal conductivity of n-hexane in [1], at 300 K:

```
>>> Chemsep_16(300.0, -0.12682, -1.5015, -1.0467, -0.00088709, -9.3679E-07)
0.11924904787869
```

1.28.9 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.thermal_conductivity.**k_data_Perrys_8E_2_315**
Data from [1] with *chemicals.dippr.EQ100* coefficients for liquids.

chemicals.thermal_conductivity.**k_data_Perrys_8E_2_314**
Data from [1] with *chemicals.dippr.EQ102* coefficients for gases.

chemicals.thermal_conductivity.**k_data_VDI_PPDS_9**
Data from [2] with polynomial coefficients for liquids.

chemicals.thermal_conductivity.**k_data_VDI_PPDS_10**
Data from [2] with polynomial coefficients for gases.

```
In [1]: import chemicals

In [2]: chemicals.thermal_conductivity.k_data_Perrys_8E_2_315
Out[2]:
                         Chemical       C1        C2  ...   C5    Tmin    Tmax
CAS                                                   ...
50-00-0               Formaldehyde  0.37329 -0.000650  ...  0.0  204.00  234.00
55-21-0                  Benzamide  0.28485 -0.000252  ...  0.0  403.00  563.15
56-23-5       Carbon tetrachloride  0.15890 -0.000199  ...  0.0  250.33  349.79
57-55-6      1,2-Propylene glycol   0.21520 -0.000050  ...  0.0  213.15  460.75
60-29-7             Diethyl ether  0.24950 -0.000407  ...  0.0  156.85  433.15
...                           ...      ...       ...  ...  ...     ...     ...
10028-15-6                 Ozone    0.17483  0.000753  ...  0.0   77.35  161.85
10035-10-6       Hydrogen bromide  0.23400 -0.000464  ...  0.0  185.15  290.62
10102-43-9           Nitric oxide  0.18780  0.001029  ...  0.0  110.00  176.40
13511-13-2     Propenylcyclohexene  0.18310 -0.000203  ...  0.0  199.00  431.65
132259-10-0                  Air    0.28472 -0.001739  ...  0.0   75.00  125.00

[340 rows x 8 columns]

In [3]: chemicals.thermal_conductivity.k_data_Perrys_8E_2_314
Out[3]:
                         Chemical         C1  ...    Tmin     Tmax
CAS                                           ...
50-00-0               Formaldehyde  44.847000  ...  254.05   994.05
55-21-0                  Benzamide   0.025389  ...  563.15  1000.00
56-23-5       Carbon tetrachloride   0.000166  ...  349.79  1000.00
57-55-6      1,2-Propylene glycol    0.000167  ...  460.75  1000.00
60-29-7             Diethyl ether  -0.004489  ...  200.00   600.00
...                           ...        ...  ...     ...      ...
10028-15-6                 Ozone     0.004315  ...  161.85  1000.00
10035-10-6       Hydrogen bromide    0.000497  ...  206.45   600.00
10102-43-9           Nitric oxide    0.000410  ...  121.38   750.00
13511-13-2     Propenylcyclohexene    0.000102  ...  431.65  1000.00
132259-10-0                  Air     0.000314  ...   70.00  2000.00

[345 rows x 7 columns]

In [4]: chemicals.thermal_conductivity.k_data_VDI_PPDS_9
```

(continues on next page)

(continued from previous page)

```
Out[4]:
                        Chemical       A  ...             D             E
CAS                                       ...
50-00-0              Formaldehyde  0.3834  ...  1.156000e-09 -2.638000e-12
56-23-5      Carbon tetrachloride  0.1509  ... -7.100000e-11  3.980000e-13
56-81-5                  Glycerol  0.2562  ... -1.050000e-10  1.020000e-13
60-29-7             Diethyl ether  0.2499  ... -8.600000e-11  7.300000e-14
62-53-3                   Aniline  0.2365  ... -3.600000e-11  2.100000e-14
...                           ...     ...  ...           ...           ...
10097-32-2                Bromine -0.1426  ...  2.690200e-08 -1.774400e-11
10102-43-9            Nitric oxide  0.2268  ... -1.993600e-08  1.448400e-11
10102-44-0       Nitrogen dioxide  0.3147  ...  2.620000e-10 -6.980000e-13
10544-72-6    Dinitrogentetroxide  0.1864  ... -5.440000e-10  1.509000e-12
132259-10-0                   Air -0.0006  ...  1.114335e-06 -2.670110e-09

[271 rows x 6 columns]

In [5]: chemicals.thermal_conductivity.k_data_VDI_PPDS_10
Out[5]:
                        Chemical             A  ...             D             E
CAS                                             ...
50-00-0              Formaldehyde  8.870000e-04  ...  0.000000e+00  0.000000e+00
56-23-5      Carbon tetrachloride -2.101000e-03  ...  0.000000e+00  0.000000e+00
56-81-5                  Glycerol -9.158000e-03  ...  0.000000e+00  0.000000e+00
60-29-7             Diethyl ether -5.130000e-04  ...  0.000000e+00  0.000000e+00
62-53-3                   Aniline -9.960000e-03  ...  0.000000e+00  0.000000e+00
...                           ...           ...  ...           ...           ...
10097-32-2                Bromine  5.455000e-03  ...  0.000000e+00  0.000000e+00
10102-43-9            Nitric oxide  1.440000e-04  ...  0.000000e+00  0.000000e+00
10102-44-0       Nitrogen dioxide  6.608500e-02  ...  0.000000e+00  0.000000e+00
10544-72-6    Dinitrogentetroxide  1.460000e-09  ...  0.000000e+00  0.000000e+00
132259-10-0                   Air -9.080000e-04  ...  5.696400e-11 -1.563100e-14

[275 rows x 6 columns]
```

1.29 Triple Point (chemicals.triple)

This module contains lookup functions for a chemical's triple temperature and pressure. The triple temperature is the unique co-existence point between a pure chemicals's solid, gas, and liquid state.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

- *Triple Temperature*
- *Triple Pressure*

1.29.1 Triple Temperature

chemicals.triple.**Tt**(*CASRN*, *method=None*)

This function handles the retrieval of a chemical's triple temperature. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Returns data from [1], [2] or [3], or a chemical's melting point if available.

Parameters

CASRN [str] CASRN [-]

Returns

Tt [float] Triple point temperature, [K].

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *Tt_all_methods*.

See also:

Tt_methods

Notes

Median difference between melting points and triple points is 0.02 K. Accordingly, this should be more than good enough for engineering applications.

The data in [1] is originally on the ITS-68 temperature scale, but was converted to ITS-90. The numbers were rounded to 6 decimal places arbitrarily and the conversion was performed with this library.

References

[1], [2], [3]

Examples

Ammonia

```
>>> Tt('7664-41-7')
195.49
```

chemicals.triple.**Tt_methods**(*CASRN*)

Return all methods available to obtain the triple temperature for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Tt with the given inputs.

See also:

Tt

chemicals.triple.**Tt_all_methods = ('HEOS', 'STAVELEY', 'WEBBOOK', 'MELTING')**
Tuple of method name keys. See the *Tt* for the actual references

1.29.2 Triple Pressure

chemicals.triple.**Pt**(*CASRN*, *method=None*)
This function handles the retrieval of a chemical's triple pressure. Lookup is based on CASRNs. Will automatically select a data source to use if no method is provided; returns None if the data is not available.

Returns data from [1], [2], or [3].

This function does not implement it but it is also possible to calculate the vapor pressure at the triple temperature from a vapor pressure correlation, if data is available; note most Antoine-type correlations do not extrapolate well to this low of a pressure.

Parameters

CASRN [str] CASRN [-]

Returns

Pt [float] Triple point pressure, [Pa]

Other Parameters

method [string, optional] A string for the method name to use, as defined in the variable, *Pt_all_methods*.

See also:

Pt_methods

References

[1], [2], [3]

Examples

Ammonia

```
>>> Pt('7664-41-7')
6053.386
```

chemicals.triple.**Pt_methods**(*CASRN*)
Return all methods available to obtain the Pt for the desired chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

methods [list[str]] Methods which can be used to obtain the Pt with the given inputs.

See also:

Pt

chemicals.triple.**Pt_all_methods = ('HEOS', 'STAVELEY', 'WEBBOOK')**
Tuple of method name keys. See the *Pt* for the actual references

1.30 Utilities (chemicals.utils)

This module contains miscellaneous functions which may be useful. This includes definitions of some chemical properties, and conversions between others.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

chemicals.utils.**API_to_SG**(*API*)
Calculates specific gravity of a liquid given its API, as shown in [1].

$$\text{SG at } 60^\circ\text{F} = \frac{141.5}{\text{API gravity} + 131.5}$$

Parameters

API [float] API of the fluid [-]

Returns

SG [float] Specific gravity of the fluid at 60 degrees Farenheight [-]

Notes

Defined only at 60 degrees Fahrenheit.

References

[1]

Examples

```
>>> API_to_SG(60.62)
0.7365188423901728
```

chemicals.utils.**API_to_rho**(*API*, *rho_ref=999.0170824078306*)
Calculates mass density of a liquid given its API, as shown in [1].

$$\rho\ 60^\circ\text{F} = \frac{141.5\rho_{ref}}{\text{API} + 131.5}$$

Parameters

API [float] API of the fluid [-]

rho_ref [float, optional] Density of the reference substance, [kg/m^3]

Returns

rho [float] Mass density the fluid at 60 degrees Farenheight [kg/m^3]

Notes

Defined only at 60 degrees Fahrenheit.

References

[1]

Examples

```
>>> API_to_rho(rho_to_API(820))
820.0
```

chemicals.utils.**Cp_minus_Cv**(*T*, *dP_dT*, *dP_dV*)

Calculate the difference between a real gas's constant-pressure heat capacity and constant-volume heat capacity, as given in [1], [2], and [3]. The required derivatives should be calculated with an equation of state.

$$C_p - C_v = -T \left(\frac{\partial P}{\partial T}\right)_V^2 / \left(\frac{\partial P}{\partial V}\right)_T$$

Parameters

T [float] Temperature of fluid [K]

dP_dT [float] Derivative of *P* with respect to *T*, [Pa/K]

dP_dV [float] Derivative of *P* with respect to *V*, [Pa*mol/m^3]

Returns

Cp_minus_Cv [float] Cp - Cv for a real gas, [J/mol/K]

Notes

Equivalent expressions are:

$$C_p - C_v = -T \left(\frac{\partial V}{\partial T}\right)_P^2 / \left(\frac{\partial V}{\partial P}\right)_T$$

$$C_p - C_v = T \left(\frac{\partial P}{\partial T}\right) \left(\frac{\partial V}{\partial T}\right)$$

Note that these are not second derivatives, only first derivatives, some of which are squared.

References

[1], [2], [3]

Examples

Calculated for hexane from the PR EOS at 299 K and 1 MPa (liquid):

```
>>> Cp_minus_Cv(299, 582232.475794113, -3665180614672.253)
27.654681381642394
```

chemicals.utils.**Joule_Thomson**(*T*, *V*, *Cp*, *dV_dT=None*, *beta=None*)

Calculate a real fluid's Joule Thomson coefficient. The required derivative should be calculated with an equation of state, and *Cp* is the real fluid versions. This can either be calculated with *dV_dT* directly, or with *beta* if it is already known.

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Parameters

T [float] Temperature of fluid, [K]

V [float] Molar volume of fluid, [m^3/mol]

Cp [float] Real fluid heat capacity at constant pressure, [J/mol/K]

dV_dT [float, optional] Derivative of V with respect to T, [m^3/mol/K]

beta [float, optional] Isobaric coefficient of a thermal expansion, [1/K]

Returns

mu_JT [float] Joule-Thomson coefficient [K/Pa]

References

[1], [2]

Examples

Example from [2]:

```
>>> Joule_Thomson(T=390, V=0.00229754, Cp=153.235, dV_dT=1.226396e-05)
1.6219560805299 05e-05
```

chemicals.utils.**Parachor**(*MW*, *rhol*, *rhog*, *sigma*)

Calculate Parachor for a pure species, using its density in the liquid and gas phases, surface tension, and molecular weight.

$$P = \frac{\sigma^{0.25} MW}{\rho_L - \rho_V}$$

Parameters

MW [float] Molecular weight, [g/mol]

rhol [float] Liquid density [kg/m^3]

rhog [float] Gas density [kg/m^3]

sigma [float] Surface tension, [N/m]

Returns

P [float] Parachor, [N^0.25*m^2.75/mol]

Notes

To convert the output of this function to units of [mN^0.25*m^2.75/kmol], multiply by 5623.4132519.

Values in group contribution tables for Parachor are often listed as dimensionless, in which they are multiplied by 5623413 and the appropriate units to make them dimensionless.

References

[1], [2], [3]

Examples

Calculating Parachor from a known surface tension for methyl isobutyl ketone at 293.15 K

```
>>> Parachor(100.15888, 800.8088185536124, 4.97865317223119, 0.02672166960656005)
5.088443542210164e-05
```

Converting to the *dimensionless* form:

```
>>> 5623413*5.088443542210164e-05
286.14419565030687
```

Compared to 274.9 according to a group contribution method described in [3].

chemicals.utils.**SG**(*rho*, *rho_ref=999.0170824078306*)

Calculates the specific gravity of a substance with respect to another substance; by default, this is water at 15.555 °C (60 °F). For gases, normally the reference density is 1.2 kg/m^3, that of dry air. However, in general specific gravity should always be specified with respect to the temperature and pressure of its reference fluid. This can vary widely.

$$SG = \frac{\rho}{\rho_{ref}}$$

Parameters

rho [float] Density of the substance, [kg/m^3]

rho_ref [float, optional] Density of the reference substance, [kg/m^3]

Returns

SG [float] Specific gravity of the substance with respect to the reference density, [-]

Notes

Another common reference point is water at 4°C (rho_ref=999.9748691393087). Specific gravity is often used by consumers instead of density. The reference for solids is normally the same as for liquids - water.

Examples

```
>>> SG(860)
0.8608461408159591
```

chemicals.utils.**SG_to_API**(*SG*)

Calculates API of a liquid given its specific gravity, as shown in [1].

$$\text{API gravity} = \frac{141.5}{\text{SG}} - 131.5$$

Parameters

SG [float] Specific gravity of the fluid at 60 degrees Farenheight [-]

Returns

API [float] API of the fluid [-]

Notes

Defined only at 60 degrees Fahrenheit.

References

[1]

Examples

```
>>> SG_to_API(0.7365)
60.62491513917175
```

chemicals.utils.**Vfs_to_zs**(*Vfs*, *Vms*)

Converts a list of mass fractions to mole fractions. Requires molecular weights for all species.

$$z_i = \frac{\frac{\text{Vf}_i}{V_{m,i}}}{\sum_i \frac{\text{Vf}_i}{V_{m,i}}}$$

Parameters

Vfs [iterable] Molar volume fractions [-]

VMs [iterable] Molar volumes of species [m^3/mol]

Returns

zs [list] Mole fractions [-]

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

Molar volumes are specified in terms of pure components only. Function works with any phase.

Examples

Acetone and benzene example

```
>>> Vfs_to_zs([0.596, 0.404], [8.0234e-05, 9.543e-05])
[0.6369779395901142, 0.3630220604098858]
```

chemicals.utils.**Vm_to_rho**(*Vm*, *MW*)

Calculate the density of a chemical, given its molar volume and molecular weight.

$$\rho = \frac{MW}{1000 \cdot VM}$$

Parameters

Vm [float] Molar volume, [m^3/mol]

MW [float] Molecular weight, [g/mol]

Returns

rho [float] Density, [kg/m^3]

References

[1]

Examples

```
>>> Vm_to_rho(0.000132, 86.18)
652.8787878787879
```

chemicals.utils.**Watson_K**(*Tb*, *SG*)

Calculates the Watson or UOP K Characterization factor of a liquid of a liquid given its specific gravity, and its average boiling point as shown in [1].

$$K_W = \frac{T_b^{1/3}}{\text{SG at } 60^\circ\text{F}}$$

Parameters

SG [float] Specific gravity of the fluid at 60 degrees Farenheight [-]

Tb [float] Average normal boiling point, [K]

Returns

K_W [float] Watson characterization factor

Notes

There are different ways to compute the average boiling point, so two different definitions are often used - K_UOP using volume average boiling point (VABP) using distillation points of 10%, 30%, 50%, 70%, and 90%; and K_Watson using mean average boiling point (MeABP).

References

[1]

Examples

```
>>> Watson_K(400, .8)
11.20351186639291
```

Sample problem in Comments on Procedure 2B5.1 of [1]; a fluids has a MEAB of 580 F and a SG of 34.5.

```
>>> from fluids.core import F2K
>>> Watson_K(F2K(580), API_to_SG(34.5))
11.884570347084471
```

chemicals.utils.**Z**(*T*, *P*, *V*)

Calculates the compressibility factor of a gas, given its temperature, pressure, and molar volume.

$$Z = \frac{PV}{RT}$$

Parameters

T [float] Temperature, [K]

P [float] Pressure [Pa]

V [float] Molar volume, [m^3/mol]

Returns

Z [float] Compressibility factor, [-]

References

[1]

Examples

```
>>> Z(600, P=1E6, V=0.00463)
0.9281016730797026
```

chemicals.utils.**d2ns_to_dn2_partials**(*d2ns*, *dns*)

Convert second-order mole number derivatives of a quantity to the following second-order partial derivative:

$$\frac{\partial^2 nF}{\partial n_j \partial n_i} = \frac{\partial^2 F}{\partial n_i \partial n_j} + \frac{\partial F}{\partial n_i} + \frac{\partial F}{\partial n_j}$$

Requires the second order mole number derivatives and the first order mole number derivatives of the mixture only.

Parameters

d2ns [list[float]] Second order derivatives of a quantity with respect to mole number (summing to 1), [prop/mol^2]

dns [list[float]] Derivatives of a quantity with respect to mole number (summing to 1), [prop/mol]

Returns

second_partial_properties [list[list[float]]] Derivatives of a quantity with respect to mole number (summing to 1), [prop]

See also:

dxs_to_dns

dns_to_dn_partials

dxs_to_dn_partials

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

This was originally implemented to allow for the calculation of first mole number derivatices of log fugacity coefficients; the two arguments are the second and first mole number derivatives of the overall mixture log fugacity coefficient.

Derived with the following SymPy code.

```
>>> from sympy import *
>>> n1, n2 = symbols('n1, n2')
>>> f, g, h = symbols('f, g, h', cls=Function)
>>> diff(h(n1, n2)*f(n1,  n2), n1, n2)
f(n1, n2)*Derivative(h(n1, n2), n1, n2) + h(n1, n2)*Derivative(f(n1, n2), n1, n2) +␣
↪Derivative(f(n1, n2), n1)*Derivative(h(n1, n2), n2) + Derivative(f(n1, n2),␣
↪n2)*Derivative(h(n1, n2), n1)
```

Examples

```
>>> d2ns = [[0.152, 0.08, 0.547], [0.08, 0.674, 0.729], [0.547, 0.729, 0.131]]
>>> d2ns_to_dn2_partials(d2ns, [20.0, .124, 900.52])
[[40.152, 20.203999999999997, 921.067], [20.204, 0.922, 901.3729999999999], [921.
↪067, 901.373, 1801.1709999999998]]
```

chemicals.utils.**d2xs_to_d2xsn1**(*d2xs*)

Convert the second mole fraction derivatives of a quantity (calculated so they do not sum to 1) to derivatives such that they do sum to 1 Requires the second derivatives of the mixture only. The size of the returned array is one less than the input in both dimensions

$$\left(\frac{\partial^2 F}{\partial x_i \partial x_j}\right)_{\sum_{x_i}^{N}=1} = \left(\frac{\partial^2 F}{\partial x_i \partial x_j} - \frac{\partial^2 F}{\partial x_i \partial x_N} - \frac{\partial^2 F}{\partial x_j \partial x_N} + \frac{\partial^2 F}{\partial x_N \partial x_N}\right)_{\sum_{x_i}^{N}\neq 1}$$

Parameters

second [list[float]] Second of a quantity with respect to mole fraction (not summing to 1), [prop]

Returns

d2xsm1 [list[float]] Second derivatives of a quantity with respect to mole fraction (summing to 1 by altering the last component's composition), [prop]

Examples

```
>>> d2xs_to_d2xsn1([[-2890.4327598108, -6687.0990540960065, -1549.375443699441], [-
↪6687.099054095983, -2811.2832904869883, -1228.6223853777503], [-1549.
↪3754436994498, -1228.6223853777562, -3667.388098758508]])
[[-3459.069971170426, -7576.489323777324], [-7576.489323777299, -4021.
↪4266184899957]]
```

chemicals.utils.**d2xs_to_dxdn_partials**(*d2xs*, *xs*)

Convert second-order mole fraction derivatives of a quantity (calculated so they do not sum to 1) to the following second-order partial derivative:

$$\frac{\partial^2 nF}{\partial x_j \partial n_i} = \frac{\partial^2 F}{\partial x_i x_j} - \sum_k x_k \frac{\partial^2 F}{\partial x_k \partial x_j}$$

Requires the second derivatives and the mole fractions of the mixture only.

Parameters

d2xs [list[float]]

Second derivatives of a quantity with respect to mole fraction (not summing to 1), [prop]

xs [list[float]] Mole fractions of the species, [-]

Returns

partial_properties [list[float]] Derivatives of a quantity with respect to mole number (summing to 1), [prop]

See also:

dxs_to_dns

dns_to_dn_partials

dxs_to_dn_partials

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

Examples

```
>>> d2xs = [[0.152, 0.08, 0.547], [0.08, 0.674, 0.729], [0.547, 0.729, 0.131]]
>>> d2xs_to_dxdn_partials(d2xs, [0.7, 0.2, 0.1])
[[-0.025100000000000001, -0.18369999999999997, 0.005199999999999982], [-0.0971, 0.
↪41030000000000005, 0.18719999999999992], [0.3699, 0.4653, -0.41080000000000005]]
```

chemicals.utils.**dns_to_dn_partials**(*dns*, *F*, *partial_properties=None*)

Convert the mole number derivatives of a quantity (calculated so they do sum to 1) to partial molar quantites.

$$\left(\frac{\partial nF}{\partial n_i}\right)_{n_{k\neq i}} = F_i + n\left(\frac{\partial F}{\partial n_i}\right)_{n_{k\neq i}}$$

In the formula, the n is 1.

Parameters

dns [list[float]] Derivatives of a quantity with respect to mole number (summing to 1), [prop/mol]

F [float] Property evaluated at constant composition, [prop]

partial_properties [list[float], optional] Optional output array for derivatives of a quantity with respect to mole number (summing to 1), [prop]

Returns

partial_properties [list[float]] Derivatives of a quantity with respect to mole number (summing to 1), [prop]

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

This applies to a specific phase only, not to a mixture of multiple phases.

This is especially useful for fugacity calculations.

Examples

```
>>> dns_to_dn_partials([0.001459, -0.002939, -0.004334], -0.0016567)
[-0.00019770000000000001, -0.0045957, -0.0059907]
```

chemicals.utils.**dxs_to_dn_partials**(*dxs*, *xs*, *F*, *partial_properties=None*)

Convert the mole fraction derivatives of a quantity (calculated so they do not sum to 1) to partial molar quantites. Requires the derivatives and the mole fractions of the mixture.

$$\left(\frac{\partial nF}{\partial n_i}\right) = \left(\frac{\partial F}{\partial x_i}\right) + F - \sum_j x_j \left(\frac{\partial F}{\partial x_j}\right)$$

Parameters

dxs [list[float]] Derivatives of a quantity with respect to mole fraction (not summing to 1), [prop]

xs [list[float]] Mole fractions of the species, [-]

F [float] Property evaluated at constant composition, [prop]

partial_properties [list[float], optional] Array for Derivatives of a quantity with respect to mole number (summing to 1), [prop]

Returns

partial_properties [list[float]] Derivatives of a quantity with respect to mole number (summing to 1), [prop]

See also:

dxs_to_dns

dns_to_dn_partials

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

This applies to a specific phase only, not to a mixture of multiple phases.

Examples

```
>>> dxs_to_dn_partials([-0.0026404, -0.00719, -0.00859], [0.7, 0.2, 0.1],
... -0.0016567)
[-0.00015182, -0.0047014199999999996, -0.00610142]
```

chemicals.utils.**dxs_to_dns**(*dxs*, *xs*, *dns=None*)

Convert the mole fraction derivatives of a quantity (calculated so they do not sum to 1) to mole number derivatives (where the mole fractions do sum to one). Requires the derivatives and the mole fractions of the mixture.

$$\left(\frac{\partial M}{\partial n_i}\right)_{n_{k\neq i}} = \left[\left(\frac{\partial M}{\partial x_i}\right)_{x_{k\neq i}} - \sum_j x_j \left(\frac{\partial M}{\partial x_j}\right)_{x_{k\neq j}}\right]$$

Parameters

dxs [list[float]] Derivatives of a quantity with respect to mole fraction (not summing to 1), [prop]

xs [list[float]] Mole fractions of the species, [-]

dns [list[float], optional] Return array, [prop/mol]

Returns

dns [list[float]] Derivatives of a quantity with respect to mole number (summing to 1), [prop/mol]

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

This applies to a specific phase only, not to a mixture of multiple phases.

Examples

```
>>> dxs_to_dns([-0.0028, -0.00719, -0.00859], [0.7, 0.2, 0.1])
[0.0014570000000000004, -0.002933, -0.004333]
```

chemicals.utils.**dxs_to_dxsn1**(*dxs*)

Convert the mole fraction derivatives of a quantity (calculated so they do not sum to 1) to derivatives such that they do sum to 1 by changing the composition of the last component in the negative of the component which is changed. Requires the derivatives of the mixture only. The size of the returned array is one less than the input.

$$\left(\frac{\partial F}{\partial x_i}\right)_{\sum_{x_i}^{N}=1} = \left(\frac{\partial F}{\partial x_i} - \frac{\partial F}{\partial x_N}\right)_{\sum_{x_i}^{N}\neq 1}$$

Parameters

dxs [list[float]] Derivatives of a quantity with respect to mole fraction (not summing to 1), [prop]

Returns

dxsm1 [list[float]] Derivatives of a quantity with respect to mole fraction (summing to 1 by altering the last component's composition), [prop]

Examples

```
>>> dxs_to_dxsn1([-2651.3181821109024, -2085.574403592012, -2295.0860830203587])
[-356.23209909054367, 209.51167942834672]
```

chemicals.utils.**isentropic_exponent**(*Cp*, *Cv*)

Calculate the isentropic coefficient of an ideal gas, given its constant- pressure and constant-volume heat capacity.

$$k = \frac{C_p}{C_v}$$

Parameters

Cp [float] Ideal gas heat capacity at constant pressure, [J/mol/K]

Cv [float] Ideal gas heat capacity at constant volume, [J/mol/K]

Returns

k [float] Isentropic exponent, [-]

See also:

isentropic_exponent_PV

isentropic_exponent_PT

isentropic_exponent_TV

Notes

For real gases, there are more complexities and formulas. Each of the formulas reverts to this formula in the case of an ideal gas.

References

[1]

Examples

```
>>> isentropic_exponent(33.6, 25.27)
1.329639889196676
```

chemicals.utils.**isentropic_exponent_PT**(*Cp*, *P*, *dV_dT_P*)

Calculate the isentropic coefficient of real fluid using the definition of $P^{(1-k)}T^k = \text{const}$.

$$k = \frac{1}{1 - \frac{P}{C_p}\left(\frac{\partial V}{\partial T}\right)_P}$$

Parameters

Cp [float] Real heat capacity at constant pressure, [J/mol/K]

P [float] Pressure [Pa]

dV_dT_P [float] Derivative of V with respect to T (at constant pressure), [m^3/(mol*K)]

Returns

k_PT [float] Isentropic exponent of a real fluid, [-]

See also:

isentropic_exponent_PV

isentropic_exponent

isentropic_exponent_TV

References

[1], [2]

Examples

Isentropic exponent of air according to Lemmon (2000) at 1000 bar and 300 K:

```
>>> isentropic_exponent_PT(Cp=38.36583283578205, P=100000000.0, dV_dT_P=9.
↪407705210161724e-08)
1.32487270350443
```

chemicals.utils.**isentropic_exponent_PV**(*Cp*, *Cv*, *Vm*, *P*, *dP_dV_T*)

Calculate the isentropic coefficient of real fluid using the definition of $PV^k = \text{const}$.

$$k = -\frac{V}{P}\frac{C_p}{C_v}\left(\frac{\partial P}{\partial V}\right)_T$$

Parameters

Cp [float] Real heat capacity at constant pressure, [J/mol/K]

Cv [float] Real heat capacity at constant volume, [J/mol/K]

Vm [float] Molar volume, [m^3/mol]

P [float] Pressure [Pa]

dP_dV_T [float] Derivative of *P* with respect to *V* (at constant temperature), [Pa*mol/m^3]

Returns

k_PV [float] Isentropic exponent of a real fluid, [-]

See also:

isentropic_exponent

isentropic_exponent_PT

isentropic_exponent_TV

References

[1], [2]

Examples

Isentropic exponent of air according to Lemmon (2000) at 1000 bar and 300 K:

```
>>> isentropic_exponent_PV(Cp=38.36583283578205, Cv=23.98081290153672, Vm=4.
↪730885141495376e-05, P=100000000.0, dP_dV_T=-5417785576072.434)
4.100576762582646
```

chemicals.utils.**isentropic_exponent_TV**(*Cv*, *Vm*, *dP_dT_V*)

Calculate the isentropic coefficient of real fluid using the definition of $TV^{k-1} = \text{const}$.

$$k = 1 + \frac{V}{C_v}\left(\frac{\partial P}{\partial T}\right)_V$$

Parameters

Cv [float] Real heat capacity at constant volume, [J/mol/K]

Vm [float] Molar volume, [m^3/mol]

dP_dT_V [float] Derivative of *P* with respect to *T* (at constant volume), [Pa/K]

Returns

k_TV [float] Isentropic exponent of a real fluid, [-]

See also:

isentropic_exponent_PV

isentropic_exponent_PT

isentropic_exponent

References

[1], [2]

Examples

Isentropic exponent of air according to Lemmon (2000) at 1000 bar and 300 K:

```
>>> isentropic_exponent_TV(Cv=23.98081290153672, Vm=4.730885141495376e-05, dP_dT_
↪V=509689.2959155567)
2.005504495083
```

chemicals.utils.**isobaric_expansion**(*V*, *dV_dT*)

Calculate the isobaric coefficient of a thermal expansion, given its molar volume at a certain T and P, and its derivative of molar volume with respect to T.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Parameters

V [float] Molar volume at T and P, [m^3/mol]

dV_dT [float] Derivative of molar volume with respect to T, [m^3/mol/K]

Returns

beta [float] Isobaric coefficient of a thermal expansion, [1/K]

Notes

For an ideal gas, this expression simplified to:

$$\beta = \frac{1}{T}$$

References

[1]

Examples

Calculated for hexane from the PR EOS at 299 K and 1 MPa (liquid):

```
>>> isobaric_expansion(0.000130229900873546, 1.58875261849113e-7)
0.0012199599384121608
```

chemicals.utils.**isothermal_compressibility**(*V*, *dV_dP*)

Calculate the isothermal coefficient of compressibility, given its molar volume at a certain T and P, and its derivative of molar volume with respect to P.

$$\kappa = -\frac{1}{V}\left(\frac{\partial V}{\partial P}\right)_T$$

Parameters

V [float] Molar volume at T and P, [m^3/mol]

dV_dP [float] Derivative of molar volume with respect to P, [m^3/mol/Pa]

Returns

kappa [float] Isothermal coefficient of compressibility, [1/Pa]

Notes

For an ideal gas, this expression simplified to:

$$\kappa = \frac{1}{P}$$

The isothermal bulk modulus is the inverse of this quantity:

$$K = -V\left(\frac{\partial P}{\partial V}\right)_T$$

The ideal gas isothermal bulk modulus is simply the gas's pressure.

References

[1]

Examples

Calculated for hexane from the PR EOS at 299 K and 1 MPa (liquid):

```
>>> isothermal_compressibility(0.000130229900873546, -2.72902118209903e-13)
2.095541165119158e-09
```

Calculate the bulk modulus of propane from the PR EOS at 294 K as a gas:

```
>>> 1/isothermal_compressibility(0.0024576770482135617, -3.5943321700795866e-09)
683764.5859979445
```

chemicals.utils.**mix_component_flows**(*IDs1*, *IDs2*, *flow1*, *flow2*, *fractions1*, *fractions2*)

Mix two flows of potentially different chemicals of given overall flow rates and flow fractions to determine the outlet components, flow rates, and compositions. The flows do not need to be of the same length.

Parameters

IDs1 [list[str]] List of identifiers of the chemical species in flow one, [-]

IDs2 [list[str]] List of identifiers of the chemical species in flow two, [-]

flow1 [float] Total flow rate of the chemicals in flow one, [mol/s]

flow2 [float] Total flow rate of the chemicals in flow two, [mol/s]

fractions1 [list[float]] Mole fractions of each chemical in flow one, [-]

fractions2 [list[float]] Mole fractions of each chemical in flow two, [-]

Returns

cmps [list[str]] List of identifiers of the chemical species in the combined flow, [-]

moles [list[float]] Flow rates of all chemical species in the combined flow, [mol/s]

Notes

Mass or volume flows and fractions can be used instead of molar ones.

If the two flows have the same components, the output list will be in the same order as the one given; otherwise they are sorted alphabetically.

Examples

```
>>> mix_component_flows(['7732-18-5', '64-17-5'], ['7732-18-5', '67-56-1'], 1, 1,␣
↪[0.5, 0.5], [0.5, 0.5])
(['64-17-5', '67-56-1', '7732-18-5'], [0.5, 0.5, 1.0])
```

chemicals.utils.**mix_component_partial_flows**(*IDs1*, *IDs2*, *ns1=None*, *ns2=None*)

Mix two flows of potentially different chemicals; with the feature that the mole flows of either or both streams may be unknown.

The flows do not need to be of the same length.

Parameters

IDs1 [list[str]] List of identifiers of the chemical species in flow one, [-]

IDs2 [list[str]] List of identifiers of the chemical species in flow two, [-]

ns1 [list[float]] Total flow rate of the chemicals in flow one, [mol/s]

ns2 [list[float]] Total flow rate of the chemicals in flow two, [mol/s]

Returns

cmps [list[str]] List of identifiers of the chemical species in the combined flow, [-]

moles [list[float]] Flow rates of all chemical species in the combined flow, [mol/s]

Notes

Mass or volume flows and fractions can be used instead of molar ones.

If the two flows have the same components, the output list will be in the same order as the one given; otherwise they are sorted alphabetically.

Examples

```
>>> mix_component_partial_flows(['7732-18-5', '64-17-5'], ['7732-18-5', '67-56-1'],␣
↪[0.5, 0.5], [0.5, 0.5])
(['64-17-5', '67-56-1', '7732-18-5'], [0.5, 0.5, 1.0])
>>> mix_component_partial_flows(['7732-18-5', '64-17-5'], ['7732-18-5', '67-56-1'],␣
↪None, [0.5, 0.5])
(['64-17-5', '67-56-1', '7732-18-5'], [0.0, 0.5, 0.5])
>>> mix_component_partial_flows(['7732-18-5', '64-17-5'], ['7732-18-5', '67-56-1'],␣
↪[0.5, 0.5], None)
(['64-17-5', '67-56-1', '7732-18-5'], [0.5, 0.0, 0.5])
>>> mix_component_partial_flows(['7732-18-5', '64-17-5'], ['7732-18-5', '67-56-1'],␣
↪None, None)
(['64-17-5', '67-56-1', '7732-18-5'], [0.0, 0.0, 0.0])
```

chemicals.utils.**mix_multiple_component_flows**(*IDs*, *flows*, *fractions*)

Mix multiple flows of potentially different chemicals of given overall flow rates and flow fractions to determine the outlet components, flow rates, and compositions. The flows do not need to be of the same length.

Parameters

IDs [list[list[str]]] List of lists of identifiers of the chemical species in the flows, [-]

flows [list[float]] List of total flow rates of the chemicals in the streams, [mol/s]

fractions [list[list[float]]] List of lists of mole fractions of each chemical in each flow, [-]

Returns

cmps [list[str]] List of identifiers of the chemical species in the combined flow, [-]

moles [list[float]] Flow rates of all chemical species in the combined flow, [mol/s]

Notes

Mass or volume flows and fractions can be used instead of molar ones.

If the every flow have the same components, the output list will be in the same order as the one given; otherwise they are sorted alphabetically.

Examples

```
>>> mix_multiple_component_flows([['7732-18-5', '64-17-5'], ['7732-18-5', '67-56-1
↪']],
... [1, 1], [[0.5, 0.5], [0.5, 0.5]])
(['64-17-5', '67-56-1', '7732-18-5'], [0.5, 0.5, 1.0])
```

chemicals.utils.**mixing_logarithmic**(*fracs*, *props*)

Simple function calculates a property based on weighted averages of logarithmic properties.

$$y = \sum_i \text{frac}_i \cdot \ln(\text{prop}_i)$$

Parameters

fracs [array-like] Fractions of a mixture

props: array-like Properties

Returns

prop [value] Calculated property

Notes

Does not work on negative values. Returns None if any fractions or properties are missing or are not of the same length.

Examples

```
>>> mixing_logarithmic([0.1, 0.9], [0.01, 0.02])
0.01866065983073615
```

chemicals.utils.**mixing_power**(*fracs*, *props*, *r*)

Power law mixing rule for any property, with a variable exponent *r* as input. Optimiezd routines are available for r=-4,-3,-2,-1,1,2,3,4.

$$\text{prop}_{mix}^r = \sum_i z_i \left(\text{prop}_i\right)^r$$

Parameters

fracs [list[float]] Mole fractions of components (or mass, or volume, etc.), [-]

props [list[float]] Properties of all components, [various]

r [float] Power mixing exponent, [-]

Returns

prop [float] Property for mixture, [*props*]

Notes

This equation is entirely dimensionless; all dimensions cancel.

The following recommendations in [1] exist for different properties:

Surface tension: r = 1 Recommended by an author in [1]; but often non-linear behavior is shown and r= -1 to r=-3 is recommended. r = -1 is most often used.

Liquid thermal conductivity: r = -2 in [1]; this is known also as procedure DIPPR9B.

References

[1]

Examples

```
>>> mixing_power([0.258, 0.742], [0.1692, 0.1528], -2)
0.15657104706719646
```

chemicals.utils.**mixing_simple**(*fracs*, *props*)

Simple function calculates a property based on weighted averages of properties. Weights could be mole fractions, volume fractions, mass fractions, or anything else.

$$y = \sum_i \text{frac}_i \cdot \text{prop}_i$$

Parameters

fracs [array-like] Fractions of a mixture

props: array-like Properties

Returns

prop [value] Calculated property

Notes

Returns None if there is an error, normally if one of the properties is missing or if they are not the same length as the fractions.

Examples

```
>>> mixing_simple([0.1, 0.9], [0.01, 0.02])
0.019000000000000003
```

chemicals.utils.**molar_velocity_to_velocity**(*v_molar*, *MW*)

Calculate the mass-based velocity (m/s) from the molar velocity of the fluid.

$$v = \frac{v_{molar}\sqrt{1000}}{\sqrt{\text{MW}}}$$

Parameters

v_molar [float] Molar velcoity, [m*kg^0.5/s/mol^0.5]

MW [float] Molecular weight, [g/mol]

Returns

v [float] Velocity, [m/s]

Examples

```
>>> molar_velocity_to_velocity(46., 40.445)
228.73
```

chemicals.utils.**none_and_length_check**(*all_inputs*, *length=None*)

Checks inputs for suitability of use by a mixing rule which requires all inputs to be of the same length and non-None. A number of variations were attempted for this function; this was found to be the quickest.

Parameters

all_inputs [array-like of array-like] list of all the lists of inputs, [-]

length [int, optional] Length of the desired inputs, [-]

Returns

False/True [bool] Returns True only if all inputs are the same length (or length *length*) and none of the inputs contain None [-]

Notes

Does not check for nan values.

Examples

```
>>> none_and_length_check(([1, 1], [1, 1], [1, 30], [10,0]), length=2)
True
```

chemicals.utils.**normalize**(*values*)

Simple function which normalizes a series of values to be from 0 to 1, and for their sum to add to 1.

$$x = \frac{x}{sum_i x_i}$$

Parameters

values [array-like] array of values

Returns

fractions [array-like] Array of values from 0 to 1

Notes

Does not work on negative values, or handle the case where the sum is zero.

Examples

```
>>> normalize([3, 2, 1])
[0.5, 0.3333333333333333, 0.16666666666666666]
```

chemicals.utils.**phase_identification_parameter**(*V*, *dP_dT*, *dP_dV*, *d2P_dV2*, *d2P_dVdT*)

Calculate the Phase Identification Parameter developed in [1] for the accurate and efficient determination of whether a fluid is a liquid or a gas based on the results of an equation of state. For supercritical conditions, this provides a good method for choosing which property correlations to use.

$$\Pi = V\left[\frac{\frac{\partial^2 P}{\partial V \partial T}}{\frac{\partial P}{\partial T}} - \frac{\frac{\partial^2 P}{\partial V^2}}{\frac{\partial P}{\partial V}}\right]$$

Parameters

V [float] Molar volume at T and P, [m^3/mol]

dP_dT [float] Derivative of P with respect to T, [Pa/K]

dP_dV [float] Derivative of P with respect to V, [Pa*mol/m^3]

d2P_dV2 [float] Second derivative of P with respect to V, [Pa*mol^2/m^6]

d2P_dVdT [float] Second derivative of P with respect to both V and T, [Pa*mol/m^3/K]

Returns

PIP [float] Phase Identification Parameter, [-]

Notes

Heuristics were used by process simulators before the invent of this parameter.

The criteria for liquid is Pi > 1; for vapor, Pi <= 1.

There is also a solid phase mechanism available. For solids, the Solid Phase Identification Parameter is greater than 1, like liquids; however, unlike liquids, d2P_dVdT is always >0; it is < 0 for liquids and gases.

References

[1], [2]

Examples

Calculated for hexane from the PR EOS at 299 K and 1 MPa (liquid):

```
>>> phase_identification_parameter(0.000130229900874, 582169.397484,
... -3.66431747236e+12, 4.48067893805e+17, -20518995218.2)
11.33428990564796
```

chemicals.utils.**phase_identification_parameter_phase**(*d2P_dVdT*, *V=None*, *dP_dT=None*, *dP_dV=None*, *d2P_dV2=None*)

Uses the Phase Identification Parameter concept developed in [1] and [2] to determine if a chemical is a solid, liquid, or vapor given the appropriate thermodynamic conditions.

The criteria for liquid is PIP > 1; for vapor, PIP <= 1.

For solids, PIP(solid) is defined to be d2P_dVdT. If it is larger than 0, the species is a solid. It is less than 0 for all liquids and gases.

Parameters

d2P_dVdT [float] Second derivative of P with respect to both V and T, [Pa*mol/m^3/K]

V [float, optional] Molar volume at T and P, [m^3/mol]

dP_dT [float, optional] Derivative of P with respect to T, [Pa/K]

dP_dV [float, optional] Derivative of P with respect to V, [Pa*mol/m^3]

d2P_dV2 [float, optionsl] Second derivative of P with respect to V, [Pa*mol^2/m^6]

Returns

phase [str] Either 's', 'l' or 'g'

Notes

The criteria for being a solid phase is checked first, which only requires d2P_dVdT. All other inputs are optional for this reason. However, an exception will be raised if the other inputs become needed to determine if a species is a liquid or a gas.

References

[1], [2]

Examples

Calculated for hexane from the PR EOS at 299 K and 1 MPa (liquid):

```
>>> phase_identification_parameter_phase(-20518995218.2, 0.000130229900874,
... 582169.397484, -3.66431747236e+12, 4.48067893805e+17)
'l'
```

chemicals.utils.**property_mass_to_molar**(*A_mass*, *MW*)

Convert a quantity in mass units [thing/kg] to molar units [thing/mol]. The standard gram-mole is used here, as it is everwhere in this library.

$$A_{\text{molar}} = \frac{A_{\text{mass}}\text{MW}}{1000}$$

Parameters

A_mass [float] Quantity in molar units [thing/kg]

MW [float] Molecular weight, [g/mol]

Returns

A_molar [float] Quantity in molar units [thing/mol]

Notes

For legacy reasons, if the value *A_mass* is None, None is also returned and no exception is returned.

Examples

```
>>> property_mass_to_molar(20.0, 18.015)
0.3603
```

chemicals.utils.**property_molar_to_mass**(*A_molar*, *MW*)

Convert a quantity in molar units [thing/mol] to mass units [thing/kg]. The standard gram-mole is used here, as it is everwhere in this library.

$$A_{\text{mass}} = \frac{1000 A_{\text{molar}}}{\text{MW}}$$

Parameters

A_molar [float] Quantity in molar units [thing/mol]

MW [float] Molecular weight, [g/mol]

Returns

A_mass [float] Quantity in molar units [thing/kg]

Notes

For legacy reasons, if the value *A_molar* is None, None is also returned and no exception is returned.

Examples

```
>>> property_molar_to_mass(500, 18.015)
27754.648903691366
```

chemicals.utils.**radius_of_gyration**(*MW*, *A*, *B*, *C*, *planar=False*)

Calculates the radius of gyration of a molecule using the DIPPR definition. The parameters *A*, *B*, and *C* must be obtained from either vibrational scpectra and analysis or quantum chemistry calculations of programs such as *psi <https://psicode.org/>*.

For planar molecules defined by only two moments of inertia,

$$R_g = \sqrt{\sqrt{AB}\frac{N_A}{\text{MW}}}$$

For non-planar molecules with three moments of inertia,

$$R_g = \sqrt{\frac{2\pi(ABC)^{1/3}N_A}{\text{MW}}}$$

Parameters

MW [float] Molecular weight, [g/mol]

A [float] First principle moment of inertia, [kg*m^2]

B [float] Second principle moment of inertia, [kg*m^2]

C [float] Third principle moment of inertia, [kg*m^2]

planar [bool] Whether the molecule is flat or not, [-]

Returns

Rg [float] Radius of gyration, [m]

Notes

There are many, many quantum chemistry models available which give different results.

References

[1], [2]

Examples

Example calcultion in [1] for hydrazine (optimized with HF/6-31G model):

```
>>> radius_of_gyration(MW=32.00452, planar=False, A=5.692E-47, B=3.367E-46, C=3.
↪681E-46)
1.50581642e-10
```

The same calculation was performed with *psi* and somewhat different parameters obtained

```
>>> radius_of_gyration(MW=32.00452, planar=False, A=6.345205205562681e-47, B=3.
↪2663291891213418e-46, C=3.4321304373822523e-46)
1.507895671e-10
```

A planar molecule, bromosilane, has two principle moments of inertia in [2]. They are 2.80700 cm^-1 and 0.14416 cm^-1. These can be converted to MHz as follows:

These can then be converted to units of AMU*Angstrom^2, and from there to kg*m^2.

```
>>> A, B = 2.80700, 0.14416
>>> from scipy.constants import atomic_mass, c, angstrom
>>> A, B = A*c*1e-4, B*c*1e-4 # from cm^-1 to MHz
>>> A, B = [505379.15/i for i in (A, B)] #  TODO which constants did this␣
↪conversion factor come from, AMU*Angstrom^2
>>> A, B = [i*atomic_mass*angstrom**2 for i in (A, B)] # amu*angstrom^2 to kg*m^2
>>> radius_of_gyration(A=A, B=B, planar=True, MW=111.01, C=0)
4.8859099776e-11
```

Alternatively, doing the conversion all in one:

```
>>> A, B = 2.80700, 0.14416
>>> from scipy.constants import c, h, pi
>>> A, B = A*c*100, B*c*100 # from cm^-1 to Hz
>>> A, B = [h/(8*pi**2)/i for i in (A, B)] # from Hz to kg*m^2
>>> radius_of_gyration(A=A, B=B, planar=True, MW=111.01, C=0)
4.885909296e-11
```

This is also nicely documented on this page: https://cccbdb.nist.gov/convertmomint.asp which was unfortunately found by the author after figuring it out the hard way.

chemicals.utils.**remove_zeros**(*values*, *tol=1e-06*)

Simple function which removes zero values from an array, and replaces them with a user-specified value, normally a very small number. Helpful for the case where a function can work with values very close to zero but not quite zero. The resulting array is normalized so the sum is still one.

Parameters

values [array-like] array of values

tol [float] The replacement value for zeroes

Returns

fractions [array-like] Array of values from 0 to 1

Notes

Works on numpy arrays, and returns numpy arrays only for that case.

Examples

```
>>> remove_zeros([0, 1e-9, 1], 1e-12)
[9.99999998999e-13, 9.99999998999e-10, 0.999999998999]
```

chemicals.utils.**rho_to_API**(*rho*, *rho_ref=999.0170824078306*)

Calculates API of a liquid given its mass density, as shown in [1].

$$\text{API gravity} = \frac{141.5\rho_{ref}}{\rho} - 131.5$$

Parameters

rho [float] Mass density the fluid at 60 degrees Farenheight [kg/m^3]

rho_ref [float, optional] Density of the reference substance, [kg/m^3]

Returns

API [float] API of the fluid [-]

Notes

Defined only at 60 degrees Fahrenheit.

References

[1]

Examples

```
>>> rho_to_API(820)
40.8913623
>>> SG_to_API(SG(820))
40.8913623
```

chemicals.utils.**rho_to_Vm**(*rho*, *MW*)

Calculate the molar volume of a chemical, given its density and molecular weight.

$$V_m = \left(\frac{1000\rho}{MW}\right)^{-1}$$

Parameters

rho [float] Density, [kg/m^3]

MW [float] Molecular weight, [g/mol]

Returns

Vm [float] Molar volume, [m^3/mol]

References

[1]

Examples

```
>>> rho_to_Vm(652.9, 86.18)
0.0001319957114412621
```

chemicals.utils.**solve_flow_composition_mix**(*Fs*, *zs*, *ws*, *MWs*)

Solve a stream composition problem where some specs are mole flow rates; some are mass fractions; and some are mole fractions. This algorithm requires at least one mole flow rate; and for every other component, a single spec in mole or mass or a flow rate. It is permissible for no components to have mole fractions; or no components to have weight fractions; or both.

Parameters

Fs [list[float]] List of mole flow rates; None if not specified for a component, [mol/s]

zs [list[float]] Mole fractions; None if not specified for a component [-]

ws [list[float]] Mass fractions; None if not specified for a component [-]

MWs [list[float]] Molecular weights, [g/mol]

Returns

Fs [list[float]] List of mole flow rates, [mol/s]

zs [list[float]] Mole fractions, [-]

ws [list[float]] Mass fractions, [-]

Notes

A fast path is used if no weight fractions are provided; the calculation is much simpler for that case.

This algorithm was derived using SymPy, and framed in a form which allows for explicit solving. This is capable of solving large-scale problems i.e. with 1000 components a solve time is 1 ms; with 10000 it is 10 ms.

Examples

```
>>> Fs = [3600, None, None, None, None]
>>> zs = [None, .1, .2, None, None]
>>> ws = [None, None, None, .01, .02]
>>> MWs = [18.01528, 46.06844, 32.04186, 72.151, 142.286]
>>> Fs, zs, ws = solve_flow_composition_mix(Fs, zs, ws, MWs)
>>> Fs
[3600, 519.3039148597746, 1038.6078297195493, 17.44015034881175, 17.687253669610733]
>>> zs
[0.6932356751002141, 0.1, 0.2, 0.0033583706669188186, 0.003405954232867038]
>>> ws
[0.5154077420893426, 0.19012206531421305, 0.26447019259644433, 0.01, 0.02]
```

chemicals.utils.**speed_of_sound**(*V*, *dP_dV*, *Cp*, *Cv*, *MW=None*)

Calculate a real fluid's speed of sound. The required derivatives should be calculated with an equation of state, and *Cp* and *Cv* are both the real fluid versions. Expression is given in [1] and [2]; a unit conversion is further performed to obtain a result in m/s. If MW is not provided the result is returned in units of m*kg^0.5/s/mol^0.5.

$$w = \left[-V^2 \left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

Parameters

V [float] Molar volume of fluid, [m^3/mol]

dP_dV [float] Derivative of *P* with respect to *V*, [Pa*mol/m^3]

Cp [float] Real fluid heat capacity at constant pressure, [J/mol/K]

Cv [float] Real fluid heat capacity at constant volume, [J/mol/K]

MW [float, optional] Molecular weight, [g/mol]

Returns

w [float] Speed of sound for a real gas, m/s or m*kg^0.5/s/mol^0.5 if MW missing

Notes

An alternate expression based on molar density is as follows:

$$w = \left[\left(\frac{\partial P}{\partial \rho}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

The form with the unit conversion performed inside it is as follows:

$$w = \left[-V^2 \frac{1000}{MW} \left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

References

[1], [2]

Examples

Example from [2]:

```
>>> speed_of_sound(V=0.00229754, dP_dV=-3.5459e+08, Cp=153.235, Cv=132.435, MW=67.
↪152)
179.5868138460819
```

chemicals.utils.**to_num**(*values*)

Legacy function to turn a list of strings into either floats (if numeric), stripped strings (if not) or None if the string is empty. Accepts any numeric formatting the float function does.

Parameters

values [list] list of strings

Returns

values [list] list of floats, strings, and None values [-]

Examples

```
>>> to_num(['1', '1.1', '1E5', '0xB4', ''])
[1.0, 1.1, 100000.0, '0xB4', None]
```

chemicals.utils.**v_molar_to_v**(*v_molar*, *MW*)

Convert a velocity from units of the molar velocity form to standard m/s units.

$$v(\text{m/s}) = v\left(\frac{\text{m}\sqrt{\text{kg}}}{s\sqrt{\text{mol}}}\right)\text{MW (g/mol)}^{-0.5}\cdot\left(\frac{1000\text{g}}{1\text{kg}}\right)^{0.5}$$

Parameters

v_molar [float] Molar velocity, [m*kg^0.5/s/mol^0.5]

MW [float] Molecular weight, [g/mol]

Returns

v [float] Velocity, [m/s]

Examples

```
>>> v_molar_to_v(67.10998435404377, 18.015)
499.99999999999994
```

chemicals.utils.**v_to_v_molar**(*v*, *MW*)

Convert a velocity from units of m/s to a "molar" form of velocity, compatible with thermodynamic calculations on a molar basis.

$$v\left(\frac{\text{m}\sqrt{\text{kg}}}{s\sqrt{\text{mol}}}\right) = v(\text{m/s})\sqrt{\text{MW (g/mol)}}\cdot\left(\frac{1000\text{g}}{1\text{kg}}\right)^{-0.5}$$

Parameters

v [float] Velocity, [m/s]

MW [float] Molecular weight, [g/mol]

Returns

v_molar [float] Molar velocity, [m*kg^0.5/s/mol^0.5]

Examples

```
>>> v_to_v_molar(500, 18.015)
67.10998435404377
```

chemicals.utils.**vapor_mass_quality**(*VF*, *MWl*, *MWg*)

Calculates the vapor quality on a mass basis of a two-phase mixture; this is the most common definition, where 1 means a pure vapor and 0 means a pure liquid. The vapor quality on a mass basis is related to the mole basis vapor fraction according to the following relationship:

$$x = \frac{\frac{V}{F} \cdot \text{MW}_g}{(1 - \frac{V}{F})\text{MW}_l + \frac{V}{F}\text{MW}_g}$$

Parameters

VF [float] Mole-basis vapor fraction (0 = pure vapor, 1 = pure liquid), [-]

MWl [float] Average molecular weight of the liquid phase, [g/mol]

MWg [float] Average molecular weight of the vapor phase, [g/mol]

Returns

quality [float] Vapor mass fraction of the two-phase system, [-]

Notes

Other definitions of vapor fraction use an enthalpy basis instead of a mass basis; still other less common ones take 1 to be the value of the liquid, and 0 as pure vapor.

References

[1]

Examples

```
>>> vapor_mass_quality(0.5, 60, 30)
0.3333333333333333
```

chemicals.utils.**velocity_to_molar_velocity**(*v*, *MW*)

Calculate the molar velocity from the mass-based (m/s) velocity of the fluid.

$$v_{molar} = \frac{v\sqrt{\text{MW}}}{\sqrt{1000}}$$

Parameters

v [float] Velocity, [m/s]

MW [float] Molecular weight, [g/mol]

Returns

v_molar [float] Molar velcoity, [m*kg^0.5/s/mol^0.5]

Examples

```
>>> velocity_to_molar_velocity(228.73, 40.445)
46.
```

chemicals.utils.**ws_to_zs**(*ws*, *MWs*)

Converts a list of mass fractions to mole fractions. Requires molecular weights for all species.

$$z_i = \frac{\frac{w_i}{MW_i}}{\sum_i \frac{w_i}{MW_i}}$$

Parameters

ws [iterable] Mass fractions [-]

MWs [iterable] Molecular weights [g/mol]

Returns

zs [iterable] Mole fractions [-]

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

Examples

```
>>> ws_to_zs([0.3333333333333333, 0.6666666666666666], [10, 20])
[0.5, 0.5]
```

chemicals.utils.**zs_to_Vfs**(*zs*, *Vms*)

Converts a list of mole fractions to volume fractions. Requires molar volumes for all species.

$$\text{Vf}_i = \frac{z_i V_{m,i}}{\sum_i z_i V_{m,i}}$$

Parameters

zs [iterable] Mole fractions [-]

VMs [iterable] Molar volumes of species [m^3/mol]

Returns

Vfs [list] Molar volume fractions [-]

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

Molar volumes are specified in terms of pure components only. Function works with any phase.

Examples

Acetone and benzene example

```
>>> zs_to_Vfs([0.637, 0.363], [8.0234e-05, 9.543e-05])
[0.5960229712956298, 0.4039770287043703]
```

chemicals.utils.**zs_to_ws**(*zs*, *MWs*)

Converts a list of mole fractions to mass fractions. Requires molecular weights for all species.

$$w_i = \frac{z_i MW_i}{MW_{avg}}$$

$$MW_{avg} = \sum_i z_i MW_i$$

Parameters

zs [iterable] Mole fractions [-]

MWs [iterable] Molecular weights [g/mol]

Returns

ws [iterable] Mass fractions [-]

Notes

Does not check that the sums add to one. Does not check that inputs are of the same length.

Examples

```
>>> zs_to_ws([0.5, 0.5], [10, 20])
[0.3333333333333333, 0.6666666666666666]
```

1.31 Support for pint Quantities (chemicals.units)

Basic module which wraps all chemicals functions and classes to be compatible with the pint unit handling library. All other object - dicts, lists, etc - are not wrapped.

```
>>> import chemicals
>>> chemicals.units.Antoine
<function Antoine at 0x...>
```

The chemicals.units module also supports star imports; the same objects exported when importing from the main library will be imported from chemicals.units.

```
>>> from chemicals.units import *
```

```
>>> CAS = CAS_from_any('methanol')
>>> Tc(CAS), Pc(CAS), Vc(CAS), Zc(CAS)
(<Quantity(513.38, 'kelvin')>, <Quantity(8215850.0, 'pascal')>, <Quantity(0.00011382819,
↪'meter ** 3 / mole')>, <Quantity(0.219093353, 'dimensionless')>)
>>> (Tt(CAS), Tm(CAS), Tb(CAS), Pt(CAS))
(<Quantity(175.61, 'kelvin')>, <Quantity(175.15, 'kelvin')>, <Quantity(337.632383,
↪'kelvin')>, <Quantity(0.186349762, 'pascal')>)
```

```
>>> iapws95_rho(T=55*u.degF, P=500*u.psi)
<Quantity(1000.97992, 'kilogram / meter ** 3')>
>>> sigma_IAPWS(200*u.degR)
<Quantity(0.0897667127, 'newton / meter')>
```

```
>>> molecular_weight({'H': 12, 'C': 20, 'O': 5})
<Quantity(332.30628, 'gram / mole')>
```

Functions that do not return numbers are not converted into pint quantities, for example:

```
>>> atoms_to_Hill({'H': 5, 'C': 2, 'Br': 1})
'C2H5Br'
```

Functions that return dimensionless numbers are pint quantities.

```
>>> logP('67-56-1')
<Quantity(-0.74, 'dimensionless')>
>>> Stiel_polar_factor(Psat=169745*u.Pa, Pc=22048321.0*u.Pa, omega=0.344)
<Quantity(0.0232214674, 'dimensionless')>
```

It is also possible to use *chemicals.units* without the star import:

```
>>> import chemicals.units
```

When a function is used with inputs that should have units but they aren't provided by the user, an error is raised.

```
>>> ideal_gas(298.15, 101325.)
Traceback (most recent call last):
TypeError: 298.15 has no quantity
```

For further information on this interface, please see the documentation of fluids.units which is built in the same way.

1.32 Vapor Pressure (chemicals.vapor_pressure)

This module contains various vapor pressure estimation routines, dataframes of fit coefficients, some compound-specific equations, some analytical fitting routines, and sublimation pressure routines.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.32.1 Fit Correlations

chemicals.vapor_pressure.**Antoine**(*T*, *A*, *B*, *C*, *base=10.0*)

Calculates vapor pressure of a chemical using the Antoine equation. Parameters *A*, *B*, and *C* are chemical-dependent. Parameters can be found in numerous sources; however units of the coefficients used vary. Originally proposed by Antoine (1888) [2].

$$\log_{\text{base}} P^{\text{sat}} = A - \frac{B}{T + C}$$

Parameters

T [float] Temperature of fluid, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

base [float, optional] Optional base of logarithm; 10 by default

Returns

Psat [float] Vapor pressure calculated with coefficients [Pa]

Notes

Assumes coefficients are for calculating vapor pressure in Pascal. Coefficients should be consistent with input temperatures in Kelvin; however, if both the given temperature and units are specific to degrees Celcius, the result will still be correct.

Converting units in input coefficients:

- **ln to log10**: Divide A and B by ln(10)=2.302585 to change parameters for a ln equation to a log10 equation.
- **log10 to ln**: Multiply A and B by ln(10)=2.302585 to change parameters for a log equation to a ln equation.
- **mmHg to Pa**: Add log10(101325/760)= 2.1249 to A.
- **kPa to Pa**: Add log_{base}(1000)= 6.908 to A for log(base)
- **bar to Pa**: Add log_{base}(100000)= 11.5129254 to A for log(base)
- **°C to K**: Subtract 273.15 from C only!

Note that if *C* is negative and *T* is less than *C*, the predicted vapor pressure would be high and positive at those temperatures under *C*; and a singularity would occur at *T* == *C*. This implementation is corrected to return zero for the case of *T* + *C* < *0.0*, which matches the intention of the Antoine equation.

References

[1], [2], [3]

Examples

Methane, coefficients from [1], at 100 K:

```
>>> Antoine(100.0, 8.7687, 395.744, -6.469)
34478.367349639906
```

Tetrafluoromethane, coefficients from [1], at 180 K

```
>>> Antoine(180, A=8.95894, B=510.595, C=-15.95)
702271.0518579542
```

Oxygen at 94.91 K, with coefficients from [3] in units of °C, mmHg, log10, showing the conversion of coefficients A (mmHg to Pa) and C (°C to K)

```
>>> Antoine(94.91, 6.83706+2.1249, 339.2095, 268.70-273.15)
162978.88655572367
```

n-hexane with Antoine coefficients from the NIST webbook in units of K and bar, calculating the vapor pressure in Pa at 200 K:

```
>>> Antoine(T=200, A=3.45604+5, B=1044.038, C=-53.893)
20.4329803671
```

chemicals.vapor_pressure.**Wagner**(*T*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates vapor pressure using the Wagner equation (2.5, 5 form).

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

$$\ln P^{sat} = \ln P_c + \frac{a\tau + b\tau^{1.5} + c\tau^{2.5} + d\tau^5}{T_r}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

Psat [float] Vapor pressure at T [Pa]

Notes

Warning: Pc is often treated as adjustable constant. This is also called the PPDS16 equation [3].

References

[1], [2], [3]

Examples

Methane, coefficients from [2], at 100 K.

```
>>> Wagner(100., 190.551, 4599200, -6.02242, 1.26652, -0.5707, -1.366)
34415.004762637
```

chemicals.vapor_pressure.**Wagner_original**(*T*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates vapor pressure using the Wagner equation (3, 6 form).

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

$$\ln P^{sat} = \ln P_c + \frac{a\tau + b\tau^{1.5} + c\tau^3 + d\tau^6}{T_r}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

Psat [float] Vapor pressure at T [Pa]

Notes

Warning: Pc is often treated as adjustable constant. This is also called the PPDS1 equation [3].

References

[1], [2], [3]

Examples

Methane, coefficients from [2], at 100 K.

```
>>> Wagner_original(100.0, 190.53, 4596420., a=-6.00435, b=1.1885,
... c=-0.834082, d=-1.22833)
34520.44601450499
```

chemicals.vapor_pressure.**TRC_Antoine_extended**(*T*, *Tc*, *to*, *A*, *B*, *C*, *n*, *E*, *F*)

Calculates vapor pressure of a chemical using the TRC Extended Antoine equation. Parameters are chemical dependent, and said to be from the Thermodynamics Research Center (TRC) at Texas A&M. Coefficients for various chemicals can be found in [1].

$$\log_{10} P^{sat} = A - \frac{B}{T + C} + 0.43429x^n + Ex^8 + Fx^{12}$$

$$x = \max\left(\frac{T - t_o - 273.15}{T_c}, 0\right)$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature of fluid, [K]

to [float] Fit temperature-transition parameter, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

n [float] Fit parameter, [-]

E [float] Fit parameter, [-]

F [float] Fit parameter, [-]

Returns

Psat [float] Vapor pressure calculated with coefficients [Pa]

Notes

Assumes coefficients are for calculating vapor pressure in Pascal. Coefficients should be consistent with input temperatures in Kelvin;

References

[1]

Examples

Tetrafluoromethane, coefficients from [1], at 180 K:

```
>>> TRC_Antoine_extended(T=180.0, Tc=227.51, to=-120., A=8.95894,
... B=510.595, C=-15.95, n=2.41377, E=-93.74, F=7425.9)
706317.0898414153
```

chemicals.vapor_pressure.**Yaws_Psat**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculates vapor pressure of a chemical using the Yaws equation for vapor pressure. Parameters *A*, *B*, *C*, *D*, and *E* are chemical-dependent. Parameters can be found in numerous sources; however units of the coefficients used vary.

$$\log_{10} P^{\text{sat}} = A + \frac{B}{T} + C\log_{10}(T) + DT + ET^2$$

Parameters

T [float] Temperature of fluid, [K]

A [float] *A* parameter, [-]

B [float] *B* parameter, [K]

C [float] *C* parameter, [-]

D [float] *D* parameter, [1/K]

E [float] *E* parameter, [1/K^2]

Returns

Psat [float] Vapor pressure calculated with coefficients [Pa]

Notes

Assumes coefficients are for calculating vapor pressure in Pascal. Coefficients should be consistent with input temperatures in Kelvin;

Converting units in input coefficients:

- **mmHg to Pa**: Add log10(101325/760)= 2.1249 to A.
- **kPa to Pa**: Add log_{10}(1000)= 3 to A
- **bar to Pa**: Add log_{10}(100000)= 5 to A

References

[1], [2]

Examples

Acetone, coefficients from [1], at 400 K and with the conversion of *A* to obtain a result in Pa:

```
>>> Yaws_Psat(T=400.0, A=28.588 + log10(101325/760), B=-2469, C=-7.351, D=2.8025E-
↪10, E=2.7361E-6)
708657.089106
```

Coefficients for benzene from [2] at 400 K; that source outputs vapor pressure in kPa. That style of coefficients can be converted to *Pa* by adding *3* to *A*.

```
>>> Yaws_Psat(T=400.0, A=39.7918+3, B=-2965.83, C=-12.073, D=0.0033269, E=1.58609e-
↪6)
352443.191026
```

chemicals.vapor_pressure.**TDE_PVExpansion**(*T*, *a1*, *a2*, *a3*, *a4=0.0*, *a5=0.0*, *a6=0.0*, *a7=0.0*, *a8=0.0*)

Calculates vapor pressure or sublimation pressure of a chemical using the PVExpansion equation for vapor pressure or sublimation pressure. Parameters *a1*, *a2*, *a3*, *a4*, *a5*, *a6*, *a7*, and *a8* are chemical-dependent. Parameters can be found in various sources; however units of the coefficients used vary.

$$\log P^{\text{sat}} = a_1 + \frac{a_2}{T} + a_3 \ln(T) + a_4 T + a_5 T^2 + \frac{a_6}{T^2} + a_7 T^6 + \frac{a_8}{T^4}$$

Parameters

T [float] Temperature of fluid, [K]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

a5 [float] Regression parameter, [-]

a6 [float] Regression parameter, [-]

a7 [float] Regression parameter, [-]

a8 [float] Regression parameter, [-]

Returns

Psat [float] Vapor pressure calculated with coefficients [Pa]

Notes

Coefficients in [1] produce a vapor pressure in kPa; add log(1000) to *a1* to make them produce vapor pressure in Pa.

References

[1]

Examples

Coefficients for sublimation pressure from [1]:

```
>>> TDE_PVExpansion(T=273.16, a1=23.7969+log(1000), a2=-11422, a3=0.177978)
4.06220657398e-05
```

1.32.2 Fit Correlation Derivatives

chemicals.vapor_pressure.**dAntoine_dT**(*T*, *A*, *B*, *C*, *base=10.0*)

Calculates the first temperature derivative of vapor pressure of a chemical using the Antoine equation. Parameters *A*, *B*, and *C* are chemical-dependent.

$$\frac{\partial P^{\text{sat}}}{\partial T} = \frac{B \text{base}^{A-\frac{B}{C+T}} \log(\text{base})}{(C+T)^2}$$

Parameters

T [float] Temperature of fluid, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

base [float, optional] Optional base of logarithm; 10 by default

Returns

dPsat_dT [float] First temperature derivative of vapor pressure calculated with coefficients [Pa/K]

Examples

Methane at 100 K:

```
>>> dAntoine_dT(100.0, 8.7687, 395.744, -6.469)
3591.4147747481
```

chemicals.vapor_pressure.**d2Antoine_dT2**(*T*, *A*, *B*, *C*, *base=10.0*)

Calculates the second temperature derivative of vapor pressure of a chemical using the Antoine equation. Parameters *A*, *B*, and *C* are chemical-dependent.

$$\frac{\partial^2 P^{\text{sat}}}{\partial T^2} = \frac{B \text{base}^{A-\frac{B}{C+T}} \left(\frac{B \log(\text{base})}{C+T} - 2\right) \log(\text{base})}{(C+T)^3}$$

Parameters

T [float] Temperature of fluid, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

base [float, optional] Optional base of logarithm; 10 by default

Returns

d2Psat_dT2 [float] Second temperature derivative of vapor pressure calculated with coefficients [Pa/K^2]

Examples

Methane at 100 K:

```
>>> d2Antoine_dT2(100.0, 8.7687, 395.744, -6.469)
297.30093799054
```

chemicals.vapor_pressure.**dWagner_dT**(*T*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates the first temperature derivative of vapor pressure using the Wagner equation (2.5, 5 form).

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

$$\frac{\partial P^{\text{sat}}}{\partial T} = P_c \left(\frac{T_c \left(-\frac{a}{T_c} - \frac{1.5 b \tau^{0.5}}{T_c} - \frac{2.5 c \tau^{1.5}}{T_c} - \frac{5 d \tau^4}{T_c} \right)}{T} - \frac{T_c \left(a\tau + b\tau^{1.5} + c\tau^{2.5} + d\tau^5 \right)}{T^2} \right) e^{\frac{T_c \left(a\tau + b\tau^{1.5} + c\tau^{2.5} + d\tau^5 \right)}{T}}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

dPsat_dT [float] First temperature derivative of vapor pressure at T [Pa/K]

Examples

Methane at 100 K.

```
>>> dWagner_dT(100., 190.551, 4599200, -6.02242, 1.26652, -0.5707, -1.366)
3587.2910498076
```

chemicals.vapor_pressure.**d2Wagner_dT2**(*T*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates the second temperature derivative of vapor pressure using the Wagner equation (2.5, 5 form).

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

$$\frac{\partial^2 P^{\text{sat}}}{\partial T^2} = \frac{P_c \left(\frac{\frac{0.75b}{\tau^{0.5}} + 3.75 c \tau^{0.5} + 20 d \tau^3}{T_c} + \frac{2\left(a + 1.5 b \tau^{0.5} + 2.5 c \tau^{1.5} + 5 d \tau^4\right)}{T} + \frac{25\left(\frac{a}{5} + 0.3 b \tau^{0.5} + 0.5 c \tau^{1.5} + d\tau^4 - \frac{T_c\left(-a\tau - b\tau^{1.5} - c\tau^{2.5} - d\tau^5\right)}{5T}\right)^2}{T} - \right.}{T}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

d2Psat_dT2 [float] Second temperature derivative of vapor pressure at T [Pa/K^2]

Notes

This second derivative is infinity at T == Tc.

Examples

Methane at 100 K.

```
>>> d2Wagner_dT2(100., 190.551, 4599200, -6.02242, 1.26652, -0.5707, -1.366)
296.7091513877
```

chemicals.vapor_pressure.**dWagner_original_dT**(*T*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates first temperature derivative of vapor pressure using the Wagner equation (3, 6 form).

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

$$\frac{\partial P^{\text{sat}}}{\partial T} = P_c \left(\frac{T_c \left(-\frac{a}{T_c} - \frac{1.5 b \tau^{0.5}}{T_c} - \frac{3 c \tau^2}{T_c} - \frac{6 d \tau^5}{T_c} \right)}{T} - \frac{T_c \left(a\tau + b\tau^{1.5} + c\tau^3 + d\tau^6 \right)}{T^2} \right) e^{\frac{T_c \left(a\tau + b\tau^{1.5} + c\tau^3 + d\tau^6 \right)}{T}}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

dPsat_dT [float] First temperature derivative of vapor pressure at T [Pa/K]

Examples

Methane at 100 K.

```
>>> dWagner_original_dT(100.0, 190.53, 4596420., a=-6.00435, b=1.1885,
... c=-0.834082, d=-1.22833)
3593.70783283
```

chemicals.vapor_pressure.**d2Wagner_original_dT2**(*T*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates second temperature derivative of vapor pressure using the Wagner equation (3, 6 form).

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

$$\frac{\partial^2 P^{\text{sat}}}{\partial T^2} = \frac{P_c \left(\frac{\frac{0.75b}{\tau^{0.5}} + 6c\tau + 30d\tau^4}{T_c} + \frac{2\left(a + 1.5b\tau^{0.5} + 3c\tau^2 + 6d\tau^5\right)}{T} + \frac{36\left(\frac{a}{6} + 0.25b\tau^{0.5} + \frac{c\tau^2}{2} + d\tau^5 - \frac{T_c\left(-a\tau - b\tau^{1.5} - c\tau^3 - d\tau^6\right)}{6T}\right)^2}{T} - \frac{2T_c\left(-a\tau - b\tau^1}{T}\right.}{T}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

d2Psat_dT2 [float] Second temperature derivative of vapor pressure at T [Pa/K^2]

Notes

This second derivative is infinity at T == Tc.

Examples

Methane at 100 K.

```
>>> d2Wagner_original_dT2(100.0, 190.53, 4596420., a=-6.00435, b=1.1885,
... c=-0.834082, d=-1.22833)
296.87593368224
```

chemicals.vapor_pressure.**dTRC_Antoine_extended_dT**(*T*, *Tc*, *to*, *A*, *B*, *C*, *n*, *E*, *F*)

Calculates the first temperature derivative of vapor pressure of a chemical using the TRC Extended Antoine equation.

$$\frac{\partial P^{\text{sat}}}{\partial T} = 10^{A - \frac{B}{C+T} + \frac{E\left(T - T_{ref} - to\right)^8}{T_c^8} + \frac{F\left(T - T_{ref} - to\right)^{12}}{T_c^{12}} + f\left(\frac{T - T_{ref} - to}{T_c}\right)^n} \left(\frac{B}{(C+T)^2} + \frac{8E\left(T - T_{ref} - to\right)^7}{T_c^8} + \frac{12F\left(T - T_{ref} - t\right.}{T_c^{12}}\right.$$

$$x = \max\left(\frac{T - t_o - 273.15}{T_c}, 0\right)$$

$$T_{ref} = 273.15 \text{ K}$$

$$f = 0.43429$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature of fluid, [K]

to [float] Fit temperature-transition parameter, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

n [float] Fit parameter, [-]

E [float] Fit parameter, [-]

F [float] Fit parameter, [-]

Returns

dPsat_dT [float] First temperature derivative of vapor pressure calculated with coefficients [Pa/K]

Examples

Tetrafluoromethane at 180 K:

```
>>> dTRC_Antoine_extended_dT(T=180.0, Tc=227.51, to=-120., A=8.95894,
... B=510.595, C=-15.95, n=2.41377, E=-93.74, F=7425.9)
31219.6061263
```

chemicals.vapor_pressure.**d2TRC_Antoine_extended_dT2**(*T*, *Tc*, *to*, *A*, *B*, *C*, *n*, *E*, *F*)

Calculates the second temperature derivative of vapor pressure of a chemical using the TRC Extended Antoine equation.

$$\frac{\partial^2 P^{\text{sat}}}{\partial T^2} = 10^{A-\frac{B}{C+T}+\frac{E\left(-T+T_{ref}+to\right)^8}{T_c^8}+\frac{F\left(-T+T_{ref}+to\right)^{12}}{T_c^{12}}+f\left(-\frac{-T+T_{ref}+to}{T_c}\right)^n}\left(-\frac{2B}{(C+T)^3}+\frac{56E\left(-T+T_{ref}+to\right)^6}{T_c^8}+\frac{132F\left(-\right.\right.}{}$$

$$x = \max\left(\frac{T - t_o - 273.15}{T_c}, 0\right)$$

$$T_{ref} = 273.15 \text{ K}$$

$$f = 0.43429$$

Parameters

T [float] Temperature of fluid, [K]

Tc [float] Critical temperature of fluid, [K]

to [float] Fit temperature-transition parameter, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

n [float] Fit parameter, [-]

E [float] Fit parameter, [-]

F [float] Fit parameter, [-]

Returns

d2Psat_dT2 [float] Second temperature derivative of vapor pressure calculated with coefficients [Pa/K]

Examples

Tetrafluoromethane at 180 K:

```
>>> d2TRC_Antoine_extended_dT2(T=180.0, Tc=227.51, to=-120., A=8.95894,
... B=510.595, C=-15.95, n=2.41377, E=-93.74, F=7425.9)
1022.550368944
```

chemicals.vapor_pressure.**dYaws_Psat_dT**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculates the first temperature derivative of vapor pressure of a chemical using the Yaws equation for vapor pressure. Parameters *A*, *B*, *C*, *D*, and *E* are chemical-dependent. Parameters can be found in numerous sources; however units of the coefficients used vary.

$$\frac{\partial P^{\text{sat}}}{\partial T} = 10^{A+\frac{B}{T}+\frac{C\log(T)}{\log(10)}+DT+ET^2}\left(-\frac{B}{T^2}+\frac{C}{T\log(10)}+D+2ET\right)\log(10)$$

Parameters

T [float] Temperature of fluid, [K]

A [float] *A* parameter, [-]

B [float] *B* parameter, [K]

C [float] *C* parameter, [-]

D [float] *D* parameter, [1/K]

E [float] *E* parameter, [1/K^2]

Returns

dPsat_dT [float] First temperature derivative of vapor pressure calculated with coefficients [Pa/K]

Examples

Benzene:

```
>>> dYaws_Psat_dT(T=400.0, A=42.7918, B=-2965.83, C=-12.073, D=0.0033269, E=1.
↪58609e-6)
8134.87548930
```

chemicals.vapor_pressure.**d2Yaws_Psat_dT2**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculates the second temperature derivative of vapor pressure of a chemical using the Yaws equation for vapor pressure. Parameters *A*, *B*, *C*, *D*, and *E* are chemical-dependent. Parameters can be found in numerous sources;

however units of the coefficients used vary.

$$\frac{\partial^2 P^{\text{sat}}}{\partial T^2} = 10^{A+\frac{B}{T}+\frac{C\log(T)}{\log(10)}+DT+ET^2}\left(\frac{2B}{T^3} - \frac{C}{T^2\log(10)} + 2E + \left(-\frac{B}{T^2} + \frac{C}{T\log(10)} + D + 2ET\right)^2 \log(10)\right)\log(10)$$

Parameters

T [float] Temperature of fluid, [K]

A [float] *A* parameter, [-]

B [float] *B* parameter, [K]

C [float] *C* parameter, [-]

D [float] *D* parameter, [1/K]

E [float] *E* parameter, [1/K^2]

Returns

d2Psat_dT2 [float] Second temperature derivative of vapor pressure calculated with coefficients [Pa/K^2]

Examples

Benzene:

```
>>> d2Yaws_Psat_dT2(T=400.0, A=42.7918, B=-2965.83, C=-12.073, D=0.0033269, E=1.
↪58609e-6)
141.7181045862
```

1.32.3 Jacobians (for fitting)

chemicals.vapor_pressure.**Wagner_fitting_jacobian**(*Ts*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates the jacobian of the Wagner (2.5, 5) vapor pressure equation for use in fitting these parameters when experimental values are known.

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

Parameters

Ts [list[float]] Temperatures of fluid data points, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

jac [list[list[float, 4], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.vapor_pressure.**Wagner_original_fitting_jacobian**(*Ts*, *Tc*, *Pc*, *a*, *b*, *c*, *d*)

Calculates the jacobian of the Wagner (3, 6) vapor pressure equation for use in fitting these parameters when experimental values are known.

Requires critical temperature and pressure as well as four coefficients specific to each chemical.

Parameters

Ts [list[float]] Temperatures of fluid data points, [K]

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

a, b, c, d [floats] Parameters for wagner equation. Specific to each chemical. [-]

Returns

jac [list[list[float, 4], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.vapor_pressure.**Antoine_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *base=10.0*)

Calculates the jacobian of the Antoine vapor pressure equation for use in fitting these parameters when experimental values are known.

Requires three coefficients specific to each chemical.

Parameters

Ts [list[float]] Temperatures of fluid data points, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

base [float, optional] Optional base of logarithm; 10 by default, [-]

Returns

jac [list[list[float, 3], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.vapor_pressure.**Yaws_Psat_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *D*, *E*)

Compute and return the Jacobian of the property predicted by the Yaws vapor pressure equation with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

A [float] *A* parameter, [-]

B [float] *B* parameter, [K]

C [float] *C* parameter, [-]

D [float] *D* parameter, [1/K]

E [float] *E* parameter, [1/K^2]

Returns

jac [list[list[float, 5], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.vapor_pressure.**TRC_Antoine_extended_fitting_jacobian**(*Ts*, *Tc*, *to*, *A*, *B*, *C*, *n*, *E*, *F*)

Calculates the jacobian of the TRC Antoine extended vapor pressure equation for use in fitting these parameters when experimental values are known.

Requires 7 coefficients specific to each chemical.

Parameters

Ts [list[float]] Temperatures of fluid data points, [K]

Tc [float] Critical temperature of fluid, [K]

to [float] Fit temperature-transition parameter, [K]

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

n [float] Fit parameter, [-]

E [float] Fit parameter, [-]

F [float] Fit parameter, [-]

Returns

jac [list[list[float, 7], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

1.32.4 Vapor Pressure Estimation Correlations

chemicals.vapor_pressure.**Lee_Kesler**(*T*, *Tc*, *Pc*, *omega*)

Calculates vapor pressure of a fluid at arbitrary temperatures using a CSP relationship by [1]; requires a chemical's critical temperature and acentric factor.

The vapor pressure is given by:

$$\ln P_r^{sat} = f^{(0)} + \omega f^{(1)}$$

$$f^{(0)} = 5.92714 - \frac{6.09648}{T_r} - 1.28862 \ln T_r + 0.169347 T_r^6$$

$$f^{(1)} = 15.2518 - \frac{15.6875}{T_r} - 13.4721 \ln T_r + 0.43577 T_r^6$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor [-]

Returns

Psat [float] Vapor pressure at T [Pa]

Notes

This equation appears in [1] in expanded form. The reduced pressure form of the equation ensures predicted vapor pressure cannot surpass the critical pressure.

References

[1], [2]

Examples

Example from [2]; ethylbenzene at 347.2 K.

```
>>> Lee_Kesler(347.2, 617.1, 36E5, 0.299)
13078.694162949312
```

chemicals.vapor_pressure.**Ambrose_Walton**(*T*, *Tc*, *Pc*, *omega*)

Calculates vapor pressure of a fluid at arbitrary temperatures using a CSP relationship by [1]; requires a chemical's critical temperature and acentric factor.

The vapor pressure is given by:

$$\ln P_r = f^{(0)} + \omega f^{(1)} + \omega^2 f^{(2)}$$

$$f^{(0)} = \frac{-5.97616\tau + 1.29874\tau^{1.5} - 0.60394\tau^{2.5} - 1.06841\tau^5}{T_r}$$

$$f^{(1)} = \frac{-5.03365\tau + 1.11505\tau^{1.5} - 5.41217\tau^{2.5} - 7.46628\tau^5}{T_r}$$

$$f^{(2)} = \frac{-0.64771\tau + 2.41539\tau^{1.5} - 4.26979\tau^{2.5} + 3.25259\tau^5}{T_r}$$

$$\tau = 1 - T_r$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor [-]

Returns

Psat [float] Vapor pressure at T [Pa]

Notes

Somewhat more accurate than the *Lee_Kesler* formulation.

References

[1], [2]

Examples

Example from [2]; ethylbenzene at 347.25 K.

```
>>> Ambrose_Walton(347.25, 617.15, 36.09E5, 0.304)
13278.878504306222
```

chemicals.vapor_pressure.**boiling_critical_relation**(*T*, *Tb*, *Tc*, *Pc*)

Calculates vapor pressure of a fluid at arbitrary temperatures using a CSP relationship as in [1]; requires a chemical's critical temperature and pressure as well as boiling point.

The vapor pressure is given by:

$$\ln P_r^{sat} = h\left(1 - \frac{1}{T_r}\right)$$

$$h = T_{br}\frac{\ln(P_c/101325)}{1 - T_{br}}$$

Parameters

T [float] Temperature of fluid [K]

Tb [float] Boiling temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Returns

Psat [float] Vapor pressure at T [Pa]

Notes

Units are Pa. Formulation makes intuitive sense; a logarithmic form of interpolation.

References

[1]

Examples

Example as in [1] for ethylbenzene

```
>>> boiling_critical_relation(347.2, 409.3, 617.1, 36E5)
15209.467273093938
```

chemicals.vapor_pressure.**Sanjari**(*T*, *Tc*, *Pc*, *omega*)

Calculates vapor pressure of a fluid at arbitrary temperatures using a CSP relationship by [1]. Requires a chemical's critical temperature, pressure, and acentric factor. Although developed for refrigerants, this model should have some general predictive ability.

The vapor pressure of a chemical at T is given by:

$$P^{sat} = P_c \exp(f^{(0)} + \omega f^{(1)} + \omega^2 f^{(2)})$$

$$f^{(0)} = a_1 + \frac{a_2}{T_r} + a_3 \ln T_r + a_4 T_r^{1.9}$$

$$f^{(1)} = a_5 + \frac{a_6}{T_r} + a_7 \ln T_r + a_8 T_r^{1.9}$$

$$f^{(2)} = a_9 + \frac{a_{10}}{T_r} + a_{11} \ln T_r + a_{12} T_r^{1.9}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor [-]

Returns

Psat [float] Vapor pressure, [Pa]

Notes

a[1-12] are as follows: 6.83377, -5.76051, 0.90654, -1.16906, 5.32034, -28.1460, -58.0352, 23.57466, 18.19967, 16.33839, 65.6995, -35.9739.

For a claimed fluid not included in the regression, R128, the claimed AARD was 0.428%. A re-calculation using 200 data points from 125.45 K to 343.90225 K evenly spaced by 1.09775 K as generated by NIST Webbook April 2016 produced an AARD of 0.644%. It is likely that the author's regression used more precision in its coefficients than was shown here. Nevertheless, the function is reproduced as shown in [1].

For Tc=808 K, Pc=1100000 Pa, omega=1.1571, this function actually declines after 770 K.

References

[1]

Examples

```
>>> Sanjari(347.2, 617.1, 36E5, 0.299)
13651.916109552523
```

chemicals.vapor_pressure.**Edalat**(*T*, *Tc*, *Pc*, *omega*)

Calculates vapor pressure of a fluid at arbitrary temperatures using a CSP relationship by [1]. Requires a chemical's critical temperature, pressure, and acentric factor. Claimed to have a higher accuracy than the Lee-Kesler CSP relationship.

The vapor pressure of a chemical at T is given by:

$$\ln(P^{sat}/P_c) = \frac{a\tau + b\tau^{1.5} + c\tau^3 + d\tau^6}{1-\tau}$$

$$a = -6.1559 - 4.0855\omega$$

$$b = 1.5737 - 1.0540\omega - 4.4365 \times 10^{-3} d$$

$$c = -0.8747 - 7.8874\omega$$

$$d = \frac{1}{-0.4893 - 0.9912\omega + 3.1551\omega^2}$$

$$\tau = 1 - \frac{T}{T_c}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor [-]

Returns

Psat [float] Vapor pressure, [Pa]

Notes

[1] found an average error of 6.06% on 94 compounds and 1106 data points.

References

[1]

Examples

```
>>> Edalat(347.2, 617.1, 36E5, 0.299)
13461.273080743307
```

1.32.5 Sublimation Pressure Estimation Correlations

chemicals.vapor_pressure.**Psub_Clapeyron**(*T*, *Tt*, *Pt*, *Hsub_t*)

Calculates sublimation pressure of a solid at arbitrary temperatures using an approximate themodynamic identity - the Clapeyron equation as described in [1] and [2]. Requires a chemical's triple temperature, triple pressure, and triple enthalpy of sublimation.

The sublimation pressure of a chemical at T is given by:

$$\ln \frac{P}{P_{tp}} = -\frac{\Delta H_{sub}}{R}\left(\frac{1}{T} - \frac{1}{T_{tp}}\right)$$

Parameters

T [float] Temperature of solid [K]

Tt [float] Triple temperature of solid [K]

Pt [float] Truple pressure of solid [Pa]

Hsub_t [float] Enthalpy of fusion at the triple point of the chemical, [J/mol]

Returns

Psub [float] Sublimation pressure, [Pa]

Notes

Does not seem to capture the decrease in sublimation pressure quickly enough.

References

[1], [2]

Examples

```
>>> Psub_Clapeyron(250, Tt=273.15, Pt=611.0, Hsub_t=51100.0)
76.06457150831804
>>> Psub_Clapeyron(300, Tt=273.15, Pt=611.0, Hsub_t=51100.0)
4577.282832876156
```

1.32.6 Correlations for Specific Substances

chemicals.vapor_pressure.**Psat_IAPWS**(*T*)

Calculates vapor pressure of water using the IAPWS explicit equation.

$$P^{sat} = 10^6 \left[\frac{2C}{-B + \sqrt{B^2 - 4AC}}\right]^4$$

$$A = \nu^2 + n_1\nu + n_2$$

$$B = n_3\nu^2 + n_4\nu + n_5$$

$$C = n_6\nu^2 + n_7\nu + n_8$$

$$\nu = T + \frac{n_9}{T - n_{10}}$$

Parameters

T [float] Temperature of water, [K]

Returns

Psat [float] Vapor pressure at T [Pa]

Notes

This formulation is quite efficient, and can also be solved backward. The range of validity of this equation is 273.15 K < T < 647.096 K, the IAPWS critical point.

Extrapolation to lower temperatures is very poor. The function continues to decrease until a pressure of 5.7 mPa is reached at 159.77353993926621 K; under that pressure the vapor pressure increases, which is obviously wrong.

References

[1]

Examples

```
>>> Psat_IAPWS(300.)
3536.58941301301
```

chemicals.vapor_pressure.**dPsat_IAPWS_dT**(*T*)

Calculates the first temperature dervative of vapor pressure of water using the IAPWS explicit equation. This was derived with SymPy, using the CSE method.

Parameters

T [float] Temperature of water, [K]

Returns

dPsat_dT [float] Temperature dervative of vapor pressure at T [Pa/K]

Notes

The derivative of this is useful when solving for water dew point.

References

[1]

Examples

```
>>> dPsat_IAPWS_dT(300.)
207.88388134164282
```

chemicals.vapor_pressure.**Tsat_IAPWS**(*P*)

Calculates the saturation temperature of water using the IAPWS explicit equation.

$$T_s = \frac{n_{10} + D - \left[(n_{10} + D)^2 - 4(n_9 + n_{10}D)\right]^{0.5}}{2}$$

$$E = \beta^2 + n_3\beta + n_6$$

$$F = n_1\beta^2 + n_4\beta + n_7$$

$$G = n_2\beta^2 + n_5\beta + n_8$$

$$\beta = (P_{sat})^{0.25}$$

Parameters

Psat [float] Vapor pressure at T [Pa]

Returns

T [float] Temperature of water along the saturation curve at *Psat*, [K]

Notes

The range of validity of this equation is 273.15 K < T < 647.096 K, the IAPWS critical point.

The coefficients *n1* to *n10* are (0.11670521452767E4, -0.72421316703206E6, -0.17073846940092E2, 0.12020824702470E5, -0.32325550322333E7, 0.14915108613530E2, -0.48232657361591E4, 0.40511340542057E6, -0.23855557567849, 0.65017534844798E3)

References

[1]

Examples

```
>>> Tsat_IAPWS(1E5)
372.75591861133773
```

1.32.7 Analytical Fit Equations

chemicals.vapor_pressure.**Antoine_coeffs_from_point**(*T*, *Psat*, *dPsat_dT*, *d2Psat_dT2*, *base=10.0*)

Calculates the antoine coefficients *A*, *B*, and *C* from a known vapor pressure and its first and second temperature derivative.

Parameters

T [float] Temperature of fluid, [K]

Psat [float] Vapor pressure at specified *T* [Pa]

dPsat_dT [float] First temperature derivative of vapor pressure at specified *T* [Pa/K]

d2Psat_dT2 [float] Second temperature derivative of vapor pressure at specified *T* [Pa/K^2]

Base [float, optional] Base of logarithm; 10 by default

Returns

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

C [float] Antoine *C* parameter, [K]

Notes

Coefficients are for calculating vapor pressure in Pascal. This is primarily useful for interconverting vapor pressure models, not fitting experimental data.

Derived with SymPy as follows:

```
>>> from sympy import *
>>> base, A, B, C, T = symbols('base, A, B, C, T')
>>> v = base**(A - B/(T + C))
>>> d1, d2 = diff(v, T), diff(v, T, 2)
>>> vk, d1k, d2k = symbols('vk, d1k, d2k')
>>> solve([Eq(v, vk), Eq(d1, d1k), Eq(d2, d2k)], [A, B, C])
```

References

[1]

Examples

Recalculate some coefficients from a calcualted value and its derivative:

```
>>> T = 178.01
>>> A, B, C = (24.0989474955895, 4346.793091137991, -18.96968471040141)
>>> Psat = Antoine(T, A, B, C, base=exp(1))
>>> dPsat_dT, d2Psat_dT2 = (0.006781441203850251, 0.0010801244983894853) #␣
↪precomputed
>>> Antoine_coeffs_from_point(T, Psat, dPsat_dT, d2Psat_dT2, base=exp(1))
(24.098947495155, 4346.793090994, -18.969684713118)
```

chemicals.vapor_pressure.**Antoine_AB_coeffs_from_point**(*T*, *Psat*, *dPsat_dT*, *base=10.0*)

Calculates the antoine coefficients *A*, *B*, with *C* set to zero to improve low-temperature or high-temperature extrapolation, from a known vapor pressure and its first temperature derivative.

Parameters

T [float] Temperature of fluid, [K]

Psat [float] Vapor pressure at specified *T* [Pa]

dPsat_dT [float] First temperature derivative of vapor pressure at specified *T* [Pa/K]

Base [float, optional] Base of logarithm; 10 by default

Returns

A [float] Antoine *A* parameter, [-]

B [float] Antoine *B* parameter, [K]

Notes

Coefficients are for calculating vapor pressure in Pascal. This is primarily useful for interconverting vapor pressure models, not fitting experimental data.

Derived with SymPy as follows:

```
>>> from sympy import *
>>> base, A, B, T = symbols('base, A, B, T')
>>> v = base**(A - B/T)
>>> d1, d2 = diff(v, T), diff(v, T, 2)
>>> vk, d1k = symbols('vk, d1k')
>>> solve([Eq(v, vk), Eq(d1, d1k)], [A, B])
```

References

[1]

Examples

Recalculate some coefficients from a calcualted value and its derivative:

```
>>> T = 178.01
>>> A, B = (27.358925161569008, 5445.569591293226)
>>> Psat = Antoine(T, A, B, C=0, base=exp(1))
>>> dPsat_dT = B*exp(1)**(A - B/T)*log(exp(1))/T**2
>>> Antoine_AB_coeffs_from_point(T, Psat, dPsat_dT, base=exp(1))
(27.35892516156901, 5445.569591293226)
```

chemicals.vapor_pressure.**DIPPR101_ABC_coeffs_from_point**(*T*, *Psat*, *dPsat_dT*, *d2Psat_dT2*)

Calculates the first three DIPPR101 coefficients *A*, *B*, and *C* from a known vapor pressure and its first and second temperature derivative.

If the second derivative is infinity as is the case in some vapor pressure models at the critical point, only the *A* and *C* coefficients are fit, using the first derivative an the actual value of vapor pressure.

Parameters

T [float] Temperature of fluid, [K]

Psat [float] Vapor pressure at specified *T* [Pa]

dPsat_dT [float] First temperature derivative of vapor pressure at specified *T* [Pa/K]

d2Psat_dT2 [float] Second temperature derivative of vapor pressure at specified *T* [Pa/K^2]

Returns

A [float] DIPPR101 *A* parameter (same as Antoine *A*), [-]

B [float] DIPPR101 *B* parameter (same as Antoine *B*), [K]

C: float DIPPR101 *C* parameter (NOT same as Antoine *C*, multiplied by log(T)), [-]

Notes

Coefficients are for calculating vapor pressure in Pascal. This is primarily useful for interconverting vapor pressure models, not fitting experimental data.

Derived with SymPy as follows:

```
>>> from sympy import *
>>> base, A, B, C, T = symbols('base, A, B, C, T')
>>> v = exp(A + B/T + C*log(T))
>>> d1, d2 = diff(v, T), diff(v, T, 2)
>>> vk, d1k, d2k = symbols('vk, d1k, d2k')
>>> solve([Eq(v, vk), Eq(d1, d1k), Eq(d2, d2k)], [A, B, C])
```

Examples

Calculate the coefficients:

```
>>> T = 178.01
>>> Psat, dPsat_dT, d2Psat_dT2 = (0.03946094565666715, 0.006781441203850251, 0.
↪0010801244983894853)
>>> DIPPR101_ABC_coeffs_from_point(T, Psat, dPsat_dT, d2Psat_dT2)
(72.47169926642, -6744.620564969, -7.2976291987890)
```

1.32.8 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.vapor_pressure.**Psat_data_WagnerMcGarry**
: Coefficients for the Wagner 3,6 original model equation documented in *Wagner_original* with data for 245 chemicals, from [1].

chemicals.vapor_pressure.**Psat_data_WagnerPoling**
: Coefficients for the Wagner 2.5, 5 model equation documented in *Wagner* in [2], with data for 104 chemicals.

chemicals.vapor_pressure.**Psat_data_AntoinePoling**
: Standard Antoine equation coefficients, as documented in the function *Antoine* and with data for 325 fluids from [2]. Coefficients were altered to be in units of Pa and Celcius.

chemicals.vapor_pressure.**Psat_data_AntoineExtended**
: Data for 97 chemicals in [2] for the TRC extended Antoine model *TRC_Antoine_extended*.

chemicals.vapor_pressure.**Psat_data_Perrys2_8**
: A collection of 341 coefficient sets for `thermo.dippr.EQ101` from the DIPPR database published openly in [4].

chemicals.vapor_pressure.**Psat_data_VDI_PPDS_3**
: Coefficients for the Wagner equation *Wagner*, published openly in [3].

chemicals.vapor_pressure.**Psat_data_Alcock_elements**
: Coefficients for the DIPPR 101 equation *chemicals.dippr.EQ101*, published in [5] and converted to provide base SI units (and use the natural logarithm).

The structure of each dataframe is shown below:

```
In [1]: import chemicals

In [2]: chemicals.vapor_pressure.Psat_data_WagnerMcGarry
Out[2]:
                                  Name        A  ...      Tc   Tmin
CAS                                              ...
50-00-0                   formaldehyde -7.29343  ...  408.00  184.0
56-23-5           carbon tetrachloride -7.07139  ...  556.40  250.0
60-29-7                   diethylether -7.29916  ...  466.74  250.0
62-53-3                        aniline -7.65517  ...  699.00  376.0
64-17-5                        ethanol -8.51838  ...  513.92  293.0
...                                ...      ...  ...     ...    ...
7732-18-5                        water -7.76451  ...  647.35  275.0
7782-41-4                     fluorine -6.18224  ...  144.31   64.0
7782-44-7                       oxygen -6.28275  ...  154.70   54.0
```

(continues on next page)

(continued from previous page)

```
16747-38-9    2,3,3,4-tetramethylpentane -7.65000  ...  607.60  332.0
16747-50-5  1-methyl-1-ethylcyclopentane -7.09092  ...  592.00  316.0

[245 rows x 8 columns]

In [3]: chemicals.vapor_pressure.Psat_data_WagnerPoling
Out[3]:
                                                    Name  ...    Tmax
CAS                                                       ...
60-29-7                                     diethyl ether  ...  466.74
64-17-5                                           ethanol  ...  513.92
64-18-6                                     methanoic acid  ...  588.00
64-19-7                                      ethanoic acid  ...  592.71
67-56-1                                           methanol  ...  512.64
...                                                    ...  ...     ...
7727-37-9                                         nitrogen  ...  126.20
7783-81-5                                uranium hexafluoride  ...  503.35
13838-16-9  2-chloro-1,1,2-trifluoroethyl difluoromethyl e...  ...  475.03
26171-83-5                                   1,2-butandiol  ...  506.40
26675-46-7  1-chloro-2,2,2-trifluoroethyl difluoromethyl e...  ...  467.80

[104 rows x 9 columns]

In [4]: chemicals.vapor_pressure.Psat_data_AntoinePoling
Out[4]:
                            Chemical         A  ...    Tmin    Tmax
CAS                                            ...
56-23-5          tetrachloromethane    9.10445  ...  259.00  373.76
60-29-7               diethyl ether    9.10962  ...  229.71  328.31
62-53-3                benzeneamine    9.40870  ...  349.86  484.81
64-17-5                     ethanol   10.33675  ...  276.50  369.54
64-19-7               ethanoic acid    9.54456  ...  297.58  414.97
...                             ...       ...  ...     ...     ...
14762-55-1                 helium-3    6.39750  ...    1.12    4.41
16747-38-9  2,3,3,4-tetramethylpentane    8.99105  ...  307.81  443.27
20291-95-6    2,2,5-trimethylheptane    9.00345  ...  318.00  452.00
800000-51-5         hydrogen, normal    7.94928  ...   13.33   22.94
800000-54-8       deuterium, normal    8.25315  ...   17.57   26.23

[325 rows x 6 columns]

In [5]: chemicals.vapor_pressure.Psat_data_AntoineExtended
Out[5]:
                                 Chemical        A  ...    Tmin    Tmax
CAS                                                 ...
62-53-3                       benzeneamine   9.40870  ...  488.15  673.15
74-85-1                             ethene   8.91382  ...  188.15  273.15
74-89-5                        methanamine   9.21300  ...  288.15  423.15
75-04-7                        ethanamine   8.88560  ...  308.15  443.15
75-10-5                   difluoromethane   9.29712  ...  238.15  338.15
...                                   ...       ...  ...     ...     ...
1067-08-9          3-ethyl-3-methylpentane   8.98950  ...  408.15  543.15
```

(continues on next page)

(continued from previous page)

```
1511-62-2                bromodifluoromethane   8.40030  ...  273.15  403.15
1640-89-7                   ethylcyclopentane   9.00408  ...  408.15  569.52
1717-00-6         1,1-dichloro-1-fluoroethane   9.03117  ...  333.15  473.15
2837-89-0  1-chloro-1,2,2,2-tetrafluoroethane   8.98581  ...  283.15  383.15

[97 rows x 11 columns]

In [6]: chemicals.vapor_pressure.Psat_data_Perrys2_8
Out[6]:
                          Chemical       C1        C2  ...   C5    Tmin    Tmax
CAS                                                    ...
50-00-0                Formaldehyde  101.510  -4917.20  ...  1.0  181.15  408.00
55-21-0                   Benzamide   85.474 -11932.00  ...  6.0  403.00  824.00
56-23-5       Carbon tetrachloride    78.441  -6128.10  ...  2.0  250.33  556.35
57-55-6      1,2-Propylene glycol   212.800 -15420.00  ...  2.0  213.15  626.00
60-29-7              Diethyl ether   136.900  -6954.30  ...  1.0  156.85  466.70
...                            ...      ...       ...  ...  ...     ...     ...
10028-15-6                  Ozone    40.067  -2204.80  ...  6.0   80.15  261.00
10035-10-6       Hydrogen bromide    29.315  -2424.50  ...  6.0  185.15  363.15
10102-43-9           Nitric oxide    72.974  -2650.00  ...  6.0  109.50  180.15
13511-13-2    Propenylcyclohexene    64.268  -7298.90  ...  6.0  199.00  636.00
132259-10-0                   Air    21.662   -692.39  ...  1.0   59.15  132.45

[340 rows x 8 columns]

In [7]: chemicals.vapor_pressure.Psat_data_VDI_PPDS_3
Out[7]:
                         Chemical      Tm      Tc  ...        B        C        D
CAS                                                ...
50-00-0               Formaldehyde  181.15  408.05  ...  1.28290 -0.50464 -4.29089
56-23-5       Carbon tetrachloride  250.25  556.35  ...  1.96174 -2.05900 -3.26771
56-81-5                   Glycerol  291.45  850.05  ... -0.33345 -5.98569 -1.33011
60-29-7               Diethyl ether  156.75  466.63  ...  2.15613 -3.02766 -2.37858
62-53-3                     Aniline  267.15  699.05  ...  1.96206 -3.65571 -2.00622
...                             ...     ...     ...  ...      ...      ...      ...
10097-32-2                  Bromine  265.85  584.15  ...  1.50339 -0.64097 -3.62166
10102-43-9             Nitric oxide  112.15  180.15  ...  0.85755 -3.11447 -8.98765
10102-44-0         Nitrogen dioxide  261.85  431.15  ...  2.37620  0.67820 -2.53997
10544-72-6      Dinitrogentetroxide  261.85  431.10  ...  3.10196  0.59704 -5.33648
132259-10-0                     Air   63.05  132.53  ... -0.21537  0.93623 -3.02641

[275 rows x 8 columns]

In [8]: chemicals.vapor_pressure.Psat_data_Alcock_elements
Out[8]:
                  name           A             B  ...    E       Tmin    Tmax
CAS                                               ...
7439-93-2       lithium   30.888526 -19157.507974  ... -3.0   453.6500  1000.0
7440-23-5        sodium   30.867803 -12972.764414  ... -3.0   370.9440   700.0
7440-09-7     potassium   30.483272 -10806.031841  ... -3.0   336.6500   600.0
7440-17-7      Rubidium   30.674386  -9843.551273  ... -3.0   312.4500   550.0
7440-46-2       Caesium   30.480969  -9353.100648  ... -3.0   301.6500   550.0
```

(continues on next page)

(continued from previous page)

```
7429-90-5     Aluminium   35.882834 -39019.606986  ... -3.0   933.4730  1800.0
7440-55-3       gallium   19.870657 -31842.449251  ... -3.0   302.9146  1600.0
7440-74-6        Indium   34.365430 -28938.889449  ... -3.0   429.7500  1500.0
7440-28-0      Thallium   31.392793 -21605.155928  ... -3.0   577.1500  1100.0
7440-31-5           Tin   17.786817 -34785.153000  ... -3.0   505.0780  1850.0
7439-92-1          Lead   31.171744 -23239.991344  ... -3.0   600.6120  1200.0
7440-65-5       Yttrium   43.175121 -51151.927841  ... -3.0  1795.1500  2300.0
7439-91-0     Lanthanum   26.548154 -50603.912589  ... -3.0  1193.1500  2450.0
7440-32-6      Titanium   49.219406 -58091.919311  ... -3.0  1943.1500  2400.0
7440-67-7     Zirconium   15.173383 -66231.557615  ... -3.0  2127.1500  2500.0
7440-06-4      Platinum   60.472140 -71198.233660  ... -3.0  2041.3500  2500.0
7440-50-8        Copper   37.335765 -40127.150416  ... -3.0  1357.7700  1850.0
7440-57-5          Gold   35.238110 -43514.253087  ... -3.0  1337.3300  2050.0
7440-45-1        Cerium   25.394558 -48994.405609  ... -3.0  1072.1500  2450.0
7440-10-0  Praseodymium   38.965995 -43042.223143  ... -3.0  1204.1500  2200.0
7440-00-8     Neodymium   40.068933 -39717.290269  ... -3.0  1289.1500  2000.0
7440-54-2    Gadolinium   35.947306 -47214.507332  ... -3.0  1586.1500  2250.0
7440-27-9       Terbium   38.703500 -46171.436285  ... -3.0  1632.1500  2200.0
7439-94-3      Lutetium   54.932120 -54202.853089  ... -3.0  1936.1500  2350.0
7440-29-1       Thorium  148.700293 -85151.899324  ... -3.0  2023.1500  2500.0
7440-13-3  Protactinium   35.081534 -78331.642279  ... -3.0  1845.1500  2500.0
7439-99-8     Neptunium   48.979938 -55303.488764  ... -3.0   917.1500  2500.0
7440-07-5     Plutonium   41.441274 -40495.564030  ... -3.0   913.1500  2450.0
7440-51-9        Curium   56.511693 -49353.608883  ... -3.0  1618.1500  2200.0
7440-41-7     Beryllium   24.848846 -36221.966098  ... -3.0  1560.1500  1800.0
7440-39-3        Barium   20.752547 -18796.002114  ... -3.0  1000.1500  1200.0
7440-20-2      Scandium   24.869569 -40712.007029  ... -3.0  1814.1500  2000.0
7440-62-2      Vanadium   27.480701 -57589.955761  ... -3.0  2183.1500  2500.0
7439-89-6          Iron   26.140596 -45070.800610  ... -3.0  1811.1500  2100.0
7440-48-4        Cobalt   26.465261 -47382.596044  ... -3.0  1768.1500  2150.0
7440-02-0        Nickel   26.875121 -47813.179456  ... -3.0  1728.1500  2150.0
7440-05-3     Palladium   24.019915 -41213.970580  ... -3.0  1827.9500  2100.0
7440-22-4        Silver   24.770558 -31837.844081  ... -3.0  1234.9300  1600.0
7440-66-6          Zinc   23.909391 -14474.049895  ... -3.0   692.6770   750.0
7440-43-9       Cadmium   23.596240 -12415.538821  ... -3.0   594.2190   650.0
7439-97-6       Mercury   23.306114  -7345.246447  ... -3.0   298.0000   400.0
7440-52-0        Erbium   22.320607 -33111.173637  ... -3.0  1802.1500  1900.0
7440-61-1       Uranium   59.270190 -66259.188636  ... -3.0  1408.1500  2500.0

[43 rows x 8 columns]
```

1.33 Support for Numpy Arrays (chemicals.vectorized)

Basic module which wraps all chemicals functions with numpy's np.vectorize function.

All other object - dicts, classes, etc - are not wrapped. Supports star imports; so the same objects exported when importing from the main library will be imported from here.

```
>>> from chemicals.vectorized import *
>>> Antoine(np.linspace(100, 200, 5), A=8.95894, B=510.595, C=-15.95)
```

(continues on next page)

(continued from previous page)

```
array([7.65674361e+02, 1.89116754e+04, 1.41237759e+05, 5.60609191e+05,
       1.53010431e+06])
```

Inputs do not need to be numpy arrays; they can be any iterable:

```
>>> import chemicals.vectorized
>>> chemicals.vectorized.Tc(['108-88-3', '7732-18-5'])
array([591.75 , 647.096])
```

Warning: This module does not replace the functions in the *chemicals* module; it copies all the functions into the *chemicals.vectorized* module and makes them vectorized there.

For example by importing *chemicals.vectorized*, *chemicals.Antoine* won't become vectorized, but *chemicals.vectorized.Antoine* will become available and is vectorized.

Warning: *np.vectorize* does not use NumPy to accelerate any computations; it is a convenience wrapper. If you are working on a problem large enough for speed to be an issue and Numba is compatible with your version of Python, an interface to that library is available at `chemicals.numba` which does accelerate NumPy array computations and is normally faster than using numpy directly.

1.34 Virial Coefficients (chemicals.virial)

This module contains four estimation methods for second B virial coefficients, two utility covnersions for when only B is considered, and two methods to calculate Z from higher order virial expansions.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.34.1 Utilities

`chemicals.virial.B_to_Z`(B, T, P)

Calculates the compressibility factor of a gas, given its second virial coefficient.

$$Z = \frac{PV}{RT} = 1 + \frac{BP}{RT}$$

Parameters

B [float] Second virial coefficient, [m^3/mol]

T [float] Temperature, [K]

P [float] Pressure [Pa]

Returns

Z [float] Compressibility factor, [-]

Notes

Other forms of the virial coefficient exist.

References

[1]

Examples

```
>>> B_to_Z(-0.0015, 300, 1E5)
0.939863822478637
```

chemicals.virial.**B_from_Z**(*Z*, *T*, *P*)

Calculates the second virial coefficient of a pure species, given the compressibility factor of the gas.

$$B = \frac{RT(Z-1)}{P}$$

Parameters

Z [float] Compressibility factor, [-]

T [float] Temperature, [K]

P [float] Pressure [Pa]

Returns

B [float] Second virial coefficient, [m^3/mol]

Notes

Other forms of the virial coefficient exist.

References

[1]

Examples

```
>>> B_from_Z(0.94, 300, 1E5)
-0.0014966032712675846
```

chemicals.virial.**Z_from_virial_density_form**(*T*, *P*, **args*)

Calculates the compressibility factor of a gas given its temperature, pressure, and molar density-form virial coefficients. Any number of coefficients is supported.

$$Z = \frac{PV}{RT} = 1 + \frac{B}{V} + \frac{C}{V^2} + \frac{D}{V^3} + \frac{E}{V^4} \dots$$

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

B to Z [float, optional] Virial coefficients, [various]

Returns

Z [float] Compressibility factor at T, P, and with given virial coefficients, [-]

Notes

For use with B or with B and C or with B and C and D, optimized equations are used to obtain the compressibility factor directly. If more coefficients are provided, uses numpy's roots function to solve this equation. This takes substantially longer as the solution is numerical.

If no virial coefficients are given, returns 1, as per the ideal gas law.

The units of each virial coefficient are as follows, where for B, n=1, and C, n=2, and so on.

$$\left(\frac{\text{m}^3}{\text{mol}}\right)^n$$

References

[1], [2]

Examples

```
>>> Z_from_virial_density_form(300, 122057.233762653, 1E-4, 1E-5, 1E-6, 1E-7)
1.28434940526
```

chemicals.virial.**Z_from_virial_pressure_form**(*P*, **args*)

Calculates the compressibility factor of a gas given its pressure, and pressure-form virial coefficients. Any number of coefficients is supported.

$$Z = \frac{Pv}{RT} = 1 + B'P + C'P^2 + D'P^3 + E'P^4 \dots$$

Parameters

P [float] Pressure, [Pa]

B to Z [float, optional] Pressure form Virial coefficients, [various]

Returns

Z [float] Compressibility factor at P, and with given virial coefficients, [-]

Notes

Note that although this function does not require a temperature input, it is still dependent on it because the coefficients themselves normally are regressed in terms of temperature.

The use of this form is less common than the density form. Its coefficients are normally indicated with the "" suffix.

If no virial coefficients are given, returns 1, as per the ideal gas law.

The units of each virial coefficient are as follows, where for B, n=1, and C, n=2, and so on.

$$\left(\frac{1}{\text{Pa}}\right)^n$$

References

[1], [2]

Examples

```
>>> Z_from_virial_pressure_form(102919.99946855308, 4.032286555169439e-09, 1.
↪6197059494442215e-13, 6.483855042486911e-19)
1.00283753944
```

chemicals.virial.**BVirial_mixture**(*zs*, *Bijs*)

Calculate the *B* second virial coefficient from a matrix of virial cross-coefficients. The diagonal is virial coefficients of the pure components.

$$B = \sum_i \sum_j y_i y_j B(T)$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Bijs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

References

[1]

Examples

```
>>> Bijs = [[-6.24e-06, -2.013e-05, -3.9e-05], [-2.01e-05, -4.391e-05, -6.46e-05],␣
↪[-3.99e-05, -6.46e-05, -0.00012]]
>>> zs = [.5, .3, .2]
>>> BVirial_mixture(zs=zs, Bijs=Bijs)
-3.19884e-05
```

chemicals.virial.**dBVirial_mixture_dzs**(*zs*, *Bijs*, *dB_dzs=None*)

Calculate first mole fraction derivative of the *B* second virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial B}{\partial x_i} = \sum_j z_j (B_{i,j} + B_{j,i})$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Bijs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dzs [list[float], optional] Array for first mole fraction derivatives of second virial coefficient in density form [m^3/mol]

Returns

dB_dzs [list[float]] First mole fraction derivatives of second virial coefficient in density form [m^3/mol]

Examples

```
>>> Bijs = [[-6.24e-06, -2.013e-05, -3.9e-05], [-2.01e-05, -4.391e-05, -6.46e-05],␣
↪[-3.99e-05, -6.46e-05, -0.00012]]
>>> zs = [.5, .3, .2]
>>> dBVirial_mixture_dzs(zs=zs, Bijs=Bijs)
[-3.4089e-05, -7.2301e-05, -0.00012621]
```

chemicals.virial.**d2BVirial_mixture_dzizjs**(*zs*, *Bijs*, *d2B_dzizjs=None*)

Calculate second mole fraction derivative of the *B* second virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial^2 B}{\partial x_i \partial x_j} = B_{i,j} + B_{j,i}$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Bijs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

d2B_dzizjs [list[list[float]], optional] Array for First mole fraction derivatives of second virial coefficient in density form [m^3/mol]

Returns

d2B_dzizjs [list[list[float]]] First mole fraction derivatives of second virial coefficient in density form [m^3/mol]

Examples

```
>>> Bijs = [[-6.24e-06, -2.013e-05, -3.9e-05], [-2.01e-05, -4.391e-05, -6.46e-05],␣
↪[-3.99e-05, -6.46e-05, -0.00012]]
>>> zs = [.5, .3, .2]
>>> d2BVirial_mixture_dzizjs(zs=zs, Bijs=Bijs)
[[-1.248e-05, -4.023e-05, -7.89e-05], [-4.023e-05, -8.782e-05, -0.0001292], [-7.89e-
↪05, -0.0001292, -0.00024]]
```

chemicals.virial.**d3BVirial_mixture_dzizjzks**(*zs*, *Bijs*, *d3B_dzizjzks=None*)

Calculate third mole fraction derivative of the *B* third virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial^3 B}{\partial x_i \partial x_j \partial x_k} = 0$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Bijs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

d3B_dzizjzks [list[list[list[float]]]] Array for third mole fraction derivatives of second virial coefficient in density form [m^3/mol]

Returns

d3B_dzizjzks [list[list[list[float]]]] Third mole fraction derivatives of second virial coefficient in density form [m^3/mol]

Examples

```
>>> Bijs = [[-6.24e-06, -2.013e-05, -3.9e-05], [-2.01e-05, -4.391e-05, -6.46e-05],␣
↪[-3.99e-05, -6.46e-05, -0.00012]]
>>> zs = [.5, .3, .2]
>>> d3BVirial_mixture_dzizjzks(zs=zs, Bijs=Bijs)
[[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0], [0.0, 0.0,␣
↪0.0], [0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]]
```

chemicals.virial.**CVirial_mixture_Orentlicher_Prausnitz**(*zs*, *Cijs*)

Calculate the *C* third virial coefficient from a matrix of virial cross-coefficients. The diagonal is virial coefficients of the pure components.

$$C = \sum_i \sum_j \sum_k y_i y_j y_k C_{ijk}(T)$$

$$C_{ijk} = (C_{ij} C_{jk} C_{ik})^{1/3}$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

Returns

C [float] Third virial coefficient in density form [m^6/mol^2]

References

[1]

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> zs = [.5, .3, .2]
>>> CVirial_mixture_Orentlicher_Prausnitz(zs, Cijs)
2.0790440095e-09
```

chemicals.virial.**dCVirial_mixture_dT_Orentlicher_Prausnitz**(*zs*, *Cijs*, *dCij_dTs*)

Calculate the first temperature derivative of the C third virial coefficient from matrices of virial cross-coefficients and their first temperature derivatives.

$$\frac{\partial C}{\partial T} = \sum_i \sum_j \sum_k \frac{zizjzk \sqrt[3]{\text{Cij}\,(T)\,\text{Cik}\,(T)\,\text{Cjk}\,(T)}\left(\frac{\text{Cij}\,(T)\,\text{Cik}\,(T)\frac{d}{dT}\,\text{Cjk}\,(T)}{3} + \frac{\text{Cij}\,(T)\,\text{Cjk}\,(T)\frac{d}{dT}\,\text{Cik}\,(T)}{3} + \frac{\text{Cik}\,(T)\,\text{Cjk}\,(T)\frac{d}{dT}\,\text{Cij}\,(T}{3}\right.}{\text{Cij}\,(T)\,\text{Cik}\,(T)\,\text{Cjk}\,(T)}$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

dCij_dTs [list[list[float]]] First temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K]

Returns

dC_dT [float] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

References

[1]

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> dCij_dTs = [[-2.212e-12, -4.137e-12, -1.079e-11], [-4.137e-12, -7.669e-12, -1.
↪809e-11], [-1.079e-11, -1.809e-11, -2.010e-11]]
>>> zs = [.5, .3, .2]
>>> dCVirial_mixture_dT_Orentlicher_Prausnitz(zs, Cijs, dCij_dTs)
-7.2751517e-12
```

chemicals.virial.**d2CVirial_mixture_dT2_Orentlicher_Prausnitz**(*zs*, *Cijs*, *dCij_dTs*, *d2Cij_dT2s*)

Calculate the second temperature derivative of the C third virial coefficient from matrices of virial cross-coefficients and their first and second temperature derivatives.

$$\frac{\partial^2 C}{\partial T^2} = \sum_i \sum_j \sum_k z_i z_j z_k \frac{\sqrt[3]{\text{Cij}\,(T)\,\text{Cik}\,(T)\,\text{Cjk}\,(T)}\left(\frac{\left(\text{Cij}\,(T)\,\text{Cik}\,(T)\frac{d}{dT}\,\text{Cjk}\,(T)+\text{Cij}\,(T)\,\text{Cjk}\,(T)\frac{d}{dT}\,\text{Cik}\,(T)+\text{Cik}\,(T)\,\text{Cjk}\,(T)\frac{d}{dT}\,\text{Cij}\,(T)\right)^2}{\text{Cij}\,(T)\,\text{Cik}\,(T)\,\text{Cjk}\,(T)}\right.}{}$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

dCij_dTs [list[list[float]]] First temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K]

d2Cij_dT2s [list[list[float]]] Second temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K^2]

Returns

d2C_dT2 [float] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

References

[1]

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> dCij_dTs = [[-2.212e-12, -4.137e-12, -1.079e-11], [-4.137e-12, -7.669e-12, -1.
↪809e-11], [-1.079e-11, -1.809e-11, -2.010e-11]]
>>> d2Cij_dT2s = [[ 2.6469e-14,  5.0512e-14,  1.1509e-13], [ 5.0512e-14,  9.3272e-
↪14,  1.7836e-13], [ 1.1509e-13,  1.7836e-13, -1.4906e-13]]
>>> zs = [.5, .3, .2]
>>> d2CVirial_mixture_dT2_Orentlicher_Prausnitz(zs, Cijs, dCij_dTs, d2Cij_dT2s)
6.7237107787e-14
```

chemicals.virial.**d3CVirial_mixture_dT3_Orentlicher_Prausnitz**(*zs*, *Cijs*, *dCij_dTs*, *d2Cij_dT2s*, *d3Cij_dT3s*)

Calculate the third temperature derivative of the C third virial coefficient from matrices of virial cross-coefficients and their first, second, and third temperature derivatives.

The expression is quite lengthy and not shown here [1].

$$\frac{\partial^3 C}{\partial T^3}$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

dCij_dTs [list[list[float]]] First temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K]

d2Cij_dT2s [list[list[float]]] Second temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K^2]

d3Cij_dT3s [list[list[float]]] Third temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K^2^2]

Returns

d3C_dT3 [float] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

References

[1]

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> dCij_dTs = [[-2.212e-12, -4.137e-12, -1.079e-11], [-4.137e-12, -7.669e-12, -1.
↪809e-11], [-1.079e-11, -1.809e-11, -2.010e-11]]
>>> d2Cij_dT2s = [[ 2.6469e-14,  5.0512e-14,  1.1509e-13], [ 5.0512e-14,  9.3272e-
↪14,  1.7836e-13], [ 1.1509e-13,  1.7836e-13, -1.4906e-13]]
>>> d3Cij_dT3s = [[-4.2300e-16, -7.9727e-16, -1.6962e-15], [-7.9727e-16, -1.3826e-
↪15, -1.4525e-15], [-1.6962e-15, -1.4525e-15,  1.9786e-14]]
>>> zs = [.5, .3, .2]
>>> d3CVirial_mixture_dT3_Orentlicher_Prausnitz(zs, Cijs, dCij_dTs, d2Cij_dT2s,␣
↪d3Cij_dT3s)
-3.7358368e-16
```

chemicals.virial.**dCVirial_mixture_Orentlicher_Prausnitz_dzs**(*zs*, *Cijs*, *dCs=None*)

Calculate the first mole fraction derivatives of the C third virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial C}{\partial z_m} = \sum_{\substack{0 \le i \le nc \\ 0 \le j \le nc \\ 0 \le k \le nc}} \sqrt[3]{Cs_{i,j} Cs_{i,k} Cs_{j,k}}\left(\delta_{im} z s_j z s_k + \delta_{jm} z s_i z s_k + \delta_{km} z s_i z s_j\right)$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

dCs [list[float], optional] First derivatives of C with respect to mole fraction, [m^6/mol^2]

Returns

dC_dzs [list[float]] First derivatives of C with respect to mole fraction, [m^6/mol^2]

Notes

This equation can be derived with SymPy, as follows

```
>>> from sympy import *
>>> i, j, k, m, n, o = symbols("i, j, k, m, n, o", cls=Idx)
>>> zs = IndexedBase('zs')
>>> Cs = IndexedBase('Cs')
>>> nc = symbols('nc')
>>> C_expr = Sum(zs[i]*zs[j]*zs[k]*cbrt(Cs[i,j]*Cs[i,k]*Cs[j,k]),[i,0,nc],[j,0,nc],
↪[k,0,nc])
```

(continues on next page)

(continued from previous page)

```
>>> diff(C_expr, zs[m])
Sum((Cs[i, j]*Cs[i, k]*Cs[j, k])**(1/3)*KroneckerDelta(i, m)*zs[j]*zs[k] + (Cs[i,␣
↪j]*Cs[i, k]*Cs[j, k])**(1/3)*KroneckerDelta(j, m)*zs[i]*zs[k] + (Cs[i, j]*Cs[i,␣
↪k]*Cs[j, k])**(1/3)*KroneckerDelta(k, m)*zs[i]*zs[j], (i, 0, nc), (j, 0, nc), (k,␣
↪0, nc))
```

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> zs = [.5, .3, .2]
>>> dCVirial_mixture_Orentlicher_Prausnitz_dzs(zs, Cijs)
[5.44450470e-09, 6.54968776e-09, 7.74986672e-09]
```

chemicals.virial.**d2CVirial_mixture_Orentlicher_Prausnitz_dzizjs**(*zs*, *Cijs*, *d2Cs=None*)

Calculate the second mole fraction derivatives of the C third virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial^2 C}{\partial z_m \partial z_n} = \sum_{\substack{0 \le i \le nc \\ 0 \le j \le nc \\ 0 \le k \le nc}} \sqrt[3]{Cs_{i,j} Cs_{i,k} Cs_{j,k}}\left(\delta_{im}\delta_{jn}zs_k + \delta_{im}\delta_{kn}zs_j + \delta_{in}\delta_{jm}zs_k + \delta_{in}\delta_{km}zs_j + \delta_{jm}\delta_{kn}zs_i + \delta_{jn}\delta_{km}zs_i\right)$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

d2Cs [list[list[float]], optional] Second derivatives of C with respect to mole fraction, [m^6/mol^2]

Returns

d2Cs [list[list[float]]] Second derivatives of C with respect to mole fraction, [m^6/mol^2]

Notes

This equation can be derived with SymPy, as follows

```
>>> from sympy import *
>>> i, j, k, m, n, o = symbols("i, j, k, m, n, o", cls=Idx)
>>> zs = IndexedBase('zs')
>>> Cs = IndexedBase('Cs')
>>> nc = symbols('nc')
>>> C_expr = Sum(zs[i]*zs[j]*zs[k]*cbrt(Cs[i,j]*Cs[i,k]*Cs[j,k]),[i,0,nc],[j,0,nc],
↪[k,0,nc])
>>> diff(C_expr, zs[m], zs[n])
Sum((Cs[i, j]*Cs[i, k]*Cs[j, k])**(1/3)*(KroneckerDelta(i, m)*KroneckerDelta(j,␣
↪n)*zs[k] + KroneckerDelta(i, m)*KroneckerDelta(k, n)*zs[j] + KroneckerDelta(i,␣
↪n)*KroneckerDelta(j, m)*zs[k] + KroneckerDelta(i, n)*KroneckerDelta(k, m)*zs[j] +␣
↪KroneckerDelta(j, m)*KroneckerDelta(k, n)*zs[i] + KroneckerDelta(j,␣
↪n)*KroneckerDelta(k, m)*zs[i]), (i, 0, nc), (j, 0, nc), (k, 0, nc))
```

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> zs = [.5, .3, .2]
>>> d2CVirial_mixture_Orentlicher_Prausnitz_dzizjs(zs, Cijs)
[[9.6827886655e-09, 1.1449146725e-08, 1.3064355337e-08], [1.1449146725e-08, 1.
↪38557674294e-08, 1.60903596751e-08], [1.3064355337e-08, 1.60903596751e-08, 2.
↪0702239403e-08]]
```

chemicals.virial.**d3CVirial_mixture_Orentlicher_Prausnitz_dzizjzks**(*zs*, *Cijs*, *d3Cs=None*)

Calculate the third mole fraction derivatives of the C third virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial^3 C}{\partial z_m \partial z_n \partial z_o} = \sum_{\substack{0 \le i \le nc \\ 0 \le j \le nc \\ 0 \le k \le nc}} \sqrt[3]{Cs_{i,j} Cs_{i,k} Cs_{j,k}} \left(\delta_{im}\delta_{jn}\delta_{ko} + \delta_{im}\delta_{jo}\delta_{kn} + \delta_{in}\delta_{jm}\delta_{ko} + \delta_{in}\delta_{jo}\delta_{km} + \delta_{io}\delta_{jm}\delta_{kn} + \delta_{io}\delta_{jn}\delta_{km}\right)$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

d3Cs [list[list[list[float]]], optional] Third derivatives of C with respect to mole fraction, [m^6/mol^2]

Returns

d3Cs [list[list[list[float]]]] Third derivatives of C with respect to mole fraction, [m^6/mol^2]

Notes

This equation can be derived with SymPy, as follows

```
>>> from sympy import *
>>> i, j, k, m, n, o = symbols("i, j, k, m, n, o", cls=Idx)
>>> zs = IndexedBase('zs')
>>> Cs = IndexedBase('Cs')
>>> nc = symbols('nc')
>>> C_expr = Sum(zs[i]*zs[j]*zs[k]*cbrt(Cs[i,j]*Cs[i,k]*Cs[j,k]),[i,0,nc],[j,0,nc],
↪[k,0,nc])
>>> diff(C_expr, zs[m], zs[n], zs[o])
Sum((Cs[i, j]*Cs[i, k]*Cs[j, k])**(1/3)*(KroneckerDelta(i, m)*KroneckerDelta(j,␣
↪n)*KroneckerDelta(k, o) + KroneckerDelta(i, m)*KroneckerDelta(j,␣
↪o)*KroneckerDelta(k, n) + KroneckerDelta(i, n)*KroneckerDelta(j,␣
↪m)*KroneckerDelta(k, o) + KroneckerDelta(i, n)*KroneckerDelta(j,␣
↪o)*KroneckerDelta(k, m) + KroneckerDelta(i, o)*KroneckerDelta(j,␣
↪m)*KroneckerDelta(k, n) + KroneckerDelta(i, o)*KroneckerDelta(j,␣
↪n)*KroneckerDelta(k, m)), (i, 0, nc), (j, 0, nc), (k, 0, nc))
```

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> zs = [.5, .3, .2]
>>> d3CVirial_mixture_Orentlicher_Prausnitz_dzizjzks(zs, Cijs)
[[[8.760000000e-09, 1.0187346981e-08, 1.12329228549e-08], [1.01873469818e-08, 1.
↪21223973593e-08, 1.35937701316e-08], [1.12329228549e-08, 1.35937701316e-08, 1.
↪68488143533e-08]], [[1.01873469818e-08, 1.21223973593e-08, 1.35937701316e-08], [1.
↪2122397359e-08, 1.47600000000e-08, 1.68328437491e-08], [1.35937701316e-08, 1.
↪68328437491e-08, 2.12181074230e-08]], [[1.12329228549e-08, 1.35937701316e-08, 1.
↪68488143533e-08], [1.35937701316e-08, 1.68328437491e-08, 2.12181074230e-08], [1.
↪68488143533e-08, 2.12181074230e-08, 2.95620000000e-08]]]
```

chemicals.virial.**d2CVirial_mixture_Orentlicher_Prausnitz_dTdzs**(*zs*, *Cijs*, *dCij_dTs*, *d2C_dTdzs=None*)

Calculate the first mole fraction derivatives of the C third virial coefficient from a matrix of virial cross-coefficients.

$$\frac{\partial^2 C}{\partial T \partial z_m}$$

Parameters

zs [list[float]] Mole fractions of each species, [-]

Cijs [list[list[float]]] Third virial binary interaction coefficients in density form [m^6/mol^2]

dCij_dTs [list[list[float]]] First temperature derivative of third virial binary interaction coefficients in density form [m^6/mol^2/K]

d2C_dTdzs [list[float], optional] Array for second derivatives of C with respect to mole fraction and temperature, [m^6/mol^2/K]

Returns

d2C_dTdzs [list[float]] Second derivatives of C with respect to mole fraction and temperature, [m^6/mol^2/K]

Notes

This equation can be derived with SymPy, as follows

```
>>> from sympy import *
>>> from sympy import *
>>> i, j, k, m, n, o, T = symbols("i, j, k, m, n, o, T", cls=Idx)
>>> zs = IndexedBase('zs')
>>> Cs = IndexedBase('Cs')
>>> dC_dTs = IndexedBase('dC_dTs')
>>> nc = symbols('nc')
>>> C_expr = Sum(zs[i]*zs[j]*zs[k]/3*cbrt(Cs[i,j]*Cs[i,k]*Cs[j,k])/(Cs[i,j]*Cs[i,
↪k]*Cs[j,k])*(Cs[i,j]*Cs[i,k]*dC_dTs[j,k] + Cs[i,j]*dC_dTs[i,k]*Cs[j,k] + dC_dTs[i,
↪j]*Cs[i,k]*Cs[j,k]),[i,0,nc],[j,0,nc],[k,0,nc])
>>> diff(C_expr, zs[m])
Sum((Cs[i, j]*Cs[i, k]*Cs[j, k])**(1/3)*(Cs[i, j]*Cs[i, k]*dC_dTs[j, k] + Cs[i,␣
↪j]*Cs[j, k]*dC_dTs[i, k] + Cs[i, k]*Cs[j, k]*dC_dTs[i, j])*KroneckerDelta(i,␣
↪m)*zs[j]*zs[k]/(3*Cs[i, j]*Cs[i, k]*Cs[j, k]) + (Cs[i, j]*Cs[i, k]*Cs[j, k])**(1/
↪3)*(Cs[i, j]*Cs[i, k]*dC_dTs[j, k] + Cs[i, j]*Cs[j, k]*dC_dTs[i, k] + Cs[j,␣
↪k]*Cs[j, k]*dC_dTs[i, j])*KroneckerDelta(j, m)*zs[i]*zs[k]/(3*Cs[i, j]*Cs[i,␣
↪k]*Cs[j, k]) + (Cs[i, j]*Cs[i, k]*Cs[j, k])**(1/3)*(Cs[i, j]*Cs[i, k]*dC_dTs[j,␣
↪k] + Cs[i, j]*Cs[j, k]*dC_dTs[i, k] + Cs[i, k]*Cs[j, k]*dC_dTs[i,␣
↪j])*KroneckerDelta(k, m)*zs[i]*zs[j]/(3*Cs[i, j]*Cs[i, k]*Cs[j, k]), (i, 0, nc),␣
↪(j, 0, nc), (k, 0, nc))
```

(continues on next page)

(continued from previous page)

Examples

```
>>> Cijs = [[1.46e-09, 1.831e-09, 2.12e-09], [1.831e-09, 2.46e-09, 2.996e-09], [2.
↪12e-09, 2.996e-09, 4.927e-09]]
>>> dCij_dTs = [[-2.212e-12, -4.137e-12, -1.079e-11], [-4.137e-12, -7.669e-12, -1.
↪809e-11], [-1.079e-11, -1.809e-11, -2.010e-11]]
>>> zs = [.5, .3, .2]
>>> d2CVirial_mixture_Orentlicher_Prausnitz_dTdzs(zs, Cijs, dCij_dTs)
[-1.5740994103e-11, -2.27267309501e-11, -3.56846953115e-11]
```

chemicals.virial.**dV_dzs_virial**(*B*, *C*, *V*, *dB_dzs*, *dC_dzs*, *dV_dzs=None*)

Calculates first mole fraction derivative of volume for the virial equation of state.

$$\frac{\partial V}{\partial z_i} = \frac{V(V\frac{\partial B}{\partial z_i} + \frac{\partial C}{\partial z_i})}{2BV + 3C + V^2}$$

Parameters

B [float] Second virial coefficient in density form [m^3/mol]

C [float] Third virial coefficient in density form [m^6/mol^2]

V [float] Molar volume from virial equation, [m^3/mol]

dB_dzs [list[float]] First mole fraction derivatives of second virial coefficient in density form [m^3/mol]

dC_dzs [list[float]] First derivatives of C with respect to mole fraction, [m^6/mol^2]

dV_dzs [list[float], optional] Array for first derivatives of molar volume with respect to mole fraction, [m^3/mol]

Returns

dV_dzs [list[float]] First derivatives of molar volume with respect to mole fraction, [m^3/mol]

Notes

This expression was derived with SymPy as follows:

```
>>> from sympy import *
>>> Z, R, T, P, z1 = symbols('Z, R, T, P, z1')
>>> B, C, V = symbols('B, C, V', cls=Function)
>>> base =Eq(P*V(z1)/(R*T), 1 + B(z1)/V(z1) + C(z1)/V(z1)**2)
>>> P_sln = solve(base, P)[0]
>>> solve(diff(P_sln, z1), Derivative(V(z1), z1))
[(V(z1)*Derivative(B(z1), z1) + Derivative(C(z1), z1))*V(z1)/(2*B(z1)*V(z1) +␣
↪3*C(z1) + V(z1)**2)]
```

Examples

```
>>> dV_dzs_virial(B=-5.130920247359858e-05, C=2.6627784284381213e-09, V=0.
↪024892080086430797, dB_dzs=[-4.457911131778849e-05, -9.174964457681726e-05, -0.
↪00015942586798410 28], dC_dzs=[6.270599057032657e-09, 7.766612052069565e-09, 9.
↪503031492910165e-09])
[-4.4510120473455416e-05, -9.181495962913208e-05, -0.00015970040988493522]
```

chemicals.virial.**d2V_dzizjs_virial**(*B*, *C*, *V*, *dB_dzs*, *dC_dzs*, *dV_dzs*, *d2B_dzizjs*, *d2C_dzizjs*, *d2V_dzizjs=None*)

Calculates second mole fraction derivative of volume for the virial equation of state.

$$\frac{\partial^2 V}{\partial z_i \partial z_j}$$

Parameters

B [float] Second virial coefficient in density form [m^3/mol]

C [float] Third virial coefficient in density form [m^6/mol^2]

V [float] Molar volume from virial equation, [m^3/mol]

dB_dzs [list[float]] First mole fraction derivatives of second virial coefficient in density form [m^3/mol]

dC_dzs [list[float]] First derivatives of C with respect to mole fraction, [m^6/mol^2]

dV_dzs [list[float]] First derivatives of molar volume with respect to mole fraction, [m^3/mol]

d2B_dzizjs [list[list[float]]] Second mole fraction derivatives of second virial coefficient in density form [m^3/mol]

d2C_dzizjs [list[list[float]]] Second derivatives of C with respect to mole fraction, [m^6/mol^2]

d2V_dzizjs [list[list[float]], optional] Array for second derivatives of molar volume with respect to mole fraction, [m^3/mol]

Returns

d2V_dzizjs [list[list[float]]] Second derivatives of molar volume with respect to mole fraction, [m^3/mol]

Notes

This expression was derived with SymPy as follows:

```
>>> from sympy import *
>>> Z, R, T, P, z1 = symbols('Z, R, T, P, z1')
>>> B, C, V = symbols('B, C, V', cls=Function)
>>> base =Eq(P*V(z1)/(R*T), 1 + B(z1)/V(z1) + C(z1)/V(z1)**2)
>>> P_sln = solve(base, P)[0]
>>> solve(diff(P_sln, z1), Derivative(V(z1), z1))
[(V(z1)*Derivative(B(z1), z1) + Derivative(C(z1), z1))*V(z1)/(2*B(z1)*V(z1) +␣
↪3*C(z1) + V(z1)**2)]
```

Examples

```
>>> d2C_dzizjs = [[1.0287075724127612e-08, 1.2388277824773021e-08, 1.
↪4298813522844275e-08], [1.2388277824773021e-08, 1.514162073913238e-08, 1.
↪8282527232061114e-08], [1.4298813522844275e-08, 1.8282527232061114e-08, 2.
↪3350122217403063e-08]]
>>> d2B_dzizjs = [[-1.0639357784985337e-05, -3.966321845899801e-05, -7.
↪53987684376414e-05], [-3.966321845899801e-05, -8.286257232134107e-05, -0.
↪00014128571574782375], [-7.53987684376414e-05, -0.00014128571574782375, -0.
↪00024567752140887547]]
>>> dB_dzs = [-4.457911131778849e-05, -9.174964457681726e-05, -0.
↪0001594258679841028]
>>> dC_dzs = [6.270599057032657e-09, 7.766612052069565e-09, 9.503031492910165e-09]
>>> dV_dzs = [-4.4510120473455416e-05, -9.181495962913208e-05, -0.
↪00015970040988493522]
>>> d2V_dzizjs_virial(B=-5.130920247359858e-05, C=2.6627784284381213e-09, V=0.
↪024892080086430797, dB_dzs=dB_dzs, dC_dzs=dC_dzs, dV_dzs=dV_dzs, d2B_dzizjs=d2B_
↪dzizjs, d2C_dzizjs=d2C_dzizjs)
[[-1.04268917389e-05, -3.9654694588e-05, -7.570310078e-05], [-3.9654694588e-05, -8.
↪3270116767e-05, -0.0001423083584], [-7.5703100789e-05, -0.000142308358, -0.
↪00024779788]]
```

1.34.2 Second Virial Correlations

chemicals.virial.**BVirial_Pitzer_Curl**(*T*, *Tc*, *Pc*, *omega*, *order=0*)

Calculates the second virial coefficient using the model in [1]. Designed for simple calculations.

$$B_r = B^{(0)} + \omega B^{(1)}$$

$$B^{(0)} = 0.1445 - 0.33/T_r - 0.1385/T_r^2 - 0.0121/T_r^3$$

$$B^{(1)} = 0.073 + 0.46/T_r - 0.5/T_r^2 - 0.097/T_r^3 - 0.0073/T_r^8$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

order [int, optional] Order of the calculation. 0 for the calculation of B itself; for 1/2/3, the first/second/third derivative of B with respect to temperature; and for -1/-2, the first/second indefinite integral of B with respect to temperature. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

B [float] Second virial coefficient in density form or its integral/derivative if specified, [m^3/mol or m^3/mol/K^order]

Notes

Analytical models for derivatives and integrals are available for orders -2, -1, 1, 2, and 3, all obtained with SymPy.

For first temperature derivative of B:

$$\frac{dB^{(0)}}{dT} = \frac{33Tc}{100T^2} + \frac{277Tc^2}{1000T^3} + \frac{363Tc^3}{10000T^4}$$

$$\frac{dB^{(1)}}{dT} = -\frac{23Tc}{50T^2} + \frac{Tc^2}{T^3} + \frac{291Tc^3}{1000T^4} + \frac{73Tc^8}{1250T^9}$$

For the second temperature derivative of B:

$$\frac{d^2B^{(0)}}{dT^2} = -\frac{3Tc}{5000T^3}\left(1100 + \frac{1385Tc}{T} + \frac{242Tc^2}{T^2}\right)$$

$$\frac{d^2B^{(1)}}{dT^2} = \frac{Tc}{T^3}\left(\frac{23}{25} - \frac{3Tc}{T} - \frac{291Tc^2}{250T^2} - \frac{657Tc^7}{1250T^7}\right)$$

For the third temperature derivative of B:

$$\frac{d^3B^{(0)}}{dT^3} = \frac{3Tc}{500T^4}\left(330 + \frac{554Tc}{T} + \frac{121Tc^2}{T^2}\right)$$

$$\frac{d^3B^{(1)}}{dT^3} = \frac{3Tc}{T^4}\left(-\frac{23}{25} + \frac{4Tc}{T} + \frac{97Tc^2}{50T^2} + \frac{219Tc^7}{125T^7}\right)$$

For the first indefinite integral of B:

$$\int B^{(0)}dT = \frac{289T}{2000} - \frac{33Tc}{100}\ln(T) + \frac{1}{20000T^2}\left(2770TTc^2 + 121Tc^3\right)$$

$$\int B^{(1)}dT = \frac{73T}{1000} + \frac{23Tc}{50}\ln(T) + \frac{1}{70000T^7}\left(35000T^6Tc^2 + 3395T^5Tc^3 + 73Tc^8\right)$$

For the second indefinite integral of B:

$$\int\int B^{(0)}dTdT = \frac{289T^2}{4000} - \frac{33T}{100}Tc\ln(T) + \frac{33T}{100}Tc + \frac{277Tc^2}{2000}\ln(T) - \frac{121Tc^3}{20000T}$$

$$\int\int B^{(1)}dTdT = \frac{73T^2}{2000} + \frac{23T}{50}Tc\ln(T) - \frac{23T}{50}Tc + \frac{Tc^2}{2}\ln(T) - \frac{1}{420000T^6}\left(20370T^5Tc^3 + 73Tc^8\right)$$

References

[1]

Examples

Example matching that in BVirial_Abbott, for isobutane.

```
>>> BVirial_Pitzer_Curl(510., 425.2, 38E5, 0.193)
-0.00020845362479301725
```

chemicals.virial.**BVirial_Abbott**(*T*, *Tc*, *Pc*, *omega*, *order=0*)

Calculates the second virial coefficient using the model in [1]. Simple fit to the Lee-Kesler equation.

$$B_r = B^{(0)} + \omega B^{(1)}$$

$$B^{(0)} = 0.083 + \frac{0.422}{T_r^{1.6}}$$

$$B^{(1)} = 0.139 - \frac{0.172}{T_r^{4.2}}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

order [int, optional] Order of the calculation. 0 for the calculation of B itself; for 1/2/3, the first/second/third derivative of B with respect to temperature; and for -1/-2, the first/second indefinite integral of B with respect to temperature. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

B [float] Second virial coefficient in density form or its integral/derivative if specified, [m^3/mol or m^3/mol/K^order]

Notes

Analytical models for derivatives and integrals are available for orders -2, -1, 1, 2, and 3, all obtained with SymPy.

For first temperature derivative of B:

$$\frac{dB^{(0)}}{dT} = \frac{0.6752}{T\left(\frac{T}{Tc}\right)^{1.6}}$$

$$\frac{dB^{(1)}}{dT} = \frac{0.7224}{T\left(\frac{T}{Tc}\right)^{4.2}}$$

For the second temperature derivative of B:

$$\frac{d^2B^{(0)}}{dT^2} = -\frac{1.75552}{T^2\left(\frac{T}{Tc}\right)^{1.6}}$$

$$\frac{d^2B^{(1)}}{dT^2} = -\frac{3.75648}{T^2\left(\frac{T}{Tc}\right)^{4.2}}$$

For the third temperature derivative of B:

$$\frac{d^3B^{(0)}}{dT^3} = \frac{6.319872}{T^3\left(\frac{T}{Tc}\right)^{1.6}}$$

$$\frac{d^3B^{(1)}}{dT^3} = \frac{23.290176}{T^3\left(\frac{T}{Tc}\right)^{4.2}}$$

For the first indefinite integral of B:

$$\int B^{(0)} dT = 0.083T + \frac{\frac{211}{300}Tc}{\left(\frac{T}{Tc}\right)^{0.6}}$$

$$\int B^{(1)} dT = 0.139T + \frac{0.05375Tc}{\left(\frac{T}{Tc}\right)^{3.2}}$$

For the second indefinite integral of B:

$$\int\int B^{(0)} dT dT = 0.0415T^2 + \frac{211}{120}Tc^2\left(\frac{T}{Tc}\right)^{0.4}$$

$$\int\int B^{(1)} dT dT = 0.0695T^2 - \frac{\frac{43}{1760}Tc^2}{\left(\frac{T}{Tc}\right)^{2.2}}$$

References

[1]

Examples

Example is from [1], p. 93, and matches the result exactly, for isobutane.

```
>>> BVirial_Abbott(510., 425.2, 38E5, 0.193)
-0.0002057018500095
```

chemicals.virial.**BVirial_Tsonopoulos**(*T*, *Tc*, *Pc*, *omega*, *order=0*)

Calculates the second virial coefficient using the model in [1].

$$B_r = B^{(0)} + \omega B^{(1)}$$

$$B^{(0)} = 0.1445 - 0.330/T_r - 0.1385/T_r^2 - 0.0121/T_r^3 - 0.000607/T_r^8$$

$$B^{(1)} = 0.0637 + 0.331/T_r^2 - 0.423/T_r^3 - 0.423/T_r^3 - 0.008/T_r^8$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

order [int, optional] Order of the calculation. 0 for the calculation of B itself; for 1/2/3, the first/second/third derivative of B with respect to temperature; and for -1/-2, the first/second indefinite integral of B with respect to temperature. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

B [float] Second virial coefficient in density form or its integral/derivative if specified, [m^3/mol or m^3/mol/K^order]

Notes

A more complete expression is also available, in BVirial_Tsonopoulos_extended.

Analytical models for derivatives and integrals are available for orders -2, -1, 1, 2, and 3, all obtained with SymPy.

For first temperature derivative of B:

$$\frac{dB^{(0)}}{dT} = \frac{33Tc}{100T^2} + \frac{277Tc^2}{1000T^3} + \frac{363Tc^3}{10000T^4} + \frac{607Tc^8}{125000T^9}$$

$$\frac{dB^{(1)}}{dT} = -\frac{331Tc^2}{500T^3} + \frac{1269Tc^3}{1000T^4} + \frac{8Tc^8}{125T^9}$$

For the second temperature derivative of B:

$$\frac{d^2B^{(0)}}{dT^2} = -\frac{3Tc}{125000T^3}\left(27500 + \frac{34625Tc}{T} + \frac{6050Tc^2}{T^2} + \frac{1821Tc^7}{T^7}\right)$$

$$\frac{d^2B^{(1)}}{dT^2} = \frac{3Tc^2}{500T^4}\left(331 - \frac{846Tc}{T} - \frac{96Tc^6}{T^6}\right)$$

For the third temperature derivative of B:

$$\frac{d^3B^{(0)}}{dT^3} = \frac{3Tc}{12500T^4}\left(8250 + \frac{13850Tc}{T} + \frac{3025Tc^2}{T^2} + \frac{1821Tc^7}{T^7}\right)$$

$$\frac{d^3B^{(1)}}{dT^3} = \frac{3Tc^2}{250T^5}\left(-662 + \frac{2115Tc}{T} + \frac{480Tc^6}{T^6}\right)$$

For the first indefinite integral of B:

$$\int B^{(0)}dT = \frac{289T}{2000} - \frac{33Tc}{100}\ln(T) + \frac{1}{7000000T^7}\left(969500T^6Tc^2 + 42350T^5Tc^3 + 607Tc^8\right)$$

$$\int B^{(1)}dT = \frac{637T}{10000} - \frac{1}{70000T^7}\left(23170T^6Tc^2 - 14805T^5Tc^3 - 80Tc^8\right)$$

For the second indefinite integral of B:

$$\int\int B^{(0)}dTdT = \frac{289T^2}{4000} - \frac{33T}{100}Tc\ln(T) + \frac{33T}{100}Tc + \frac{277Tc^2}{2000}\ln(T) - \frac{1}{42000000T^6}\left(254100T^5Tc^3 + 607Tc^8\right)$$

$$\int\int B^{(1)}dTdT = \frac{637T^2}{20000} - \frac{331Tc^2}{1000}\ln(T) - \frac{1}{210000T^6}\left(44415T^5Tc^3 + 40Tc^8\right)$$

References

[1]

Examples

Example matching that in BVirial_Abbott, for isobutane.

```
>>> BVirial_Tsonopoulos(510., 425.2, 38E5, 0.193)
-0.0002093529540
```

chemicals.virial.**BVirial_Tsonopoulos_extended**(*T*, *Tc*, *Pc*, *omega*, *a=0*, *b=0*, *species_type=''*, *dipole=0*, *order=0*)

Calculates the second virial coefficient using the comprehensive model in [1]. See the notes for the calculation of *a* and *b*.

$$\frac{BP_c}{RT_c} = B^{(0)} + \omega B^{(1)} + aB^{(2)} + bB^{(3)}$$

$$B^{(0)} = 0.1445 - 0.33/T_r - 0.1385/T_r^2 - 0.0121/T_r^3$$

$$B^{(1)} = 0.0637 + 0.331/T_r^2 - 0.423/T_r^3 - 0.423/T_r^3 - 0.008/T_r^8$$

$$B^{(2)} = 1/T_r^6$$

$$B^{(3)} = -1/T_r^8$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

a [float, optional] Fit parameter, calculated based on species_type if a is not given and species_type matches on of the supported chemical classes.

b [float, optional] Fit parameter, calculated based on species_type if a is not given and species_type matches on of the supported chemical classes.

species_type [str, optional] One of .

dipole [float] dipole moment, optional, [Debye]

order [int, optional] Order of the calculation. 0 for the calculation of B itself; for 1/2/3, the first/second/third derivative of B with respect to temperature; and for -1/-2, the first/second indefinite integral of B with respect to temperature. No other integrals or derivatives are implemented, and an exception will be raised if any other order is given.

Returns

B [float] Second virial coefficient in density form or its integral/derivative if specified, [m^3/mol or m^3/mol/K^order]

Notes

Analytical models for derivatives and integrals are available for orders -2, -1, 1, 2, and 3, all obtained with SymPy.

To calculate *a* or *b*, the following rules are used:

For 'simple' or 'normal' fluids:

$$a = 0$$

$$b = 0$$

For 'ketone', 'aldehyde', 'alkyl nitrile', 'ether', 'carboxylic acid', or 'ester' types of chemicals:

$$a = -2.14 \times 10^{-4}\mu_r - 4.308 \times 10^{-21}(\mu_r)^8$$

$$b = 0$$

For 'alkyl halide', 'mercaptan', 'sulfide', or 'disulfide' types of chemicals:

$$a = -2.188 \times 10^{-4}(\mu_r)^4 - 7.831 \times 10^{-21}(\mu_r)^8$$

$$b = 0$$

For 'alkanol' types of chemicals (except methanol):

$$a = 0.0878$$

$$b = 0.00908 + 0.0006957\mu_r$$

For methanol:

$$a = 0.0878$$

$$b = 0.0525$$

For water:

$$a = -0.0109$$

$$b = 0$$

If required, the form of dipole moment used in the calculation of some types of *a* and *b* values is as follows:

$$\mu_r = 100000\frac{\mu^2(Pc/101325.0)}{Tc^2}$$

For first temperature derivative of B:

$$\frac{dB^{(0)}}{dT} = \frac{33Tc}{100T^2} + \frac{277Tc^2}{1000T^3} + \frac{363Tc^3}{10000T^4} + \frac{607Tc^8}{125000T^9}$$

$$\frac{dB^{(1)}}{dT} = -\frac{331Tc^2}{500T^3} + \frac{1269Tc^3}{1000T^4} + \frac{8Tc^8}{125T^9}$$

$$\frac{dB^{(2)}}{dT} = -\frac{6Tc^6}{T^7}$$

$$\frac{dB^{(3)}}{dT} = \frac{8Tc^8}{T^9}$$

For the second temperature derivative of B:

$$\frac{d^2B^{(0)}}{dT^2} = -\frac{3Tc}{125000T^3}\left(27500 + \frac{34625Tc}{T} + \frac{6050Tc^2}{T^2} + \frac{1821Tc^7}{T^7}\right)$$

$$\frac{d^2B^{(1)}}{dT^2} = \frac{3Tc^2}{500T^4}\left(331 - \frac{846Tc}{T} - \frac{96Tc^6}{T^6}\right)$$

$$\frac{d^2B^{(2)}}{dT^2} = \frac{42Tc^6}{T^8}$$

$$\frac{d^2B^{(3)}}{dT^2} = -\frac{72Tc^8}{T^{10}}$$

For the third temperature derivative of B:

$$\frac{d^3B^{(0)}}{dT^3} = \frac{3Tc}{12500T^4}\left(8250 + \frac{13850Tc}{T} + \frac{3025Tc^2}{T^2} + \frac{1821Tc^7}{T^7}\right)$$

$$\frac{d^3B^{(1)}}{dT^3} = \frac{3Tc^2}{250T^5}\left(-662 + \frac{2115Tc}{T} + \frac{480Tc^6}{T^6}\right)$$

$$\frac{d^3B^{(2)}}{dT^3} = -\frac{336Tc^6}{T^9}$$

$$\frac{d^3B^{(3)}}{dT^3} = \frac{720Tc^8}{T^{11}}$$

For the first indefinite integral of B:

$$\int B^{(0)}dT = \frac{289T}{2000} - \frac{33Tc}{100}\ln(T) + \frac{1}{70000000T^7}\left(969500T^6Tc^2 + 42350T^5Tc^3 + 607Tc^8\right)$$

$$\int B^{(1)}dT = \frac{637T}{10000} - \frac{1}{70000T^7}\left(23170T^6Tc^2 - 14805T^5Tc^3 - 80Tc^8\right)$$

$$\int B^{(2)}dT = -\frac{Tc^6}{5T^5}$$

$$\int B^{(3)}dT = \frac{Tc^8}{7T^7}$$

For the second indefinite integral of B:

$$\int\int B^{(0)}dTdT = \frac{289T^2}{4000} - \frac{33T}{100}Tc\ln(T) + \frac{33T}{100}Tc + \frac{277Tc^2}{2000}\ln(T) - \frac{1}{420000000T^6}\left(254100T^5Tc^3 + 607Tc^8\right)$$

$$\int\int B^{(1)}dTdT = \frac{637T^2}{20000} - \frac{331Tc^2}{1000}\ln(T) - \frac{1}{210000T^6}\left(44415T^5Tc^3 + 40Tc^8\right)$$

$$\int\int B^{(2)}dTdT = \frac{Tc^6}{20T^4}$$

$$\int\int B^{(3)}dTdT = -\frac{Tc^8}{42T^6}$$

References

[1], [2]

Examples

Example from Perry's Handbook, 8E, p2-499. Matches to a decimal place.

```
>>> BVirial_Tsonopoulos_extended(430., 405.65, 11.28E6, 0.252608, a=0, b=0, species_
↪type='ketone', dipole=1.469)
-9.679718337596e-05
```

New implementations, returning the derivatives as well

chemicals.virial.**BVirial_Pitzer_Curl_fast**(*T*, *Tc*, *Pc*, *omega*)

Implementation of *BVirial_Pitzer_Curl* in the interface which calculates virial coefficients and their derivatives at the same time.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Examples

```
>>> BVirial_Pitzer_Curl_fast(510., 425.2, 38E5, 0.193)
(-0.000208453624, 1.065377516e-06, -5.7957101e-09, 4.513533043e-11)
```

chemicals.virial.**BVirial_Abbott_fast**(*T*, *Tc*, *Pc*, *omega*)

Implementation of *BVirial_Abbott* in the interface which calculates virial coefficients and their derivatives at the same time.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Examples

```
>>> BVirial_Abbott_fast(510., 425.2, 38E5, 0.193)
(-0.0002057018500, 1.039249294e-06, -5.902233639e-09, 4.78222764e-11)
```

chemicals.virial.**BVirial_Tsonopoulos_fast**(*T*, *Tc*, *Pc*, *omega*)

Implementation of *BVirial_Tsonopoulos* in the interface which calculates virial coefficients and their derivatives at the same time.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Examples

```
>>> BVirial_Tsonopoulos_fast(510., 425.2, 38E5, 0.193)
(-0.0002093529540, 9.95742355e-07, -5.54234465e-09, 4.57035160e-11)
```

chemicals.virial.**BVirial_Tsonopoulos_extended_fast**(*T*, *Tc*, *Pc*, *omega*, *a=0.0*, *b=0.0*)

Implementation of *BVirial_Tsonopoulos_extended* in the interface which calculates virial coefficients and their derivatives at the same time.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

a [float, optional] Fit parameter [-]

b [float, optional] Fit parameter [-]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Examples

```
>>> BVirial_Tsonopoulos_extended_fast(510., 425.2, 38E5, 0.193)
(-0.0002093529540, 9.9574235e-07, -5.54234465e-09, 4.5703516e-11)
```

chemicals.virial.**BVirial_Oconnell_Prausnitz**(*T*, *Tc*, *Pc*, *omega*)

Calculates the second virial coefficient using the model in [1].

$$B_r = B^{(0)} + \omega B^{(1)}$$

$$B^{(0)} = c0 + \frac{c1}{T_r} + \frac{c2}{T_r^2} + \frac{c3}{T_r^3}$$

$$B^{(1)} = d0 + \frac{d1}{T_r^2} + \frac{d2}{T_r^3} + \frac{d3}{T_r^8}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Notes

The coefficients are as follows:

c0 = 0.1445 c1 = -0.330 c2 = -0.1385 c3 = -0.0121

d0 = 0.073 d1 = 0.46 d2 = -0.50 d3 = -0.097 d4 = -0.0073

References

[1]

Examples

```
>>> BVirial_Oconnell_Prausnitz(510., 425.2, 38E5, 0.193)
(-0.000203193781, 1.036185972e-06, -6.53679132e-09, 6.59478287e-11)
```

chemicals.virial.**BVirial_Xiang**(*T*, *Tc*, *Pc*, *Vc*, *omega*)

Calculates the second virial coefficient using the model in [1].

$$B = \frac{\left(-b_0 T_r^{-3/4} \exp(b_1 T_r^{-3}) + b_2 T_r^{-1/2})\right)}{V}_c$$

$$b_0 = b_{00} + b_{01}\omega + b_{02}\theta$$

$$b_1 = b_{10} + b_{11}\omega + b_{12}\theta$$

$$b_2 = b_{20} + b_{21}\omega + b_{22}\theta$$

$$\theta = (Z_c - 0.29)^2$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

Vc [float] Critical volume of the fluid [m^3/mol]

omega [float] Acentric factor for fluid, [-]

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

References

[1]

Examples

```
>>> BVirial_Xiang(388.26, 647.1, 22050000.0, 5.543076e-05, 0.344)
(-0.0004799570, 4.6778266e-06, -7.0157656e-08, 1.4137862e-09)
```

chemicals.virial.**BVirial_Meng**(*T*, *Tc*, *Pc*, *Vc*, *omega*, *a=0.0*)

Calculates the second virial coefficient using the model in [1].

$$B = \frac{RT_c}{P_c}\left(f_0 + \omega f_1 + f_2\right)$$

$$f_0 = c_0 + c_1/T_r + c_2/T_r^2 + c_3/T_r^3 + c_4/T_r^8$$

$$f_1 = d_0 + d_1/T_r + d_2/T_r^2 + d_3/T_r^3 + d_4/T_r^8$$

$$f_2 = \frac{a}{T_r^6}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

Vc [float] Critical volume of the fluid [m^3/mol]

omega [float] Acentric factor for fluid, [-]

a [float] Polar parameter that can be estimated by *chemicals.virial.Meng_virial_a*

Returns

B [float] Second virial coefficient in density form [m^3/mol]

dB_dT [float] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2 [float] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3 [float] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

References

[1]

Examples

```
>>> BVirial_Meng(388.26, 647.1, 22050000.0, 5.543076e-05, 0.344)
(-0.00032436028, 2.47004e-06, -3.132e-08, 5.8e-10)
```

chemicals.virial.**Meng_virial_a**(*Tc*, *Pc*, *dipole=0.0*, *haloalkane=False*)

Calculate the *a* parameter which is used in the Meng *B* second virial coefficient for polar components. There are two correlations implemented - one for haloalkanes, and another for other polar molecules. If the dipole moment is not provided, a value of 0.0 will be returned.

If the compound is a haloalkane

$$a = -1.1524 \times 10^{-6}\mu_r^2 + 7.2238 \times 10^{-11}\mu_r^4 - 1.8701 \times 10^{-15}\mu_r^6$$

Otherwise

$$a = -3.0309 \times 10^{-6}\mu_r^2 + 9.503 \times 10^{-11}\mu_r^4 - 1.2469 \times 10^{-15}\mu_r^6$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

dipole [float] Dipole moment, [debye]

haloalkane [bool] Whether or not the compound is a haloalkane, [-]

Returns

a [float] Coefficient [-]

References

[1]

Examples

Ethanol

```
>>> Meng_virial_a(514.0, 6137000.0, 1.44, haloalkane=False)
-0.00637841
```

R-41 Fluoromethane

```
>>> Meng_virial_a(317.4, 5870000.0, 1.85, haloalkane=True)
-0.04493829
```

1.34.3 Third Virial Correlations

chemicals.virial.**CVirial_Orbey_Vera**(*T*, *Tc*, *Pc*, *omega*)

Calculates the third virial coefficient using the model in [1].

$$C = (RT_c/P_c)^2(fC_{Tr}^{(0)} + \omega fC_{Tr}^{(1)})$$

$$fC_{Tr}^{(0)} = 0.01407 + 0.02432T_r^{-2.8} - 0.00313T_r^{-10.5}$$

$$fC_{Tr}^{(1)} = -0.02676 + 0.01770T_r^{-2.8} + 0.040T_r^{-3} - 0.003T_r^{-6} - 0.00228T_r^{-10.5}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

C [float] Third virial coefficient in density form [m^6/mol^2]

dC_dT [float] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2 [float] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3 [float] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

References

[1]

Examples

n-octane

```
>>> CVirial_Orbey_Vera(T=300, Tc=568.7, Pc=2490000.0, omega=0.394)
(-1.1107124e-05, 4.1326808e-07, -1.6041435e-08, 6.7035158e-10)
```

chemicals.virial.**CVirial_Liu_Xiang**(*T*, *Tc*, *Pc*, *Vc*, *omega*)

Calculates the third virial coefficient using the model in [1].

$$C = V_c^2(f_{T_r}^{(0)} + \omega f_{T_r}^{(1)} + \theta f_{T_r}^{(2)})$$

$$f_{T_r}^{(0)} = a_{00} + a_{10}T_r^{-3} + a_{20}T_r^{-6} + a_{30}T_r^{-11}$$

$$f_{T_r}^{(1)} = a_{01} + a_{11}T_r^{-3} + a_{21}T_r^{-6} + a_{31}T_r^{-11}$$

$$f_{T_r}^{(2)} = a_{02} + a_{12}T_r^{-3} + a_{22}T_r^{-6} + a_{32}T_r^{-11}$$

$$\theta = (Z_c - 0.29)^2$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

Vc [float] Critical volume of the fluid [m^3/mol]

omega [float] Acentric factor for fluid, [-]

Returns

C [float] Third virial coefficient in density form [m^6/mol^2]

dC_dT [float] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2 [float] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3 [float] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

References

[1]

Examples

Water at Tr = 0.6

```
>>> CVirial_Liu_Xiang(388.26, 647.1, 22050000.0, 5.543076923076923e-05, 0.344)
(-1.4779977e-07, 4.9949901e-09, -1.652899e-10, 5.720067e-12)
```

1.34.4 Cross-Parameters

chemicals.virial.**Tarakad_Danner_virial_CSP_kijs**(*Vcs*)

Calculates a binary interaction parameter for the calculation of Bij binary virial coefficient as shown in [1] and [2].

This equation for kij is:

$$k_{ij} = 1 - \frac{8\sqrt{v_{ci}v_{cj}}}{(V_{ci}^{1/3} + V_{ci}^{1/3})^3}$$

The equation this kij is used in is

$$T_{cij} = \sqrt{T_{ci}T_{cj}}(1 - k_{ij})$$

Parameters

Vcs [list[float]] Critical volumes for each species, [m^3/mol]

Returns

kijs [list[list[float]]] Binary interaction parameters, [-]

References

[1], [2]

Examples

```
>>> Tarakad_Danner_virial_CSP_kijs(Vcs=[0.000168, 0.000316])
[[0.0, 0.01646332091], [0.0164633209, 0.0]]
```

chemicals.virial.**Tarakad_Danner_virial_CSP_Tcijs**(*Tcs*, *kijs*)

Calculates the corresponding states critical temperature for the calculation of Bij binary virial coefficient as shown in [1] and [2].

$$T_{cij} = \sqrt{T_{ci}T_{cj}}(1 - k_{ij})$$

Parameters

Tcs [list[float]] Critical temperatures for each species, [K]

kijs [list[list[float]]] Binary interaction parameters, [-]

Returns

Tcijs [list[list[float]]] CSP Critical temperatures for each pair of species, [K]

References

[1], [2]

Examples

```
>>> kijs = Tarakad_Danner_virial_CSP_kijs(Vcs=[0.000168, 0.000316])
>>> Tarakad_Danner_virial_CSP_Tcijs(Tcs=[514.0, 591.75], kijs=kijs)
[[514.0, 542.42694], [542.42694, 591.75000]]
```

chemicals.virial.**Tarakad_Danner_virial_CSP_Pcijs**(*Tcs*, *Pcs*, *Vcs*, *Tcijs*)

Calculates the corresponding states critical pressure for the calculation of Bij binary virial coefficient as shown in [1] and [2].

$$P_{cij} = \frac{4T_{cij}\left(\frac{P_{ci}V_{ci}}{T_{ci}} + \frac{P_{cj}V_{cj}}{T_{cj}}\right)}{(V_{ci}^{1/3} + V_{ci}^{1/3})^3}$$

Parameters

Tcs [list[float]] Critical temperatures for each species, [K]

Pcs [list[float]] Critical pressures for each species, [Pa]

Vcs [list[float]] Critical volumes for each species, [m^3/mol]

Tcijs [list[list[float]]] CSP Critical temperatures for each pair of species, [K]

Returns

Pcijs [list[list[float]]] CSP Critical pressures for each pair of species, [Pa]

References

[1], [2]

Examples

```
>>> kijs = Tarakad_Danner_virial_CSP_kijs(Vcs=[0.000168, 0.000316])
>>> Tcijs = Tarakad_Danner_virial_CSP_Tcijs(Tcs=[514.0, 591.75], kijs=kijs)
>>> Tarakad_Danner_virial_CSP_Pcijs(Tcs=[514.0, 591.75], Pcs=[6137000.0, 4108000.0],
↪ Vcs=[0.000168, 0.000316], Tcijs=Tcijs)
[[6136999.9, 4861936.4], [4861936.4, 4107999.9]]
```

chemicals.virial.**Tarakad_Danner_virial_CSP_omegaijs**(*omegas*)

Calculates the corresponding states acentric factor for the calculation of Bij binary virial coefficient as shown in [1] and [2].

$$\omega_{ij} = 0.5(\omega_i + \omega_j)$$

Parameters

omegas [list[float]] Acentric factor for each species, [-]

Returns

omegaijs [list[list[float]]] CSP acentric factors for each pair of species, [-]

References

[1], [2]

Examples

```
>>> Tarakad_Danner_virial_CSP_omegaijs([0.635, 0.257])
[[0.635, 0.446], [0.446, 0.257]]
```

chemicals.virial.**Lee_Kesler_virial_CSP_Vcijs**(*Vcs*)

Calculates the corresponding states critical volumes for the calculation of Vcijs binary virial coefficient as shown in [1] and [2].

$$V_{cij} = \frac{1}{8}\left(V_{c,i}^{1/3} + V_{c,j}^{1/3}\right)^3$$

Parameters

Vcs [list[float]] Critical volume of the fluids [m^3/mol]

Returns

Vcijs [list[list[float]]] CSP critical volumes for each pair of species, [m^3/mol]

Notes

[1] cites this as Lee-Kesler rules.

References

[1], [2]

Examples

```
>>> Lee_Kesler_virial_CSP_Vcijs(Vcs=[0.000168, 0.000316])
[[0.000168, 0.00023426], [0.000234265, 0.000316]]
```

chemicals.virial.**Meng_Duan_2005_virial_CSP_kijs**(*CASs*, *atomss*)

Calculates a binary interaction parameter for the calculation of Bij binary virial coefficient as shown in [1]. This implements a correlation of alkane-alkane, CO2-alkane, and N2-alkane.

The equation this kij is used in is

$$T_{cij} = \sqrt{T_{ci}T_{cj}}(1 - k_{ij})$$

Parameters

CASs [list[str]] CAS registration numbers for each component, [-]

atomss [list[dict]] Breakdown of each component into its elements and their counts, as a dict, [-]

Returns

kijs [list[list[float]]] Binary interaction parameters, [-]

References

[1]

Examples

```
>>> CASs = ['74-82-8', '74-84-0', '124-38-9', '7727-37-9', '7439-89-6']
>>> atomss = [{'C': 1, 'H': 4}, {'C': 2, 'H': 6}, {'C': 1, 'O': 2}, {'N': 2}, {'Fe':
↪ 1}]
>>> kijs = Meng_Duan_2005_virial_CSP_kijs(CASs=CASs, atomss=atomss)
```

1.34.5 Second Virial Correlations Dense Implementations

chemicals.virial.**BVirial_Xiang_vec**(*T*, *Tcs*, *Pcs*, *Vcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the Xiang B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

Vcs [list[float]] Critical volume of the fluids [m^3/mol]

omegas [list[float]] Acentric factor for fluids, [-]

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Xiang_mat**(*T*, *Tcs*, *Pcs*, *Vcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Xiang B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

Vcs [list[list[float]]] Critical volume of the fluids [m^3/mol]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Pitzer_Curl_vec**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the Pitzer-Curl B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

omegas [list[float]] Acentric factor for fluids, [-]

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Pitzer_Curl_mat**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Pitzer-Curl B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Abbott_vec**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the Abbott B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

omegas [list[float]] Acentric factor for fluids, [-]

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Abbott_mat**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Abbott B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Tsonopoulos_vec**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the Tsonopoulos B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

omegas [list[float]] Acentric factor for fluids, [-]

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Tsonopoulos_mat**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Tsonopoulos B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Meng_vec**(*T*, *Tcs*, *Pcs*, *Vcs*, *omegas*, *ais*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the Meng B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

Vcs [list[float]] Critical volume of the fluids [m^3/mol]

omegas [list[float]] Acentric factor for fluids, [-]

ais [list[float]] Polar parameters that can be estimated by *chemicals.virial.Meng_virial_a*

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Meng_mat**(*T*, *Tcs*, *Pcs*, *Vcs*, *omegas*, *ais*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Meng B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

Vcs [list[list[float]]] Critical volume of the fluids [m^3/mol]

omegas [list[list[float]]] Acentric factor for fluids, [-]

ais [list[float]] Polar parameters that can be estimated as the average of the pure component values predicted by *chemicals.virial.Meng_virial_a*

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Oconnell_Prausnitz_vec**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the O'connell Prausnitz B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

omegas [list[float]] Acentric factor for fluids, [-]

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Oconnell_Prausnitz_mat**(*T*, *Tcs*, *Pcs*, *omegas*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Oconnell_Prausnitz B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Tsonopoulos_extended_vec**(*T*, *Tcs*, *Pcs*, *omegas*, *ais*, *bs*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a vectorized calculation of the Tsonopoulos (extended) B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

omegas [list[float]] Acentric factor for fluids, [-]

ais [list[float]] Fit parameters, [-]

bs [list[float]] Fit parameters, [-]

Bs [list[float], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[float]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[float]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[float]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[float]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

chemicals.virial.**BVirial_Tsonopoulos_extended_mat**(*T*, *Tcs*, *Pcs*, *omegas*, *ais*, *bs*, *Bs=None*, *dB_dTs=None*, *d2B_dT2s=None*, *d3B_dT3s=None*)

Perform a matrix calculation of the Tsonopoulos (extended) B virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

omegas [list[list[float]]] Acentric factor for fluids, [-]

ais [list[list[float]]] Fit parameters, [-]

bs [list[list[float]]] Fit parameters, [-]

Bs [list[list[float]], optional] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]], optional] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]], optional] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]], optional] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

Returns

Bs [list[list[float]]] Second virial coefficient in density form [m^3/mol]

dB_dTs [list[list[float]]] First temperature derivative of second virial coefficient in density form [m^3/mol/K]

d2B_dT2s [list[list[float]]] Second temperature derivative of second virial coefficient in density form [m^3/mol/K^2]

d3B_dT3s [list[list[float]]] Third temperature derivative of second virial coefficient in density form [m^3/mol/K^3]

1.34.6 Third Virial Correlations Dense Implementations

chemicals.virial.**CVirial_Liu_Xiang_vec**(*T*, *Tcs*, *Pcs*, *Vcs*, *omegas*, *Cs=None*, *dC_dTs=None*, *d2C_dT2s=None*, *d3C_dT3s=None*)

Perform a vectorized calculation of the Xiang C virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

Vcs [list[float]] Critical volume of the fluids [m^3/mol]

omegas [list[float]] Acentric factor for fluids, [-]

Cs [list[float], optional] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[float], optional] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[float], optional] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[float], optional] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

Returns

Cs [list[float]] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[float]] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[float]] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[float]] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

chemicals.virial.**CVirial_Orbey_Vera_vec**(*T*, *Tcs*, *Pcs*, *omegas*, *Cs=None*, *dC_dTs=None*, *d2C_dT2s=None*, *d3C_dT3s=None*)

Perform a vectorized calculation of the Orbey-Vera C virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[float]] Critical temperature of fluids [K]

Pcs [list[float]] Critical pressure of the fluids [Pa]

omegas [list[float]] Acentric factor for fluids, [-]

Cs [list[float], optional] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[float], optional] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[float], optional] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[float], optional] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

Returns

Cs [list[float]] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[float]] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[float]] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[float]] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

chemicals.virial.**CVirial_Liu_Xiang_mat**(*T*, *Tcs*, *Pcs*, *Vcs*, *omegas*, *Cs=None*, *dC_dTs=None*, *d2C_dT2s=None*, *d3C_dT3s=None*)

Perform a matrix calculation of the Xiang C virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

Vcs [list[list[float]]] Critical volume of the fluids [m^3/mol]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Cs [list[list[float]], optional] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[list[float]], optional] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[list[float]], optional] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[list[float]], optional] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

Returns

Cs [list[list[float]]] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[list[float]]] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[list[float]]] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[list[float]]] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

chemicals.virial.**CVirial_Orbey_Vera_mat**(*T*, *Tcs*, *Pcs*, *omegas*, *Cs=None*, *dC_dTs=None*, *d2C_dT2s=None*, *d3C_dT3s=None*)

Perform a matrix calculation of the Orbey-Vera C virial coefficient model and its first three temperature derivatives.

Parameters

T [float] Temperature of fluid [K]

Tcs [list[list[float]]] Critical temperature of fluids [K]

Pcs [list[list[float]]] Critical pressure of the fluids [Pa]

omegas [list[list[float]]] Acentric factor for fluids, [-]

Cs [list[list[float]], optional] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[list[float]], optional] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[list[float]], optional] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[list[float]], optional] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

Returns

Cs [list[list[float]]] Third virial coefficient in density form [m^6/mol^2]

dC_dTs [list[list[float]]] First temperature derivative of third virial coefficient in density form [m^6/mol^2/K]

d2C_dT2s [list[list[float]]] Second temperature derivative of third virial coefficient in density form [m^6/mol^2/K^2]

d3C_dT3s [list[list[float]]] Third temperature derivative of third virial coefficient in density form [m^6/mol^2/K^3]

1.35 Viscosity (chemicals.viscosity)

This module contains various viscosity estimation routines, dataframes of fit coefficients, and mixing rules.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.35.1 Pure Low Pressure Liquid Correlations

chemicals.viscosity.**Letsou_Stiel**(*T*, *MW*, *Tc*, *Pc*, *omega*)

Calculates the viscosity of a liquid using an emperical model developed in [1]. However. the fitting parameters for tabulated values in the original article are found in ChemSep.

$$\xi = \frac{2173.424 T_c^{1/6}}{\sqrt{MW}\, P_c^{2/3}}$$

$$\xi^{(0)} = (1.5174 - 2.135 T_r + 0.75 T_r^2) \cdot 10^{-5}$$

$$\xi^{(1)} = (4.2552 - 7.674 T_r + 3.4 T_r^2) \cdot 10^{-5}$$

$$\mu = (\xi^{(0)} + \omega \xi^{(1)})/\xi$$

Parameters

T [float] Temperature of fluid [K]

MW [float] Molwcular weight of fluid [g/mol]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor of compound

Returns

mu_l [float] Viscosity of liquid, [Pa*s]

Notes

The form of this equation is a polynomial fit to tabulated data. The fitting was performed by the DIPPR. This is DIPPR Procedure 8G: Method for the viscosity of pure, nonhydrocarbon liquids at high temperatures internal units are SI standard. [1]'s units were different. DIPPR test value for ethanol is used.

Average error 34%. Range of applicability is 0.76 < Tr < 0.98.

References

[1]

Examples

```
>>> Letsou_Stiel(400., 46.07, 516.25, 6.383E6, 0.6371)
0.00020361508753 08
```

chemicals.viscosity.**Przedziecki_Sridhar**(*T*, *Tm*, *Tc*, *Pc*, *Vc*, *Vm*, *omega*, *MW*)

Calculates the viscosity of a liquid using an emperical formula developed in [1].

$$\mu = \frac{V_o}{E(V - V_o)}$$

$$E = -1.12 + \frac{V_c}{12.94 + 0.10MW - 0.23P_c + 0.0424T_m - 11.58(T_m/T_c)}$$

$$V_o = 0.0085\omega T_c - 2.02 + \frac{V_m}{0.342(T_m/T_c) + 0.894}$$

Parameters

T [float] Temperature of the fluid [K]

Tm [float] Melting point of fluid [K]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

Vc [float] Critical volume of the fluid [m^3/mol]

Vm [float] Molar volume of the fluid at temperature [K]

omega [float] Acentric factor of compound

MW [float] Molwcular weight of fluid [g/mol]

Returns

mu_l [float] Viscosity of liquid, [Pa*s]

Notes

A test by Reid (1983) is used, but only mostly correct. This function is not recommended. Internal units are bar and mL/mol.

References

[1]

Examples

```
>>> Przedziecki_Sridhar(383., 178., 591.8, 41E5, 316E-6, 95E-6, .263, 92.14)
0.00021981479956033846
```

1.35.2 Pure High Pressure Liquid Correlations

chemicals.viscosity.**Lucas**(*T*, *P*, *Tc*, *Pc*, *omega*, *Psat*, *mu_l*)

Adjustes for pressure the viscosity of a liquid using an emperical formula developed in [1], but as discussed in [2] as the original source is in German.

$$\frac{\mu}{\mu_{sat}} = \frac{1 + D(\Delta P_r/2.118)^A}{1 + C\omega\Delta P_r}$$

$$\Delta P_r = \frac{P - P^{sat}}{P_c}$$

$$A = 0.9991 - \frac{4.674 \times 10^{-4}}{1.0523 T_r^{-0.03877} - 1.0513}$$

$$D = \frac{0.3257}{(1.0039 - T_r^{2.573})^{0.2906}} - 0.2086$$

$$C = -0.07921 + 2.1616 T_r - 13.4040 T_r^2 + 44.1706 T_r^3 - 84.8291 T_r^4 + 96.1209 T_r^5 - 59.8127 T_r^6 + 15.6719 T_r^7$$

Parameters

T [float] Temperature of fluid [K]

P [float] Pressure of fluid [Pa]

Tc: float Critical point of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

omega [float] Acentric factor of compound

Psat [float] Saturation pressure of the fluid [Pa]

mu_l [float] Viscosity of liquid at 1 atm or saturation, [Pa*s]

Returns

mu_l_dense [float] Viscosity of liquid, [Pa*s]

Notes

This equation is entirely dimensionless; all dimensions cancel. The example is from Reid (1987); all results agree. Above several thousand bar, this equation does not represent true behavior. If Psat is larger than P, the fluid may not be liquid; dPr is set to 0.

References

[1], [2]

Examples

```
>>> Lucas(300., 500E5, 572.2, 34.7E5, 0.236, 0, 0.00068) # methylcyclohexane
0.0010683738499316494
```

1.35.3 Liquid Mixing Rules

No specific correlations are implemented but *chemicals.utils.mixing_logarithmic* with weight fractions is the recommended form.

1.35.4 Pure Low Pressure Gas Correlations

chemicals.viscosity.**Yoon_Thodos**(*T*, *Tc*, *Pc*, *MW*)

Calculates the viscosity of a gas using an emperical formula developed in [1].

$$\eta\xi \times 10^8 = 46.10T_r^{0.618} - 20.40\exp(-0.449T_r) + 19.40\exp(-4.058T_r) + 1$$

$$\xi = 2173.424T_c^{1/6}MW^{-1/2}P_c^{-2/3}$$

Parameters

T [float] Temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

MW [float] Molwcular weight of fluid [g/mol]

Returns

mu_g [float] Viscosity of gas, [Pa*s]

Notes

This equation has been tested. The equation uses SI units only internally. The constant 2173.424 is an adjustment factor for units. Average deviation within 3% for most compounds. Greatest accuracy with dipole moments close to 0. Hydrogen and helium have different coefficients, not implemented. This is DIPPR Procedure 8B: Method for the Viscosity of Pure, non hydrocarbon, nonpolar gases at low pressures

References

[1]

Examples

```
>>> Yoon_Thodos(300., 556.35, 4.5596E6, 153.8)
1.019488572777e-05
```

chemicals.viscosity.**Stiel_Thodos**(*T*, *Tc*, *Pc*, *MW*)

Calculates the viscosity of a gas using an emperical formula developed in [1].

if $T_r > 1.5$:

$$\mu_g = 17.78 \times 10^{-5}(4.58T_r - 1.67)^{0.625}/\xi$$

else:

$$\mu_g = 34 \times 10^{-5}T_r^{0.94}/\xi$$

$$\xi = \frac{T_c^{(1/6)}}{\sqrt{MW}P_c^{2/3}}$$

Parameters

T [float] Temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

MW [float] Molwcular weight of fluid [g/mol]

Returns

mu_g [float] Viscosity of gas, [Pa*s]

Notes

Claimed applicability from 0.2 to 5 atm. Developed with data from 52 nonpolar, and 53 polar gases. internal units are poise and atm. Seems to give reasonable results.

References

[1]

Examples

```
>>> Stiel_Thodos(300., 556.35, 4.5596E6, 153.8) #CCl4
1.040892622360e-05
```

chemicals.viscosity.**Lucas_gas**(*T*, *Tc*, *Pc*, *Zc*, *MW*, *dipole=0.0*, *CASRN=None*)

Estimate the viscosity of a gas using an emperical formula developed in several sources, but as discussed in [1] as the original sources are in German or merely personal communications with the authors of [1].

$$\eta = \left[0.807 T_r^{0.618} - 0.357\exp(-0.449 T_r) + 0.340\exp(-4.058 T_r) + 0.018\right] F_p^\circ F_Q^\circ / \xi$$

$$F_p^\circ = 1, 0 \le \mu_r < 0.022$$

$$F_p^\circ = 1 + 30.55(0.292 - Z_c)^{1.72}, 0.022 \le \mu_r < 0.075$$

$$F_p^\circ = 1 + 30.55(0.292 - Z_c)^{1.72} |0.96 + 0.1(T_r - 0.7)| 0.075 < \mu_r$$

$$F_Q^\circ = 1.22 Q^{0.15}\left\{1 + 0.00385[(T_r - 12)^2]^{1/M}\,\text{sign}(T_r - 12)\right\}$$

$$\mu_r = 52.46\frac{\mu^2 P_c}{T_c^2}$$

$$\xi = 0.176\left(\frac{T_c}{MW^3 P_c^4}\right)^{1/6}$$

Parameters

T [float] Temperature of fluid [K]

Tc: float Critical point of fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

Zc [float] Critical compressibility of the fluid [Pa]

dipole [float] Dipole moment of fluid [debye]

CASRN [str, optional] CAS of the fluid

Returns

mu_g [float] Viscosity of gas, [Pa*s]

Notes

The example is from [1]; all results agree. Viscosity is calculated in micropoise, and converted to SI internally (1E-7). Q for He = 1.38; Q for H2 = 0.76; Q for D2 = 0.52.

References

[1]

Examples

```
>>> Lucas_gas(T=550., Tc=512.6, Pc=80.9E5, Zc=0.224, MW=32.042, dipole=1.7)
1.7822676912698925e-05
```

chemicals.viscosity.**viscosity_gas_Gharagheizi**(*T*, *Tc*, *Pc*, *MW*)

Calculates the viscosity of a gas using an emperical formula developed in [1].

$$\mu = 10^{-7}|10^{-5} P_c T_r + \left(0.091 - \frac{0.477}{M}\right)T + M\left(10^{-5}P_c - \frac{8M^2}{T^2}\right)\left(\frac{10.7639}{T_c} - \frac{4.1929}{T}\right)|$$

Parameters

T [float] Temperature of the fluid [K]

Tc [float] Critical temperature of the fluid [K]

Pc [float] Critical pressure of the fluid [Pa]

MW [float] Molwcular weight of fluid [g/mol]

Returns

mu_g [float] Viscosity of gas, [Pa*s]

Notes

Example is first point in supporting information of article, for methane. This is the prefered function for gas viscosity. 7% average relative deviation. Deviation should never be above 30%. Developed with the DIPPR database. It is believed theoretically predicted values are included in the correlation.

Under 0.2Tc, this correlation has been modified to provide values at the limit.

References

[1]

Examples

```
>>> viscosity_gas_Gharagheizi(120., 190.564, 45.99E5, 16.04246)
5.215761625399613e-06
```

1.35.5 Pure High Pressure Gas Correlations

No correlations are implemented yet.

1.35.6 Gas Mixing Rules

chemicals.viscosity.**Herning_Zipperer**(*zs*, *mus*, *MWs*, *MW_roots=None*)

Calculates viscosity of a gas mixture according to mixing rules in [1].

$$\mu = \frac{\sum x_i \mu_i \sqrt{MW_i}}{\sum x_i \sqrt{MW_i}}$$

Parameters

zs [float] Mole fractions of components, [-]

mus [float] Gas viscosities of all components, [Pa*s]

MWs [float] Molecular weights of all components, [g/mol]

MW_roots [float, optional] Square roots of molecular weights of all components, [g^0.5/mol^0.5]

Returns

mug [float] Viscosity of gas mixture, [Pa*s]

Notes

This equation is entirely dimensionless; all dimensions cancel. The original source has not been reviewed.

Adding the square roots can speed up the calculation.

References

[1]

Examples

```
>>> Herning_Zipperer([0.5, 0.25, 0.25], [1.78e-05, 1.12e-05, 9.35e-06], [28.0134,␣
↪16.043, 30.07])
1.4174908599465168e-05
```

chemicals.viscosity.**Brokaw**(*T*, *ys*, *mus*, *MWs*, *molecular_diameters*, *Stockmayers*)

Calculates viscosity of a gas mixture according to mixing rules in [1].

$$\eta_{mix} = \sum_{i=1}^{n} \frac{y_i \eta_i}{\sum_{j=1}^{n} y_j \phi_{ij}}$$

$$\phi_{ij} = \left(\frac{\eta_i}{\eta_j}\right)^{0.5} S_{ij} A_{ij}$$

$$A_{ij} = m_{ij} M_{ij}^{-0.5} \left[1 + \frac{M_{ij} - M_{ij}^{0.45}}{2(1+M_{ij}) + \frac{(1+M_{ij}^{0.45}) m_{ij}^{-0.5}}{1+m_{ij}}}\right]$$

$$m_{ij} = \left[\frac{4}{(1+M_{ij}^{-1})(1+M_{ij})}\right]^{0.25}$$

$$M_{ij} = \frac{M_i}{M_j}$$

$$S_{ij} = \frac{1 + (T_i^* T_j^*)^{0.5} + (\delta_i \delta_j/4)}{[1 + T_i^* + (\delta_i^2/4)]^{0.5} [1 + T_j^* + (\delta_j^2/4)]^{0.5}}$$

$$T^* = kT/\epsilon$$

Parameters

T [float] Temperature of fluid, [K]

ys [float] Mole fractions of gas components, [-]

mus [float] Gas viscosities of all components, [Pa*s]

MWs [float] Molecular weights of all components, [g/mol]

molecular_diameters [float] L-J molecular diameter of all components, [angstroms]

Stockmayers [float] L-J Stockmayer energy parameters of all components, []

Returns

mug [float] Viscosity of gas mixture, [Pa*s]

Notes

This equation is entirely dimensionless; all dimensions cancel. The original source has not been reviewed.

This is DIPPR Procedure 8D: Method for the Viscosity of Nonhydrocarbon Vapor Mixtures at Low Pressure (Polar and Nonpolar)

References

[1], [2], [3]

Examples

```
>>> Brokaw(308.2, [0.05, 0.95], [1.34E-5, 9.5029E-6], [64.06, 46.07], [0.42, 0.19],␣
↪[347, 432])
9.699085099801568e-06
```

chemicals.viscosity.**Wilke**(*ys*, *mus*, *MWs*)

Calculates viscosity of a gas mixture according to mixing rules in [1].

$$\eta_{mix} = \sum_{i=1}^{n} \frac{y_i \eta_i}{\sum_{j=1}^{n} y_j \phi_{ij}}$$

$$\phi_{ij} = \frac{(1 + \sqrt{\eta_i/\eta_j}(MW_j/MW_i)^{0.25})^2}{\sqrt{8(1 + MW_i/MW_j)}}$$

Parameters

ys [float] Mole fractions of gas components, [-]

mus [float] Gas viscosities of all components, [Pa*s]

MWs [float] Molecular weights of all components, [g/mol]

Returns

mug [float] Viscosity of gas mixture, [Pa*s]

See also:

Wilke_prefactors

Wilke_prefactored

Wilke_large

Notes

This equation is entirely dimensionless; all dimensions cancel. The original source has not been reviewed or found.

References

[1]

Examples

```
>>> Wilke([0.05, 0.95], [1.34E-5, 9.5029E-6], [64.06, 46.07])
9.701614885866193e-06
```

chemicals.viscosity.**Wilke_prefactors**(*MWs*)

The *Wilke* gas viscosity method can be sped up by precomputing several matrices. The memory used is proportional to N^2, so it can be significant, but is still a substantial performance increase even when they are so large they cannot fit into cached memory. These matrices are functions of molecular weights only. These are used by the *Wilke_prefactored* function.

$$t0_{i,j} = \frac{\sqrt{\frac{MW_j}{MW_i}}}{\sqrt{\frac{8MW_i}{MW_j}+8}}$$

$$t1_{i,j} = \frac{2\sqrt[4]{\frac{MW_j}{MW_i}}}{\sqrt{\frac{8MW_i}{MW_j}+8}}$$

$$t2_{i,j} = \frac{1}{\sqrt{\frac{8MW_i}{MW_j}+8}}$$

Parameters

MWs [list[float]] Molecular weights of all components, [g/mol]

Returns

t0s [list[list[float]]] First terms, [-]

t1s [list[list[float]]] Second terms, [-]

t2s [list[list[float]]] Third terms, [-]

Notes

These terms are derived as follows using SymPy. The viscosity terms are not known before hand so they are not included in the factors, but otherwise these parameters simplify the computation of the ϕ_{ij} term to the following:

$$\phi_{ij} = \frac{\mu_i}{\mu_j} t0_{i,j} + \sqrt{\frac{\mu_i}{\mu_j}} t1_{i,j} + t2_{i,j}$$

```
>>> from sympy import *
>>> MWi, MWj, mui, muj = symbols('MW_i, MW_j, mu_i, mu_j')
>>> f = (1 + sqrt(mui/muj)*(MWj/MWi)**Rational(1,4))**2
>>> denom = sqrt(8*(1+MWi/MWj))
>>> (expand(simplify(expand(f))/denom))
mu_i*sqrt(MW_j/MW_i)/(mu_j*sqrt(8*MW_i/MW_j + 8)) + 2*(MW_j/MW_i)**(1/4)*sqrt(mu_i/
↪mu_j)/sqrt(8*MW_i/MW_j + 8) + 1/sqrt(8*MW_i/MW_j + 8)
```

Examples

```
>>> Wilke_prefactors([64.06, 46.07])
([[0.25, 0.19392193320396522], [0.3179655106303118, 0.25]], [[0.5, 0.
↪421161930934918], [0.5856226024677849, 0.5]], [[0.25, 0.22867110638055677], [0.
↪2696470380083788, 0.25]])
>>> Wilke_prefactored([0.05, 0.95], [1.34E-5, 9.5029E-6], *Wilke_prefactors([64.06,␣
↪46.07]))
9.701614885866193e-06
```

chemicals.viscosity.**Wilke_prefactored**(*ys*, *mus*, *t0s*, *t1s*, *t2s*)

Calculates viscosity of a gas mixture according to mixing rules in [1], using precomputed parameters.

$$\eta_{mix} = \sum_{i=1}^{n} \frac{y_i \eta_i}{\sum_{j=1}^{n} y_j \phi_{ij}}$$

$$\phi_{ij} = \frac{\mu_i}{\mu_j} t0_{i,j} + \sqrt{\frac{\mu_i}{\mu_j}} t1_{i,j} + t2_{i,j}$$

Parameters

ys [float] Mole fractions of gas components, [-]

mus [float] Gas viscosities of all components, [Pa*s]

t0s [list[list[float]]] First terms, [-]

t1s [list[list[float]]] Second terms, [-]

t2s [list[list[float]]] Third terms, [-]

Returns

mug [float] Viscosity of gas mixture, [Pa*s]

See also:

Wilke_prefactors

Wilke

Wilke_large

Notes

This equation is entirely dimensionless; all dimensions cancel.

References

[1]

Examples

```
>>> Wilke_prefactored([0.05, 0.95], [1.34E-5, 9.5029E-6], *Wilke_prefactors([64.06,␣
↪46.07]))
9.701614885866193e-06
```

chemicals.viscosity.**Wilke_large**(*ys*, *mus*, *MWs*)

Calculates viscosity of a gas mixture according to mixing rules in [1].

This function is a slightly faster version of *Wilke*. It achieves its extra speed by avoiding some checks, some powers, and by allocating less memory during the computation. For very large component vectors, this function should be called instead.

Parameters

ys [float] Mole fractions of gas components, [-]

mus [float] Gas viscosities of all components, [Pa*s]

MWs [float] Molecular weights of all components, [g/mol]

Returns

mug [float] Viscosity of gas mixture, [Pa*s]

See also:

Wilke_prefactors

Wilke_prefactored

Wilke

References

[1]

Examples

```
>>> Wilke_large([0.05, 0.95], [1.34E-5, 9.5029E-6], [64.06, 46.07])
9.701614885866193e-06
```

1.35.7 Correlations for Specific Substances

chemicals.viscosity.**mu_IAPWS**(*T*, *rho*, *drho_dP=None*, *drho_dP_Tr=None*)

Calculates and returns the viscosity of water according to the IAPWS (2008) release.

Viscosity is calculated as a function of three terms; the first is the dilute-gas limit; the second is the contribution due to finite density; and the third and most complex is a critical enhancement term.

$$\mu = \mu_0 \cdot \mu_1(T, \rho) \cdot \mu_2(T, \rho)$$

$$\mu_0(T) = \frac{100\sqrt{T}}{\sum_{i=0}^{3} \frac{H_i}{T^i}}$$

$$\mu_1(T,\rho) = \exp\left[\rho\sum_{i=0}^{5}\left(\left(\frac{1}{T}-1\right)^i\sum_{j=0}^{6}H_{ij}(\rho-1)^j\right)\right]$$

if $\xi < 0.3817016416$ nm:

$$Y = 0.2q_c\xi(q_D\xi)^5\left(1 - q_c\xi + (q_c\xi)^2 - \frac{765}{504}(q_D\xi)^2\right)$$

else:

$$Y = \frac{1}{12}\sin(3\psi_D) - \frac{1}{4q_c\xi}\sin(2\psi_D) + \frac{1}{(q_c\xi)^2}\left[1 - 1.25(q_c\xi)^2\right]\sin(\psi_D) - \frac{1}{(q_c\xi)^3}\left\{\left[1 - 1.5(q_c\xi)^2\right]\psi_D - \left|(q_c\xi)^2 - 1\right|^{1.5}L(w)\right.$$

$$w = \left|\frac{q_c\xi - 1}{q_c\xi + 1}\right|^{0.5}\tan\left(\frac{\psi_D}{2}\right)$$

$$L(w) = \ln\frac{1+w}{1-w} \text{ if } q_c\xi > 1$$

$$L(w) = 2\arctan|w| \text{ if } q_c\xi \le 1$$

$$\psi_D = \arccos\left[\left(1 + q_D^2\xi^2\right)^{-0.5}\right]$$

$$\Delta\bar{\chi}(\bar{T},\bar{\rho}) = \bar{\rho}\left[\zeta(\bar{T},\bar{\rho}) - \zeta(\bar{T}_R,\bar{\rho})\frac{\bar{T}_R}{\bar{T}}\right]$$

$$\xi = \xi_0\left(\frac{\Delta\bar{\chi}}{\Gamma_0}\right)^{\nu/\gamma}$$

$$\zeta = \left(\frac{\partial\bar{\rho}}{\partial\bar{p}}\right)_{\bar{T}}$$

Parameters

T [float] Temperature of water [K]

rho [float] Density of water [kg/m^3]

drho_dP [float, optional] Partial derivative of density with respect to pressure at constant temperature (at the temperature and density of water), [kg/m^3/Pa]

drho_dP_Tr [float, optional] Partial derivative of density with respect to pressure at constant temperature (at the reference temperature (970.644 K) and the actual density of water), [kg/m^3/Pa]

Returns

mu [float] Viscosity, [Pa*s]

Notes

There are three ways to use this formulation.

1) Compute the Industrial formulation value which does not include the critical enhacement, by leaving *drho_dP* and *drho_dP_Tr* None.

2) Compute the Scientific formulation value by accurately computing and providing *drho_dP* and *drho_dP_Tr*, both with IAPWS-95.

3) Get a non-standard but 8 decimal place matching result by providing *drho_dP* computed with either IAPWS-95 or IAPWS-97, but not providing *drho_dP_Tr*; which is calculated internally. There is a formulation for that term in the thermal conductivity IAPWS equation which is used.

xmu = 0.068

qc = (1.9E-9)**-1

qd = (1.1E-9)**-1

nu = 0.630

gamma = 1.239

xi0 = 0.13E-9

Gamma0 = 0.06

TRC = 1.5

This forulation is highly optimized, spending most of its time in the logarithm, power, and square root.

References

[1]

Examples

```
>>> mu_IAPWS(298.15, 998.)
0.000889735100149808
```

```
>>> mu_IAPWS(1173.15, 400.)
6.415460784836147e-05
```

Point 4 of formulation, compared with MPEI and IAPWS, matches.

```
>>> mu_IAPWS(T=647.35, rho=322., drho_dP=1.213641949033E-2)
4.2961578738287e-05
```

Full scientific calculation:

```
>>> from chemicals.iapws import iapws95_properties, iapws95_P, iapws95_Tc
>>> T, P = 298.15, 1e5
>>> rho, _, _, _, _, _, _, _, _, _, drho_dP = iapws95_properties(T, P)
>>> P_ref = iapws95_P(1.5*iapws95_Tc, rho)
>>> _, _, _, _, _, _, _, _, _, _, drho_dP_Tr = iapws95_properties(1.5*iapws95_Tc, P_
↪ref)
>>> mu_IAPWS(T, rho, drho_dP, drho_dP_Tr)
0.00089002267377
```

chemicals.viscosity.**mu_air_lemmon**(*T*, *rho*)

Calculates and returns the viscosity of air according to Lemmon and Jacobsen (2003) [1].

Viscosity is calculated as a function of two terms; the first is the dilute-gas limit; the second is the contribution due to finite density.

$$\mu = \mu^0(T) + \mu^r(T, \rho)$$

$$\mu^0(T) = \frac{0.9266958\sqrt{MT}}{\sigma^2 \Omega(T^*)}$$

$$\Omega(T^*) = \exp\left(\sum_{i=0}^{4} b_i [\ln(T^*)]^i\right)$$

$$\mu^r = \sum_{i=1}^{n} N_i \tau^{t_i} \delta^{d_i} \exp\left(-\gamma_i \delta^{l_i}\right)$$

Parameters

T [float] Temperature of air [K]

rho [float] Molar density of air [mol/m^3]

Returns

mu [float] Viscosity of air, [Pa*s]

Notes

The coefficients are:

Ni = [10.72, 1.122, 0.002019, -8.876, -0.02916]

ti = [0.2, 0.05, 2.4, 0.6, 3.6]

di = [1, 4, 9, 1, 8]

gammai = Ii = [0, 0, 0, 1, 1]

bi = [.431, -0.4623, 0.08406, 0.005341, -0.00331]

The reducing parameters are $T_c = 132.6312$ K and $\rho_c = 10447.7$ mol/m^3. Additional parameters used are $\sigma = 0.36$ nm, $M = 28.9586$ g/mol and $\frac{e}{k} = 103.3$ K.

This is an implementation optimized for speed, spending its time in the calclulation of 1 log; 2 exp; 1 power; and 2 divisions.

References

[1]

Examples

Viscosity at 300 K and 1 bar:

```
>>> mu_air_lemmon(300.0, 40.10292351061862)
1.85371518556e-05
```

Calculate the density in-place:

```
>>> from chemicals.air import lemmon2000_rho
>>> mu_air_lemmon(300.0, lemmon2000_rho(300.0, 1e5))
1.85371518556e-05
```

1.35.8 Petroleum Correlations

chemicals.viscosity.**Twu_1985**(*T*, *Tb*, *rho*)

Calculate the viscosity of a petroleum liquid using the Twu (1985) correlation developed in [1]. Based on a fit to n-alkanes that used as a reference. Requires the boiling point and density of the system.

Parameters

T [float] Temperature of fluid [K]

Tb [float] Normal boiling point, [K]

rho [float] Liquid density liquid as measured at 60 deg F, [kg/m^3]

Returns

mu [float] Liquid viscosity, [Pa*s]

Notes

The formulas are as follows:

$$T_c^\circ = T_b \left(0.533272 + 0.191017 \times 10^{-3} T_b + 0.779681 \times 10^{-7} T_b^2 - 0.284376 \times 10^{-10} T_b^3 + 0.959468 \times 10^{28}/T_b^{13}\right)^{-1}$$

$$\alpha = 1 - T_b/T_c^\circ$$

$$\ln\left(\nu_2^\circ + 1.5\right) = 4.73227 - 27.0975\alpha + 49.4491\alpha^2 - 50.4706\alpha^4$$

$$\ln\left(\nu_1^\circ\right) = 0.801621 + 1.37179 \ln\left(\nu_2^\circ\right)$$

$$SG^\circ = 0.843593 - 0.128624\alpha - 3.36159\alpha^3 - 13749.5\alpha^{12}$$

$$\Delta SG = SG - SG^\circ$$

$$|x| = \left|1.99873 - 56.7394/\sqrt{T_b}\right|$$

$$f_1 = 1.33932|x|\Delta SG - 21.1141\Delta SG^2/\sqrt{T_b}$$

$$f_2 = |x|\Delta SG - 21.1141\Delta SG^2/\sqrt{T_b}$$

$$\ln\left(\nu_1 + \frac{450}{T_b}\right) = \ln\left(\nu_1^\circ + \frac{450}{T_b}\right)\left(\frac{1+2f_1}{1-2f_1}\right)^2$$

$$\ln\left(\nu_2 + \frac{450}{T_b}\right) = \ln\left(\nu_2^\circ + \frac{450}{T_b}\right)\left(\frac{1+2f_2}{1-2f_2}\right)^2$$

$$Z = \nu + 0.7 + \exp\left(-1.47 - 1.84\nu - 0.51\nu^2\right)$$

$$B = \frac{\ln\ln Z_1 - \ln\ln Z_2}{\ln T_1 - \ln T_2}$$

$$\ln\ln Z = \ln\ln Z_1 + B(\ln T - \ln T_1)$$

$$\nu = (Z - 0.7) - \exp\left(-0.7487 - 3.295Z - 0.7) + 0.6119Z - 0.7)^2 - 0.3193Z - 0.7)^3\right)$$

References

[1]

Examples

Sample point from article:

```
>>> Twu_1985(T=338.7055, Tb=672.3166, rho=895.5189)
0.008235009644854494
```

chemicals.viscosity.**Lorentz_Bray_Clarke**(*T*, *P*, *Vm*, *zs*, *MWs*, *Tcs*, *Pcs*, *Vcs*)

Calculates the viscosity of a gas or a liquid using the method of Lorentz, Bray, and Clarke [1]. This method is not quite the same as the original, but rather the form commonly presented and used today. The original had a different formula for pressure correction for gases which was tabular and not presented entirely in [1]. However using that distinction introduces a discontinuity between the liquid and gas viscosity, so it is not normally used.

$$\mu[\text{centipoise}] = \mu_{\text{P low, Stiel-hThodos}}[\text{centipoise}] + \frac{\text{poly}^4 - 0.0001}{\xi}$$

$$\text{poly} = (0.1023 + 0.023364\rho_r + 0.058533\rho_r^2 - 0.040758\rho_r^3 + 0.0093724\rho_r^4)$$

$$\xi = T_c^{1/6} MW^{-1/2} (P_c[\text{atm}])^{-2/3}$$

Parameters

T [float] Temperature of the fluid [K]

P [float] Pressure of the fluid [Pa]

Vm [float] Molar volume of the fluid at the actual conditions, [m^3/mol]

zs [list[float]] Mole fractions of chemicals in the fluid, [-]

MWs [list[float]] Molwcular weights of chemicals in the fluid [g/mol]

Tcs [float] Critical temperatures of chemicals in the fluid [K]

Pcs [float] Critical pressures of chemicals in the fluid [Pa]

Vcs [float] Critical molar volumes of chemicals in the fluid; these are often used as tuning parameters, fit to match a pure component experimental viscosity value [m^3/mol]

Returns

mu [float] Viscosity of phase at actual conditions , [Pa*s]

Notes

An example from [2] was implemented and checked for validation. Somewhat different rounding is used in [2].

The mixing of the pure component Stiel-Thodos viscosities happens with the Herning-Zipperer mixing rule:

$$\mu = \frac{\sum x_i \mu_i \sqrt{MW_i}}{\sum x_i \sqrt{MW_i}}$$

References

[1], [2]

Examples

```
>>> Lorentz_Bray_Clarke(T=300.0, P=1e6, Vm=0.0023025, zs=[.4, .3, .3],
... MWs=[16.04246, 30.06904, 44.09562], Tcs=[190.564, 305.32, 369.83],
... Pcs=[4599000.0, 4872000.0, 4248000.0], Vcs=[9.86e-05, 0.0001455, 0.0002])
9.925488160761484e-06
```

1.35.9 Fit Correlations

chemicals.viscosity.**PPDS9**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculate the viscosity of a liquid using the 5-term exponential power fit developed by the PPDS and named PPDS equation 9.

$$\mu = E \exp\left[A\left(\frac{C-T}{T-D}\right)^{1/3} + B\left(\frac{C-T}{T-D}\right)^{4/3}\right]$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [-]

C [float] Coefficient, [K]

D [float] Coefficient, [K]

E [float] Coefficient, [Pa*s]

Returns

mu [float] Liquid viscosity, [Pa*s]

Notes

No other source for these coefficients has been found.

There can be a singularity in this equation when T approaches C or D; it may be helpful to take as a limit to this equation D + 5 K.

References

[1]

Examples

```
>>> PPDS9(400.0, 1.74793, 1.33728, 482.347, 41.78, 9.963e-05)
0.00035091137378230684
```

chemicals.viscosity.**dPPDS9_dT**(*T*, *A*, *B*, *C*, *D*, *E*)

Calculate the temperature derivative of viscosity of a liquid using the 5-term exponential power fit developed by the PPDS and named PPDS equation 9.

Normally, the temperature derivative is:

$$\frac{\partial \mu}{\partial T} = E\left(\frac{A\sqrt[3]{\frac{C-T}{-D+T}}\,(-D+T)\left(-\frac{C-T}{3(-D+T)^2}-\frac{1}{3(-D+T)}\right)}{C-T}-\frac{B\sqrt[3]{\frac{C-T}{-D+T}}\,(C-T)}{(-D+T)^2}+B\sqrt[3]{\frac{C-T}{-D+T}}\left(-\frac{C-T}{3\,(-D+T)^2}-\frac{3(\right.\right.$$

For the low-temperature region:

$$\frac{\partial \mu}{\partial T} = E\left(-\frac{A\sqrt[3]{\frac{-C+T}{-D+T}}\,(-D+T)\left(-\frac{-C+T}{3(-D+T)^2}+\frac{1}{3(-D+T)}\right)}{-C+T}+\frac{B\sqrt[3]{\frac{-C+T}{-D+T}}\,(C-T)}{(-D+T)^2}+\frac{B\sqrt[3]{\frac{-C+T}{-D+T}}}{-D+T}-\frac{B\sqrt[3]{\frac{-C+T}{-D+T}}\,(C-T)}{-}\right.$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [-]

C [float] Coefficient, [K]

D [float] Coefficient, [K]

E [float] Coefficient, [Pa*s]

Returns

dmu_dT [float] First temperature derivative of liquid viscosity, [Pa*s]

mu [float] Liquid viscosity, [Pa*s]

References

[1]

Examples

```
>>> dPPDS9_dT(400.0, 1.74793, 1.33728, 482.347, 41.78, 9.963e-05)
(-3.186540635882627e-06, 0.00035091137378230684)
```

chemicals.viscosity.**PPDS5**(*T*, *Tc*, *a0*, *a1*, *a2*)

Calculate the viscosity of a low-pressure gas using the 3-term exponential power fit developed by the PPDS and named PPDS equation 5.

$$\mu = \frac{a_0 T_r}{\left(1 + a_1 T_r^{a_2}(T_r - 1)\right)^{1/6}}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a0 [float] Coefficient, [-]

a1 [float] Coefficient, [-]

a2 [float] Coefficient, [-]

Returns

mu [float] Low pressure gas viscosity, [Pa*s]

References

[1]

Examples

Sample coefficients for n-pentane in [1], at 350 K:

```
>>> PPDS5(T=350.0, Tc=470.008, a0=1.08003e-5, a1=0.19583, a2=0.811897)
8.096643275836e-06
```

chemicals.viscosity.**Viswanath_Natarajan_2**(*T*, *A*, *B*)

Calculate the viscosity of a liquid using the 2-term form representation developed in [1]. Requires input coefficients. The *A* coefficient is assumed to yield coefficients in Pa*s; if it yields values in 1E-3 Pa*s, remove log(100) for A.

$$\mu = \exp\left(A + \frac{B}{T}\right)$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [K]

Returns

mu [float] Liquid viscosity, [Pa*s]

Notes

No other source for these coefficients than [1] has been found.

References

[1]

Examples

DDBST has 0.0004580 as a value at this temperature for 1-Butanol.

```
>>> Viswanath_Natarajan_2(348.15, -5.9719-log(100), 1007.0)
0.0004598368695682 95
```

chemicals.viscosity.**Viswanath_Natarajan_2_exponential**(*T*, *C*, *D*)

Calculate the viscosity of a liquid using the 2-term exponential form representation developed in [1]. Requires input coefficients. The *A* coefficient is assumed to yield coefficients in Pa*s, as all coefficients found so far have been.

$$\mu = CT^D$$

Parameters

T [float] Temperature of fluid [K]

C [float] Linear coefficient, [Pa*s]

D [float] Exponential coefficient, [-]

Returns

mu [float] Liquid viscosity, [Pa*s]

Notes

No other source for these coefficients has been found.

References

[1]

Examples

```
>>> Ts = [283.15, 288.15, 303.15, 349.65]
>>> mus = [2.2173, 2.1530, 1.741, 1.0091] # in cP
>>> Viswanath_Natarajan_2_exponential(288.15, 4900800, -3.8075)
0.002114798866203873
```

Calculation of the AARD of the fit (1% is the value stated in [1].:

```
>>> mu_calc = [Viswanath_Natarajan_2_exponential(T, 4900800, -3.8075) for T in Ts]
>>> np.mean([abs((mu - mu_i*1000)/mu) for mu, mu_i in zip(mus, mu_calc)])
0.010467928813061298
```

chemicals.viscosity.**Viswanath_Natarajan_3**(*T*, *A*, *B*, *C*)

Calculate the viscosity of a liquid using the 3-term Antoine form representation developed in [1]. Requires input coefficients. If the coefficients do not yield viscosity in Pa*s, but rather cP, remove log10(1000) from *A*.

$$\log_{10} \mu = A + B/(T + C)$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [K]

C [float] Coefficient, [K]

Returns

mu [float] Liquid viscosity, [Pa*s]

Notes

No other source for these coefficients has been found.

References

[1]

Examples

```
>>> from math import log10
>>> Viswanath_Natarajan_3(298.15, -2.7173-log10(1000), -1071.18, -129.51)
0.0006129806445142113
```

chemicals.viscosity.**mu_Yaws**(*T*, *A*, *B*, *C=0.0*, *D=0.0*)

Calculate the viscosity of a liquid using the 4-term Yaws polynomial form. Requires input coefficients. If the coefficients do not yield viscosity in Pa*s, but rather cP, remove log10(1000) from *A*; this is required for the coefficients in [1].

$$\log_{10} \mu = A + B/T + CT + DT^2$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [K]

C [float] Coefficient, [1/K]

D [float] Coefficient, [1/K^2]

Returns

mu [float] Liquid viscosity, [Pa*s]

References

[1]

Examples

```
>>> from math import log10
>>> mu_Yaws(300.0, -6.4406-log10(1000), 1117.6, 0.0137, -0.000015465)
0.00100666612081
```

chemicals.viscosity.**dmu_Yaws_dT**(*T*, *A*, *B*, *C=0.0*, *D=0.0*)

Calculate the temperature derivative of the viscosity of a liquid using the 4-term Yaws polynomial form. Requires input coefficients.

$$\frac{\partial \mu}{\partial T} = 10^{A+\frac{B}{T}+T(C+DT)}\left(-\frac{B}{T^2}+C+2DT\right)\log\left(10\right)$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [K]

C [float] Coefficient, [1/K]

D [float] Coefficient, [1/K^2]

Returns

dmu_dT [float] First temperature derivative of liquid viscosity, [Pa*s/K]

Examples

```
>>> dmu_Yaws_dT(300.0, -9.4406, 1117.6, 0.0137, -0.000015465)
-1.853591586963e-05
```

chemicals.viscosity.**mu_Yaws_fitting_jacobian**(*Ts*, *A*, *B*, *C*, *D*)

Compute and return the Jacobian of the property predicted by the Yaws viscosity equation with respect to all the coefficients. This is used in fitting parameters for chemicals.

Parameters

Ts [list[float]] Temperatures of the experimental data points, [K]

A [float] Coefficient, [-]

B [float] Coefficient, [K]

C [float] Coefficient, [1/K]

D [float] Coefficient, [1/K^2]

Returns

jac [list[list[float, 4], len(Ts)]] Matrix of derivatives of the equation with respect to the fitting parameters, [various]

chemicals.viscosity.**mu_TDE**(*T*, *A*, *B*, *C*, *D*)
Calculate the viscosity of a liquid using the 4-term exponential inverse-temperature fit equation used in NIST's TDE.

$$\mu = \exp\left[A + \frac{B}{T} + \frac{C}{T^2} + \frac{D}{T^3}\right]$$

Parameters

T [float] Temperature of fluid [K]

A [float] Coefficient, [-]

B [float] Coefficient, [K]

C [float] Coefficient, [K^2]

D [float] Coefficient, [K^3]

Returns

mu [float] Liquid viscosity, [Pa*s]

References

[1]

Examples

Coefficients for isooctane at 400 K, as shown in [1].

```
>>> mu_TDE(400.0, -14.0878, 3500.26, -678132.0, 6.17706e7)
0.00001822175281438
```

1.35.10 Conversion functions

chemicals.viscosity.**viscosity_converter**(*val*, *old_scale*, *new_scale*, *extrapolate=False*)
Converts kinematic viscosity values from different scales which have historically been used. Though they may not be in use much, some standards still specify values in these scales.

Parameters

val [float] Viscosity value in the specified scale; [m^2/s] if 'kinematic viscosity'; [degrees] if Engler or Barbey; [s] for the other scales.

old_scale [str] String representing the scale that *val* is in originally.

new_scale [str] String representing the scale that *val* should be converted to.

extrapolate [bool] If True, a conversion will be performed even if outside the limits of either scale; if False, and either value is outside a limit, an exception will be raised.

Returns

result [float] Viscosity value in the specified scale; [m^2/s] if 'kinematic viscosity'; [degrees] if Engler or Barbey; [s] for the other scales

Notes

The valid scales for this function are any of the following:

['a&w b', 'a&w crucible', 'american can', 'astm 0.07', 'astm 0.10', 'astm 0.15', 'astm 0.20', 'astm 0.25', 'barbey', 'caspers tin plate', 'continental can', 'crown cork and seal', 'demmier #1', 'demmier #10', 'engler', 'ford cup #3', 'ford cup #4', 'kinematic viscosity', 'mac michael', 'murphy varnish', 'parlin cup #10', 'parlin cup #15', 'parlin cup #20', 'parlin cup #25', 'parlin cup #30', 'parlin cup #7', 'pratt lambert a', 'pratt lambert b', 'pratt lambert c', 'pratt lambert d', 'pratt lambert e', 'pratt lambert f', 'pratt lambert g', 'pratt lambert h', 'pratt lambert i', 'redwood admiralty', 'redwood standard', 'saybolt furol', 'saybolt universal', 'scott', 'stormer 100g load', 'westinghouse', 'zahn cup #1', 'zahn cup #2', 'zahn cup #3', 'zahn cup #4', 'zahn cup #5']

Some of those scales are converted linearly; the rest use tabulated data and splines.

Because the conversion is performed by spline functions, a re-conversion of a value will not yield exactly the original value. However, it is quite close.

The method 'Saybolt universal' has a special formula implemented for its conversion, from [4]. It is designed for maximum backwards compatibility with prior experimental data. It is solved by newton's method when kinematic viscosity is desired as an output.

$$SUS_{eq} = 4.6324\nu_t + \frac{[1.0 + 0.03264\nu_t]}{[(3930.2 + 262.7\nu_t + 23.97\nu_t^2 + 1.646\nu_t^3) \times 10^{-5})]}$$

References

[1], [2], [3], [4], [5]

Examples

```
>>> viscosity_converter(8.79, 'engler', 'parlin cup #7')
52.5
>>> viscosity_converter(700, 'Saybolt Universal Seconds', 'kinematic viscosity')
0.00015108914751515542
```

chemicals.viscosity.**viscosity_index**(*nu_40*, *nu_100*, *rounding=False*)

Calculates the viscosity index of a liquid. Requires dynamic viscosity of a liquid at 40°C and 100°C. Value may either be returned with or without rounding. Rounding is performed per the standard.

if nu_100 < 70:

$$L, H = \text{interp}(nu_100)$$

else:

$$L = 0.8353\nu_{100}^2 + 14.67\nu_{100} - 216$$

$$H = 0.1684\nu_{100}^2 + 11.85\nu_{100} - 97$$

if nu_40 > H:

$$VI = \frac{L - nu_{40}}{L - H} \cdot 100$$

else:

$$N = \frac{\ln(H) - \ln(\nu_{40})}{\ln(\nu_{100})}$$

$$VI = \frac{10^N - 1}{0.00715} + 100$$

Parameters

nu_40 [float] Dynamic viscosity of fluid at 40°C, [m^2/s]

nu_100 [float] Dynamic viscosity of fluid at 100°C, [m^2/s]

rounding [bool, optional] Whether to round the value or not.

Returns

VI: float Viscosity index [-]

Notes

VI is undefined for nu_100 under 2 mm^2/s. None is returned if this is the case. Internal units are mm^2/s. Higher values of viscosity index suggest a lesser decrease in kinematic viscosity as temperature increases.

Note that viscosity is a pressure-dependent property, and that the viscosity index is defined for a fluid at whatever pressure it is at. The viscosity index is thus also a function of pressure.

References

[1]

Examples

```
>>> viscosity_index(73.3E-6, 8.86E-6, rounding=True)
92
```

1.35.11 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.viscosity.**mu_data_Dutt_Prasad**
Coefficient sfor *chemicals.viscosity.Viswanath_Natarajan_3* from [1] for 100 fluids.

chemicals.viscosity.**mu_data_VN3**
Coefficients for *chemicals.viscosity.Viswanath_Natarajan_3* from [1] with data for 432 fluids.

chemicals.viscosity.**mu_data_VN2**
Coefficients for *chemicals.viscosity.Viswanath_Natarajan_2* from [1] with data for 135 fluids.

chemicals.viscosity.**mu_data_VN2E**
Coefficients for *chemicals.viscosity.Viswanath_Natarajan_2_exponential* from [1] with data for 14 fluids.

chemicals.viscosity.**mu_data_Perrys_8E_2_313**
A collection of 337 coefficient sets for *chemicals.dippr.EQ101* from the DIPPR database published openly in [3].

chemicals.viscosity.**mu_data_Perrys_8E_2_312**
A collection of 345 coefficient sets for *chemicals.dippr.EQ102* from the DIPPR database published openly in [3].

chemicals.viscosity.**mu_data_VDI_PPDS_7**
: Coefficients for the model equation *PPDS9*, published openly in [2]. Provides no temperature limits, but has been designed for extrapolation. Extrapolated to low temperatures it provides a smooth exponential increase. However, for some chemicals such as glycerol, extrapolated to higher temperatures viscosity is predicted to increase above a certain point.

chemicals.viscosity.**mu_data_VDI_PPDS_8**
: Coefficients for a tempereture polynomial (T in Kelvin) developed by the PPDS, published openly in [2]. $\mu = A + BT + CT^2 + DT^3 + ET^4$.

The structure of each dataframe is shown below:

```
In [1]: import chemicals

In [2]: chemicals.viscosity.mu_data_Dutt_Prasad
Out[2]:
                          Chemical       A        B       C   Tmin   Tmax
CAS
56-23-5      Carbon tetrachloride  -1.4708  -324.45   71.19  273.0  373.0
60-29-7                Ethyl ether  -4.4735 -3623.26 -648.55  273.0  373.0
62-53-3                    Aniline  -1.1835  -224.31  170.82  268.0  393.0
64-17-5              Ethyl alcohol  -2.8857 -1032.53  -55.95  248.0  348.0
64-18-6                Formic acid  -1.4150  -297.43  114.74  281.0  373.0
...                            ...     ...      ...     ...    ...    ...
629-59-4              Tetra decane  -1.4424  -350.81  100.18  283.0  373.0
629-62-9              Penta decane  -1.4073  -348.84  105.48  293.0  373.0
629-78-7              Hepta decane  -1.7847  -577.32   51.72  303.0  553.0
693-02-7                1 - Hexyne  -3.0941 -1404.92 -233.99  293.0  333.0
3744-21-6  2,2 - Dimethyl propane  -0.9128   -30.15  202.98  258.0  283.0

[100 rows x 6 columns]

In [3]: chemicals.viscosity.mu_data_VN3
Out[3]:
                                                      Name  ...   Tmax
CAS                                                         ...
57-10-3                                     Palmitic acid   ...  370.0
57-50-1                                           Sucrose   ...  330.0
60-12-8                                  Phenethyl alcohol  ...  380.0
60-35-5                                          Acetamide  ...  500.0
62-53-3                                            Aniline  ...  460.0
...                                                    ...  ...    ...
66538-96-3   1,2,3,4 - Tetrahydro -6 - butyl -hexyl naphtha...  ...  380.0
87077-20-1                             2-Methyl - 7 -heptanol  ...  380.0
99332-99-7                                Hexyl thiohexanoate  ...  370.0
101433-18-5                               Ethyl tetra decanol  ...  380.0
109309-32-2                          2,2-Di - p - toly butane  ...  480.0

[432 rows x 7 columns]

In [4]: chemicals.viscosity.mu_data_VN2
Out[4]:
                                        Name          Formula  ...   Tmin   Tmax
CAS                                                            ...
```

(continues on next page)

(continued from previous page)

```
71-36-3                              1-Butanol          C4 H10O   ...  220.0  390.0
74-87-3                         Methyl chloride            CH3Cl   ...  250.0  310.0
74-88-4                            Iodo methane             CH3I   ...  270.0  320.0
75-08-1                            Ethane thiol            C2H6S   ...  270.0  300.0
75-18-3                           Methyl sulfide           C2 H6S   ...  270.0  310.0
...                                          ...              ...  ...    ...    ...
12200-64-5             Sodium hydroxide hydrate        NaOH. H2O   ...  330.0  360.0
13478-00-7        Nickle - nitrate hexa hydrate   Ni(NO3)2. 6H2O   ...  330.0  350.0
18358-66-2  3 - n - Propyl - 4 - methyl sydnone      C6 H10N2 O2   ...  290.0  320.0
29136-19-4                   Nona decyl benzene          C25H44   ...  300.0  350.0
31304-44-6               Sodium acetate hydrate   CH3COONa. 3H2O   ...  330.0  360.0

[135 rows x 6 columns]

In [5]: chemicals.viscosity.mu_data_VN2E
Out[5]:
                                Substance    Formula  ...   Tmin   Tmax
CAS                                                   ...
60-29-7                             Ether     C4H10O   ...  270.0  410.0
64-19-7                       Acetic acid     C2H4O2   ...  270.0  390.0
75-07-0                      Acetaldehyde     C2H4O2   ...  270.0  300.0
75-25-2                         Bromoform      CHBr3   ...  280.0  350.0
78-93-3                 Methylketone ethyl      C4H8O   ...  240.0  360.0
109-73-9                       Butyl amine     C4H11N   ...  270.0  360.0
110-58-7                        Amyl amine     C5H13N   ...  270.0  360.0
111-26-2                     n-Hexyl amine     C6H15N   ...  270.0  380.0
764-49-8                  Allyl thiocynate     C4H5NS   ...  290.0  400.0
2307-17-7              Hexyl thio myrisate   C20H40OS   ...  300.0  370.0
10034-85-2                 Hydrogen iodide         HI   ...  220.0  240.0
10035-10-6                Hydrogen bromide         HBr   ...  180.0  200.0
28488-34-8                   Methylacetate     C3H6O2   ...  270.0  420.0
37340-18-4  Perfluoro-1- isopropoxy hexane     C9F20O   ...  290.0  320.0

[14 rows x 6 columns]

In [6]: chemicals.viscosity.mu_data_Perrys_8E_2_313
Out[6]:
                           Chemical        C1        C2  ...    C5    Tmin    Tmax
CAS                                                      ...
50-00-0                Formaldehyde   -11.2400    751.69  ...   0.0  181.15  254.05
55-21-0                   Benzamide   -12.6320   2668.20  ...   0.0  403.00  563.15
56-23-5        Carbon tetrachloride    -8.0738   1121.10  ...   0.0  250.00  455.00
57-55-6       1,2-Propylene glycol  -804.5400  30487.00  ...   1.0  213.15  500.80
60-29-7              Diethyl ether    10.1970    -63.80  ...   0.0  200.00  373.15
...                            ...       ...       ...  ...   ...     ...     ...
10028-15-6                   Ozone   -10.9400    415.96  ...   0.0   77.55  208.80
10035-10-6       Hydrogen bromide   -11.6330    316.38  ...   0.0  185.15  206.45
10102-43-9           Nitric oxide  -246.6500   3150.30  ...   1.0  109.50  180.05
13511-13-2     Propenylcyclohexene   -11.2080   1079.80  ...   0.0  199.00  508.80
132259-10-0                    Air   -20.0770    285.15  ...  10.0   59.15  130.00

[337 rows x 8 columns]
```

(continues on next page)

(continued from previous page)

```
In [7]: chemicals.viscosity.mu_data_Perrys_8E_2_312
Out[7]:
                          Chemical            C1       C2  ...   C4    Tmin    Tmax
CAS                                                       ...
50-00-0               Formaldehyde   4.758000e-07  0.64050  ...  0.0  181.15  1000.0
55-21-0                  Benzamide   2.508200e-08  0.96663  ...  0.0  403.00  1000.0
56-23-5       Carbon tetrachloride   3.137000e-06  0.37420  ...  0.0  250.33  1000.0
57-55-6      1,2-Propylene glycol   4.543000e-08  0.91730  ...  0.0  213.15  1000.0
60-29-7             Diethyl ether   1.948000e-06  0.41000  ...  0.0  156.85  1000.0
...                           ...           ...      ...  ...  ...     ...     ...
10028-15-6                 Ozone   1.196000e-07  0.84797  ...  0.0   80.15  1000.0
10035-10-6       Hydrogen bromide   9.170000e-08  0.92730  ...  0.0  206.45   800.0
10102-43-9           Nitric oxide   1.467000e-06  0.51230  ...  0.0  110.00  1500.0
13511-13-2    Propenylcyclohexene   5.474900e-07  0.53893  ...  0.0  199.00  1000.0
132259-10-0                   Air   1.425000e-06  0.50390  ...  0.0   80.00  2000.0

[345 rows x 7 columns]

In [8]: chemicals.viscosity.mu_data_VDI_PPDS_7
Out[8]:
                        Chemical Formula        A  ...        C        D         E
CAS                                                ...
50-00-0               Formaldehyde    CH2O  0.69796  ...  549.921  -44.110  0.000036
56-23-5       Carbon tetrachloride    CCl4  0.83033  ...  562.119  -73.328  0.000099
56-81-5                   Glycerol  C3H8O3 -3.91153  ...  582.480   73.885  0.007996
60-29-7              Diethyl ether  C4H10O  2.19245  ...  520.594 -370.873  0.000020
62-53-3                    Aniline   C6H7N  0.85750  ...  462.011  136.981  0.000282
...                            ...     ...      ...  ...      ...      ...       ...
10097-32-2                 Bromine     Br2  3.19074  ...  499.481 -209.817  0.000058
10102-43-9             Nitric oxide      NO  7.22569  ...  202.500 -106.123  0.000002
10102-44-0        Nitrogen dioxide     NO2  6.86768  ...  423.463 -446.706  0.000009
10544-72-6     Dinitrogentetroxide    N2O4 -0.03739  ...  615.987   11.286  0.000139
132259-10-0                   Air     NaN  2.22755  ...  132.897    4.000  0.000016

[271 rows x 7 columns]

In [9]: chemicals.viscosity.mu_data_VDI_PPDS_8
Out[9]:
                        Chemical             A  ...             D             E
CAS                                             ...
50-00-0               Formaldehyde -8.285000e-07  ...  0.000000e+00  0.000000e+00
56-23-5       Carbon tetrachloride -7.132000e-07  ...  0.000000e+00  0.000000e+00
56-81-5                   Glycerol -1.460000e-08  ...  0.000000e+00  0.000000e+00
60-29-7              Diethyl ether -8.933000e-07  ...  0.000000e+00  0.000000e+00
62-53-3                    Aniline -9.488000e-07  ...  0.000000e+00  0.000000e+00
...                            ...           ...  ...           ...           ...
10097-32-2                 Bromine  1.948300e-06  ...  0.000000e+00  0.000000e+00
10102-43-9             Nitric oxide -9.105000e-07  ...  4.240000e-14 -1.020000e-17
10102-44-0        Nitrogen dioxide -2.285050e-05  ...  1.713400e-13 -4.920000e-17
10544-72-6     Dinitrogentetroxide -8.683000e-07  ...  0.000000e+00  0.000000e+00
132259-10-0                   Air -1.702000e-07  ...  4.960000e-14 -1.388000e-17
```

(continues on next page)

(continued from previous page)

```
[274 rows x 6 columns]
```

1.36 Density/Volume (chemicals.volume)

This module contains various volume/density estimation routines, dataframes of fit coefficients, and mixing rules.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker.

1.36.1 Pure Low Pressure Liquid Correlations

chemicals.volume.**Rackett**(*T*, *Tc*, *Pc*, *Zc*)

Calculates saturation liquid volume, using Rackett CSP method and critical properties.

The molar volume of a liquid is given by:

$$V_s = \frac{RT_c}{P_c} Z_c^{[1+(1-T/T_c)^{2/7}]}$$

Units are all currently in m^3/mol - this can be changed to kg/m^3

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

Zc [float] Critical compressibility of fluid, [-]

Returns

Vs [float] Saturation liquid volume, [m^3/mol]

Notes

According to Reid et. al, underpredicts volume for compounds with Zc < 0.22

References

[1]

Examples

Propane, example from the API Handbook

```
>>> from chemicals.utils import Vm_to_rho
>>> Vm_to_rho(Rackett(272.03889, 369.83, 4248000.0, 0.2763), 44.09562)
531.3221411755724
```

chemicals.volume.**COSTALD**(*T*, *Tc*, *Vc*, *omega*)

Calculate saturation liquid density using the COSTALD CSP method.

A popular and accurate estimation method. If possible, fit parameters are used; alternatively critical properties work well.

The density of a liquid is given by:

$$V_s = V^* V^{(0)}[1 - \omega_{SRK} V^{(\delta)}]$$

$$V^{(0)} = 1 - 1.52816(1 - T_r)^{1/3} + 1.43907(1 - T_r)^{2/3} - 0.81446(1 - T_r) + 0.190454(1 - T_r)^{4/3}$$

$$V^{(\delta)} = \frac{-0.296123 + 0.386914 T_r - 0.0427258 T_r^2 - 0.0480645 T_r^3}{T_r - 1.00001}$$

Units are that of critical or fit constant volume.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Vc [float] Critical volume of fluid [m^3/mol]. This parameter is alternatively a fit parameter

omega [float] (ideally SRK) Acentric factor for fluid, [-] This parameter is alternatively a fit parameter.

Returns

Vs [float] Saturation liquid volume

Notes

196 constants are fit to this function in [1]. Range: 0.25 < Tr < 0.95, often said to be to 1.0

This function has been checked with the API handbook example problem.

References

[1]

Examples

Propane, from an example in the API Handbook:

```
>>> from chemicals.utils import Vm_to_rho
>>> Vm_to_rho(COSTALD(272.03889, 369.83333, 0.20008161E-3, 0.1532), 44.097)
530.3009967969844
```

chemicals.volume.**Yen_Woods_saturation**(*T*, *Tc*, *Vc*, *Zc*)

Calculates saturation liquid volume, using the Yen and Woods [1] CSP method and a chemical's critical properties.

The molar volume of a liquid is given by:

$$Vc/Vs = 1 + A(1-T_r)^{1/3} + B(1-T_r)^{2/3} + D(1-T_r)^{4/3}$$

$$D = 0.93 - B$$

$$A = 17.4425 - 214.578Z_c + 989.625Z_c^2 - 1522.06Z_c^3$$

$$B = -3.28257 + 13.6377Z_c + 107.4844Z_c^2 - 384.211Z_c^3 \text{ if } Zc \leq 0.26$$

$$B = 60.2091 - 402.063Z_c + 501.0Z_c^2 + 641.0Z_c^3 \text{ if } Zc \geq 0.26$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Vc [float] Critical volume of fluid [m^3/mol]

Zc [float] Critical compressibility of fluid, [-]

Returns

Vs [float] Saturation liquid volume, [m^3/mol]

Notes

Original equation was in terms of density, but it is converted here.

No example has been found, nor are there points in the article. However, it is believed correct. For compressed liquids with the Yen-Woods method, see the *YenWoods_compressed* function.

References

[1]

Examples

```
>>> Yen_Woods_saturation(300, 647.14, 55.45E-6, 0.245)
1.769533076529574e-05
```

chemicals.volume.**Yamada_Gunn**(*T*, *Tc*, *Pc*, *omega*)

Calculates saturation liquid volume, using Yamada and Gunn CSP method and a chemical's critical properties and acentric factor.

The molar volume of a liquid is given by:

$$V_s = \frac{RT_c}{P_c}(0.29056 - 0.08775\omega)^{[1+(1-T/T_c)^{2/7}]}$$

Units are in m^3/mol.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

Vs [float] saturation liquid volume, [m^3/mol]

Notes

This equation is an improvement on the Rackett equation. This is often presented as the Rackett equation. The acentric factor is used here, instead of the critical compressibility A variant using a reference fluid also exists

References

[1], [2]

Examples

```
>>> Yamada_Gunn(300, 647.14, 22048320.0, 0.245)
2.188284384699659e-05
```

chemicals.volume.**Townsend_Hales**(*T*, *Tc*, *Vc*, *omega*)

Calculates saturation liquid density, using the Townsend and Hales CSP method as modified from the original Riedel equation. Uses chemical critical volume and temperature, as well as acentric factor

The density of a liquid is given by:

$$Vs = V_c / \left(1 + 0.85(1 - T_r) + (1.692 + 0.986\omega)(1 - T_r)^{1/3}\right)$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Vc [float] Critical volume of fluid [m^3/mol]

omega [float] Acentric factor for fluid, [-]

Returns

Vs [float] Saturation liquid volume, [m^3/mol]

Notes

The requirement for critical volume and acentric factor requires all data.

References

[1]

Examples

```
>>> Townsend_Hales(300, 647.14, 55.95E-6, 0.3449)
1.8007361992619923e-05
```

chemicals.volume.**Bhirud_normal**(*T*, *Tc*, *Pc*, *omega*)

Calculates saturation liquid density using the Bhirud [1] CSP method. Uses Critical temperature and pressure and acentric factor.

The density of a liquid is given by:

$$\ln \frac{P_c}{\rho RT} = \ln U^{(0)} + \omega \ln U^{(1)}$$

$$\ln U^{(0)} = 1.39644 - 24.076T_r + 102.615T_r^2 - 255.719T_r^3 + 355.805T_r^4 - 256.671T_r^5 + 75.1088T_r^6$$

$$\ln U^{(1)} = 13.4412 - 135.7437T_r + 533.380T_r^2 - 1091.453T_r^3 + 1231.43T_r^4 - 728.227T_r^5 + 176.737T_r^6$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] Acentric factor for fluid, [-]

Returns

Vm [float] Saturated liquid molar volume, [mol/m^3]

Notes

Claimed inadequate by others.

An interpolation table for ln U values are used from Tr = 0.98 - 1.000. Has terrible behavior at low reduced temperatures.

References

[1]

Examples

Pentane

```
>>> Bhirud_normal(280.0, 469.7, 33.7E5, 0.252)
0.00011249657842514176
```

chemicals.volume.**Campbell_Thodos**(*T*, *Tb*, *Tc*, *Pc*, *MW*, *dipole=0.0*, *has_hydroxyl=False*)

Calculate saturation liquid density using the Campbell-Thodos [1] CSP method.

An old and uncommon estimation method.

$$V_s = \frac{RT_c}{P_c} Z_{RA}^{[1+(1-T_r)^{2/7}]}$$

$$Z_{RA} = \alpha + \beta(1 - T_r)$$

$$\alpha = 0.3883 - 0.0179s$$

$$s = T_{br} \frac{\ln P_c}{(1 - T_{br})}$$

$$\beta = 0.00318s - 0.0211 + 0.625\Lambda^{1.35}$$

$$\Lambda = \frac{P_c^{1/3}}{MW^{1/2} T_c^{5/6}}$$

For polar compounds:

$$\theta = P_c \mu^2 / T_c^2$$

$$\alpha = 0.3883 - 0.0179s - 130540\theta^{2.41}$$

$$\beta = 0.00318s - 0.0211 + 0.625\Lambda^{1.35} + 9.74 \times 10^6 \theta^{3.38}$$

Polar Combounds with hydroxyl groups (water, alcohols)

$$\alpha = \left[0.690 T_{br} - 0.3342 + \frac{5.79 \times 10^{-10}}{T_{br}^{32.75}}\right] P_c^{0.145}$$

$$\beta = 0.00318s - 0.0211 + 0.625\Lambda^{1.35} + 5.90\Theta^{0.835}$$

Parameters

T [float] Temperature of fluid [K]

Tb [float] Boiling temperature of the fluid [K]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

MW [float] Molecular weight of the fluid [g/mol]

dipole [float, optional] Dipole moment of the fluid [debye]

has_hydroxyl [bool, optional] Swith to use the hydroxyl variant for polar fluids

Returns

Vs [float] Saturation liquid volume, [m^3/mol]

Notes

If a dipole is provided, the polar chemical method is used. The paper is an excellent read. Pc is internally converted to atm.

References

[1]

Examples

Ammonia, from [1].

```
>>> Campbell_Thodos(T=405.45, Tb=239.82, Tc=405.45, Pc=111.7*101325, MW=17.03,␣
↪dipole=1.47)
7.347366126245e-05
```

chemicals.volume.**SNM0**(*T*, *Tc*, *Vc*, *omega*, *delta_SRK=None*)

Calculates saturated liquid density using the Mchaweh, Moshfeghian model [1]. Designed for simple calculations.

$$V_s = V_c/(1 + 1.169\tau^{1/3} + 1.818\tau^{2/3} - 2.658\tau + 2.161\tau^{4/3}$$

$$\tau = 1 - \frac{(T/T_c)}{\alpha_{SRK}}$$

$$\alpha_{SRK} = [1 + m(1 - \sqrt{T/T_C}]^2$$

$$m = 0.480 + 1.574\omega - 0.176\omega^2$$

If the fit parameter *delta_SRK* is provided, the following is used:

$$V_s = V_C/(1 + 1.169\tau^{1/3} + 1.818\tau^{2/3} - 2.658\tau + 2.161\tau^{4/3})/\left[1 + \delta_{SRK}(\alpha_{SRK} - 1)^{1/3}\right]$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

Vc [float] Critical volume of fluid [m^3/mol]

omega [float] Acentric factor for fluid, [-]

delta_SRK [float, optional] Fitting parameter [-]

Returns

Vs [float] Saturation liquid volume, [m^3/mol]

Notes

73 fit parameters have been gathered from the article.

References

[1]

Examples

Argon, without the fit parameter and with it. Tabulated result in Perry's is 3.4613e-05. The fit increases the error on this occasion.

```
>>> SNM0(121, 150.8, 7.49e-05, -0.004)
3.440225640273e-05
>>> SNM0(121, 150.8, 7.49e-05, -0.004, -0.03259620)
3.493288100008e-05
```

1.36.2 Pure High Pressure Liquid Correlations

chemicals.volume.**COSTALD_compressed**(*T*, *P*, *Psat*, *Tc*, *Pc*, *omega*, *Vs*)

Calculates compressed-liquid volume, using the COSTALD [1] CSP method and a chemical's critical properties.

The molar volume of a liquid is given by:

$$V = V_s\left(1 - C\ln\frac{B+P}{B+P^{sat}}\right)$$

$$\frac{B}{P_c} = -1 + a\tau^{1/3} + b\tau^{2/3} + d\tau + e\tau^{4/3}$$

$$e = \exp(f + g\omega_{SRK} + h\omega_{SRK}^2)$$

$$C = j + k\omega_{SRK}$$

Parameters

T [float] Temperature of fluid [K]

P [float] Pressure of fluid [Pa]

Psat [float] Saturation pressure of the fluid [Pa]

Tc [float] Critical temperature of fluid [K]

Pc [float] Critical pressure of fluid [Pa]

omega [float] (ideally SRK) Acentric factor for fluid, [-] This parameter is alternatively a fit parameter.

Vs [float] Saturation liquid volume, [m^3/mol]

Returns

V_dense [float] High-pressure liquid volume, [m^3/mol]

Notes

Original equation was in terms of density, but it is converted here.

The example is from DIPPR, and exactly correct. This is DIPPR Procedure 4C: Method for Estimating the Density of Pure Organic Liquids under Pressure.

References

[1]

Examples

```
>>> COSTALD_compressed(303., 9.8E7, 85857.9, 466.7, 3640000.0, 0.281, 0.000105047)
9.287482879788505e-05
```

1.36.3 Liquid Mixing Rules

chemicals.volume.**Amgat**(*xs*, *Vms*)

Calculate mixture liquid density using the Amgat mixing rule. Highly inacurate, but easy to use. Assumes idea liquids with no excess volume. Average molecular weight should be used with it to obtain density.

$$V_{mix} = \sum_i x_i V_i$$

or in terms of density:

$$\rho_{mix} = \sum \frac{x_i}{\rho_i}$$

Parameters

xs [array] Mole fractions of each component, []

Vms [array] Molar volumes of each fluids at conditions [m^3/mol]

Returns

Vm [float] Mixture liquid volume [m^3/mol]

Notes

Units are that of the given volumes. It has been suggested to use this equation with weight fractions, but the results have been less accurate.

Examples

```
>>> Amgat([0.5, 0.5], [4.057e-05, 5.861e-05])
4.9590000000000005e-05
```

chemicals.volume.**Rackett_mixture**(*T*, *xs*, *MWs*, *Tcs*, *Pcs*, *Zrs*)

Calculate mixture liquid density using the Rackett-derived mixing rule as shown in [2].

$$V_m = \sum_i \frac{x_i T_{ci}}{MW_i P_{ci}} Z_{R,m}^{(1+(1-T_r)^{2/7})} R \sum_i x_i MW_i$$

Parameters

T [float] Temperature of liquid [K]

xs: list Mole fractions of each component, []

MWs [list] Molecular weights of each component [g/mol]

Tcs [list] Critical temperatures of each component [K]

Pcs [list] Critical pressures of each component [Pa]

Zrs [list] Rackett parameters of each component []

Returns

Vm [float] Mixture liquid volume [m^3/mol]

Notes

Model for pure compounds in [1] forms the basis for this model, shown in [2]. Molecular weights are used as weighing by such has been found to provide higher accuracy in [2]. The model can also be used without molecular weights, but results are somewhat different.

As with the Rackett model, critical compressibilities may be used if Rackett parameters have not been regressed.

Critical mixture temperature, and compressibility are all obtained with simple mixing rules.

References

[1], [2]

Examples

Calculation in [2] for methanol and water mixture. Result matches example.

```
>>> Rackett_mixture(T=298., xs=[0.4576, 0.5424], MWs=[32.04, 18.01], Tcs=[512.58,␣
↪647.29], Pcs=[8.096E6, 2.209E7], Zrs=[0.2332, 0.2374])
2.6252894930056885e-05
```

chemicals.volume.**COSTALD_mixture**(*xs*, *T*, *Tcs*, *Vcs*, *omegas*)

Calculate mixture liquid density using the COSTALD CSP method.

A popular and accurate estimation method. If possible, fit parameters are used; alternatively critical properties work well.

The mixing rules giving parameters for the pure component COSTALD equation are:

$$T_{cm} = \frac{\sum_i \sum_j x_i x_j (V_{ij} T_{cij})}{V_m}$$

$$V_m = 0.25 \left[\sum x_i V_i + 3 (\sum x_i V_i^{2/3})(\sum_i x_i V_i^{1/3}) \right]$$

$$V_{ij}T_{cij} = (V_iT_{ci}V_jT_{cj})^{0.5}$$

$$\omega = \sum_i z_i\omega_i$$

Parameters

xs: list Mole fractions of each component

T [float] Temperature of fluid [K]

Tcs [list] Critical temperature of fluids [K]

Vcs [list] Critical volumes of fluids [m^3/mol]. This parameter is alternatively a fit parameter

omegas [list] (ideally SRK) Acentric factor of all fluids, [-] This parameter is alternatively a fit parameter.

Returns

Vs [float] Saturation liquid mixture volume

Notes

Range: 0.25 < Tr < 0.95, often said to be to 1.0 No example has been found. Units are that of critical or fit constant volume.

References

[1]

Examples

```
>>> COSTALD_mixture([0.4576, 0.5424], 298.,  [512.58, 647.29], [0.000117, 5.6e-05],␣
↪[0.559,0.344])
2.7065887732713534e-05
```

1.36.4 Gas Correlations

Gas volumes are predicted with one of:

1) An equation of state
2) A virial coefficient model
3) The ideal gas law

Equations of state do much more than predict volume however. An implementation of many of them can be found in thermo.

Virial functions are implemented in *chemicals.virial*.

chemicals.volume.**ideal_gas**(*T*, *P*)

Calculates ideal gas molar volume. The molar volume of an ideal gas is given by:

$$V = \frac{RT}{P}$$

Parameters

T [float] Temperature of fluid [K]

P [float] Pressure of fluid [Pa]

Returns

V [float] Gas volume, [m^3/mol]

Examples

```
>>> ideal_gas(298.15, 101325.)
0.024465403697038125
```

1.36.5 Pure Solid Correlations

Solid density does not depend on pressure significantly, and unless operating in the geochemical or astronomical domain is normally neglected.

chemicals.volume.**Goodman**(*T*, *Tt*, *Vml*)

Calculates solid density at T using the simple relationship by a member of the DIPPR.

The molar volume of a solid is given by:

$$\frac{1}{V_m} = \left(1.28 - 0.16\frac{T}{T_t}\right)\frac{1}{Vm_L(T_t)}$$

Parameters

T [float] Temperature of fluid [K]

Tt [float] Triple temperature of fluid [K]

Vml [float] Liquid molar volume of the organic liquid at the triple point, [m^3/mol]

Returns

Vms [float] Solid molar volume, [m^3/mol]

Notes

Works to the next solid transition temperature or to approximately 0.3Tt.

References

[1]

Examples

Decane at 200 K:

```
>>> Goodman(200, 243.225, 0.00023585)
0.0002053665090860923
```

1.36.6 Pure Component Liquid Fit Correlations

chemicals.volume.**Rackett_fit**(*T*, *Tc*, *rhoc*, *b*, *n*, *MW=None*)

Calculates saturation liquid volume, using the Rackett equation form and a known or estimated critical temperature and density as well as fit parameters *b* and *n*.

The density of a liquid is given by:

$$\rho_{sat} = \rho_c b^{-\left(1-\frac{T}{T_c}\right)^n}$$

The density is then converted to a specific volume by taking its inverse.

Note that the units of this equation in some sources are kg/m^3, g/mL in others, and m^3/mol in others. If the units for the coefficients are in molar units, do NOT provide *MW* or an incorrect value will be returned. If the units are mass units and *MW* is not provided, the output will have the same units as *rhoc*.

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

rhoc [float] Critical density of fluid, often a fit parameter only [kg/m^3]

b [float] Fit parameter, [-]

n [float] Fit parameter, [-]

MW [float, optional] Molecular weight, [g/mol]

Returns

Vs [float] Saturation liquid volume, [m^3/mol if MW given; m^3/kg otherwise]

References

[1], [2]

Examples

Input sample from NIST (naphthalene) (m^3/kg):

```
>>> Rackett_fit(T=400.0, Tc=748.402, rhoc=314.629, b=0.257033, n=0.280338)
0.00106174320755
```

Parameters in Yaws form (butane) (note the 1000 multiplier on *rhoc*, called *A* in Yaws) (m^3/kg):

```
>>> Rackett_fit(T=298.15, Tc=425.18, rhoc=0.2283*1000, b=0.2724, n=0.2863)
0.00174520519958
```

Same Yaws point, with MW provided:

```
>>> Rackett_fit(T=298.15, Tc=425.18, rhoc=0.2283*1000, b=0.2724, n=0.2863, MW=58.
↪123)
0.00010143656181
```

chemicals.volume.**volume_VDI_PPDS**(*T*, *Tc*, *rhoc*, *a*, *b*, *c*, *d*, *MW=None*)

Calculates saturation liquid volume, using the critical properties and fitted coefficients from [1]. This is also known as the PPDS equation 10 or PPDS10.

$$\rho_{mass} = \rho_c + a\tau^{0.35} + b\tau^{2/3} + c\tau + d\tau^{4/3}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

rhoc [float] Critical density of fluid [kg/m^3]

a,b,c,d [float] Fitted coefficients [-]

MW [float, optional] Molecular weight of chemical [g/mol]

Returns

Vs [float] Saturation liquid molar volume or density, [m^3/mol if MW given; kg/m^3 otherwise]

References

[1]

Examples

Calculate density of nitrogen in kg/m3 at 300 K:

```
>>> volume_VDI_PPDS(300, 126.19, 313, 470.922, 493.251, -560.469, 389.611)
313.0
```

Calculate molar volume of nitrogen in m3/mol at 300 K:

```
>>> volume_VDI_PPDS(300, 126.19, 313, 470.922, 493.251, -560.469, 389.611, 28.01)
8.9488817891e-05
```

chemicals.volume.**TDE_VDNS_rho**(*T*, *Tc*, *rhoc*, *a1*, *a2*, *a3*, *a4*, *MW=None*)

Calculates saturation liquid volume, using the critical properties and fitted coefficients in the TDE VDNW form from [1].

$$\rho_{mass} = \rho_c + a\tau^{0.35} + b\tau + c\tau^2 + d\tau^3$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

rhoc [float] Critical density of fluid [kg/m^3]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

a3 [float] Regression parameter, [-]

a4 [float] Regression parameter, [-]

MW [float, optional] Molecular weight of chemical [g/mol]

Returns

Vs [float] Saturation liquid molar volume or density, [m^3/mol if MW given; kg/m^3 otherwise]

References

[1]

Examples

```
>>> TDE_VDNS_rho(T=400.0, Tc=772.999, rhoc=320.037, a1=795.092, a2=-169.132, a3=448.
↪929, a4=-102.931)
947.4906064903
```

chemicals.volume.**PPDS17**(*T*, *Tc*, *a0*, *a1*, *a2*, *MW=None*)

Calculates saturation liquid volume, using the critical temperature and fitted coefficients in the PPDS17 form in [1].

$$\rho_{mass} = \frac{1}{a_0(a_1 + a_2\tau)^{\left(1+\tau^{2/7}\right)}}$$

Parameters

T [float] Temperature of fluid [K]

Tc [float] Critical temperature of fluid [K]

a0 [float] Regression parameter, [-]

a1 [float] Regression parameter, [-]

a2 [float] Regression parameter, [-]

MW [float, optional] Molecular weight of chemical [g/mol]

Returns

Vs [float] Saturation liquid molar volume or density, [m^3/mol if MW given; kg/m^3 otherwise]

References

[1]

Examples

Coefficients for the liquid density of benzene from [1] at 300 K:

```
>>> PPDS17(300, 562.05, a0=0.0115508, a1=0.281004, a2=-0.00635447)
871.520087707
```

1.36.7 Pure Component Solid Fit Correlations

chemicals.volume.**CRC_inorganic**(*T*, *rho0*, *k*, *Tm*, *MW=None*)

Calculates liquid density of a molten element or salt at temperature above the melting point. Some coefficients are given nearly up to the boiling point.

The mass density of the inorganic liquid is given by:

$$\rho = \rho_0 - k(T - T_m)$$

Parameters

T [float] Temperature of the liquid, [K]

rho0 [float] Mass density of the liquid at Tm, [kg/m^3]

k [float] Linear temperature dependence of the mass density, [kg/m^3/K]

Tm [float] The normal melting point, used in the correlation [K]

MW [float, optional] Molecular weight of chemical [g/mol]

Returns

rho [float] Mass density of molten metal or salt, [m^3/mol if MW given; kg/m^3 otherwise]

Notes

[1] has units of g/mL. While the individual densities could have been converted to molar units, the temperature coefficient could only be converted by refitting to calculated data. To maintain compatibility with the form of the equations, this was not performed.

This linear form is useful only in small temperature ranges. Coefficients for one compound could be used to predict the temperature dependence of density of a similar compound.

References

[1]

Examples

```
>>> CRC_inorganic(300, 2370.0, 2.687, 239.08)
2206.30796
```

1.36.8 Fit Coefficients

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

chemicals.volume.**rho_data_COSTALD**

Coefficients for the *COSTALD* method from [3]; 192 fluids have coefficients published.

chemicals.volume.**rho_data_SNM0**

Coefficients for the *SNM0* method for 73 fluids from [2].

chemicals.volume.**rho_data_Perry_8E_105_l**

Coefficients for *chemicals.dippr.EQ105* from [1] for 344 fluids. Note this is in terms of molar density; to obtain molar volume, invert the result!

chemicals.volume.**rho_data_VDI_PPDS_2**
: Coefficients in [5] developed by the PPDS using *chemicals.dippr.EQ116* but in terms of mass density [kg/m^3]; Valid up to the critical temperature, and extrapolates to very low temperatures well.

chemicals.volume.**rho_data_CRC_inorg_l**
: Single-temperature coefficient linear model in terms of mass density for the density of inorganic liquids. Data is available for 177 fluids normally valid over a narrow range above the melting point, from [4]; described in *CRC_inorganic*.

chemicals.volume.**rho_data_CRC_inorg_l_const**
: Constant inorganic liquid molar volumes published in [4].

chemicals.volume.**rho_data_CRC_inorg_s_const**
: Constant solid densities molar volumes published in [4].

chemicals.volume.**rho_data_CRC_virial**
: Coefficients for a tempereture polynomial (T in Kelvin) for the second B virial coefficient published in [4]. The form of the equation is $B = (a_1 + t(a_2 + t(a_3 + t(a_4 + a_5 t)))) \times 10^{-6}$ with $t = \frac{298.15}{T} - 1$ and then B will be in units of m^3/mol.

The structure of each dataframe is shown below:

```
In [1]: import chemicals

In [2]: chemicals.volume.rho_data_COSTALD
Out[2]:
                            Chemical  omega_SRK     Vchar    Z_RA
CAS
60-29-7                   ethyl ether     0.2800  0.000281  0.2632
64-17-5                 ethyl alcohol     0.6378  0.000175  0.2502
67-56-1                methyl alcohol     0.5536  0.000120  0.2334
67-63-0             isopropyl alcohol     0.6637  0.000231  0.2493
67-64-1                       acetone     0.3149  0.000208  0.2477
...                               ...        ...       ...     ...
14752-75-1           heptadecylbenzene     0.9404  0.001146     NaN
30453-31-7    ethyl n-propyl disulfide     0.3876  0.000440  0.2662
33672-51-4  propyl isopropyl disulfide     0.4059  0.000502  0.2680
53966-36-2   ethyl isopropyl disulfide     0.3556  0.000439  0.2711
61828-04-4             tricosylbenzene     1.1399  0.001995     NaN

[192 rows x 4 columns]

In [3]: chemicals.volume.rho_data_SNM0
Out[3]:
                            Chemical  delta_SRK
CAS
56-23-5    Tetrachlouromethane, R-10   -0.013152
60-29-7                   Ethylether    0.001062
64-19-7                  Acetic acid   -0.010347
65-85-0                Benzoic acid    0.026866
67-56-1                    Methanol    0.007195
...                              ...        ...
7727-37-9                  Nitrogen   -0.007946
7782-39-0                 Deuterium   -0.053345
7782-41-4                  Flourine   -0.030398
7782-44-7                    Oxygen   -0.027049
```

(continues on next page)

(continued from previous page)

```
7782-50-5                   Chlorine    0.013010

[73 rows x 2 columns]

In [4]: chemicals.volume.rho_data_Perry_8E_105_l
Out[4]:
                         Chemical       C1       C2  ...       C4    Tmin    Tmax
CAS                                                  ...
50-00-0               Formaldehyde   1941.50  0.22309  ...  0.28571  181.15  408.00
55-21-0                  Benzamide    737.10  0.25487  ...  0.28571  403.00  824.00
56-23-5      Carbon tetrachloride    998.35  0.27400  ...  0.28700  250.33  556.35
57-55-6      1,2-Propylene glycol   1092.30  0.26106  ...  0.20459  213.15  626.00
60-29-7             Diethyl ether    955.40  0.26847  ...  0.28140  156.85  466.70
...                           ...      ...      ...  ...      ...     ...     ...
10028-15-6                  Ozone   3359.20  0.29884  ...  0.28523   80.15  261.00
10035-10-6       Hydrogen bromide   2832.00  0.28320  ...  0.28571  185.15  363.15
10102-43-9           Nitric oxide   5246.00  0.30440  ...  0.24200  109.50  180.15
13511-13-2    Propenylcyclohexene    612.55  0.26769  ...  0.28571  199.00  636.00
132259-10-0                   Air   2896.30  0.26733  ...  0.27341   59.15  132.45

[344 rows x 7 columns]

In [5]: chemicals.volume.rho_data_VDI_PPDS_2
Out[5]:
                         Chemical      MW  ...          C          D
CAS                                        ...
50-00-0               Formaldehyde   30.03  ...   245.3425    43.9601
56-23-5      Carbon tetrachloride  153.82  ...   535.7568   -28.0071
56-81-5                   Glycerol   92.09  ...  1429.7634  -527.7710
60-29-7             Diethyl ether   74.12  ...  -489.2726   486.7458
62-53-3                   Aniline   93.13  ...   242.0930     0.7157
...                           ...     ...  ...        ...        ...
10097-32-2                Bromine  159.82  ...   676.7593    15.3973
10102-43-9           Nitric oxide   30.01  ...  2252.1437 -1031.3210
10102-44-0        Nitrogen dioxide   46.01  ...  2233.6217  -968.0655
10544-72-6    Dinitrogentetroxide   92.01  ...   604.1720  -135.9384
132259-10-0                   Air   28.96  ...  -841.3265   495.5129

[272 rows x 8 columns]

In [6]: chemicals.volume.rho_data_CRC_inorg_l
Out[6]:
                         Chemical       MW      rho      k       Tm     Tmax
CAS
497-19-8           Sodium carbonate  105.989   1972.0  0.448  1129.15  1277.15
584-09-8         Rubidium carbonate  230.945   2840.0  0.640  1110.15  1280.15
7429-90-5                 Aluminum   26.982   2377.0  0.311   933.47  1190.15
7429-91-6               Dysprosium  162.500   8370.0  1.430  1685.15  1813.15
7439-88-5                  Iridium  192.217  19000.0  0.000  2719.15  2739.15
...                            ...      ...      ...    ...      ...      ...
13572-98-0  Gadolinium(III) iodide  537.960   4120.0  0.908  1203.15  1305.15
13709-38-1      Lanthanum fluoride  195.900   4589.0  0.682  1766.15  2450.15
```

(continues on next page)

(continued from previous page)

```
13709-59-6    Thorium(IV) fluoride  308.032   6058.0  0.759  1383.15  1651.15
13718-50-8           Barium iodide  391.136   4260.0  0.977   984.15  1248.15
13813-22-4       Lanthanum iodide  519.619   4290.0  1.110  1051.15  1180.15

[177 rows x 6 columns]

In [7]: chemicals.volume.rho_data_CRC_inorg_l_const
Out[7]:
                            Chemical        Vm
CAS
74-90-8              Hydrogen cyanide  0.000039
75-15-0              Carbon disulfide  0.000060
96-10-6          Chlorodiethylaluminum  0.000126
109-63-7    Boron trifluoride etherate  0.000126
289-22-5             Cyclopentasilane  0.000156
...                               ...       ...
19624-22-7             Pentaborane(9)  0.000105
20398-06-5     Thallium(I) ethanolate  0.000071
23777-80-2             Hexaborane(10)  0.000112
27218-16-2       Chlorine perchlorate  0.000075
52988-75-7         3-Silylpentasilane  0.000217

[116 rows x 2 columns]

In [8]: chemicals.volume.rho_data_CRC_inorg_s_const
Out[8]:
                                     Chemical        Vm
CAS
62-54-4                       Calcium acetate  0.000105
62-76-0                        Sodium oxalate  0.000057
75-20-7                       Calcium carbide  0.000029
127-08-2                    Potassium acetate  0.000063
127-09-3                       Sodium acetate  0.000054
...                                       ...       ...
75926-28-2             Selenium sulfide [Se4S4]  0.000135
84359-31-9   Chromium(III) phosphate hexahydrate  0.000120
92141-86-1                   Cesium metaborate  0.000047
133578-89-9            Vanadyl selenite hydrate  0.000060
133863-98-6        Molybdenum(VI) metaphosphate  0.000174

[1872 rows x 2 columns]

In [9]: chemicals.volume.rho_data_CRC_virial
Out[9]:
                         Chemical      a1       a2       a3       a4   a5
CAS
56-23-5        Tetrachloromethane -1600.0  -4059.0  -4653.0      0.0  0.0
60-29-7             Diethyl ether -1226.0  -4458.0  -7746.0 -10005.0  0.0
64-17-5                   Ethanol -4475.0 -29719.0 -56716.0      0.0  0.0
67-56-1                  Methanol -1752.0  -4694.0      0.0      0.0  0.0
67-63-0                2-Propanol -3165.0 -16092.0 -24197.0      0.0  0.0
...                            ...     ...      ...      ...      ...  ...
```

(continues on next page)

(continued from previous page)

```
7783-81-5    Uranium(VI) fluoride -1204.0  -2690.0  -2144.0      0.0  0.0
7783-82-6   Tungsten(VI) fluoride  -719.0  -1143.0      0.0      0.0  0.0
7803-51-2              Phosphine  -146.0   -733.0   1022.0  -1220.0  0.0
10024-97-2          Nitrous oxide  -130.0   -307.0   -248.0      0.0  0.0
10102-43-9           Nitric oxide   -12.0   -119.0     89.0    -73.0  0.0

[105 rows x 6 columns]
```

1.37 Developer's Guide and Roadmap

The chemicals project has the following core ideas as its goals:

- **Efficient**
  - Functions do only the work required.
  - Caching various values, precomputing others.
  - Using various macros and automated expressions to run code with Numba at its optimal speed.
  - Not using Numpy/SciPy most of the time, allowing PyPy or Numba to speed code up when speed is important.
- **Capable of vectorized computation**
  - Wrapped with numpy's np.vectorize.
  - Wrapped with numba's ufunc machinery.
- **Comprehensive**
  - Most correlations taught at the undergrad level included.
- **Capable of handling units**
  - Pint interface
  - All docstrings/code in base SI units

This is a hobby project primarily by Caleb Bell. Contributors are welcome! Chemicals properties is huge big field.

1.37.1 Scope and Future Features

The library has a focus on pure-component properties. Lots of data files are included and there is ample room for more files. However, it is not feasible to add data on an individual chemical alone - a compilation of data which includes that chemicals must be found instead.

The following properties have been looked at a little but may have too much data fit on PyPi. If you are interested in implementing one of them it may take multiple data packages uploaded to PyPi alongside *chemicals*:

- IR, NMR, MS, and UV-Vis spectra. Files are typically 4-40 KB. A collection of ~2000 UV files from NIST takes 6 MB space, so the space issue would not be ran into right away. Some databases like NIST and NMRShiftDB are open.

1.37.2 Contributing

Chemicals has a lot of infrastructure that makes it attractive to add code to the project. Adding functionality to *chemicals* may also make your work more visible to more people.

Adding new functionality is possible without compromising load speed, RAM usage or maintainability. If you have a chemical property you are interested in adding, a utility function, or a new data source, please feel free to open a PR and we can make any changes needed. There is no template - just do your best.

In an ideal world, new contributions would come with unit tests, docstrings, an addition to the tutorial if relevant.

1.37.3 Running Tests

From the root directory of the project you downloaded with *git clone https://github.com/CalebBell/chemicals.git*, run the following command:

```
python3 -m pytest .
```

This will run all of the tests. Additionally pytest can be used to run the doctests:

```
python3 -m pytest --doctest-modules .
```

The test suite can take some time to run; tests are marked with various markers to allow a fast subset of tests to run.

```
python3 -m pytest -m "not slow" .
```

This should only take a few seconds, and show red output if a test is broken. To keep the main test suite fast, pytest allows a flag which shows how long each test takes.

```
python3 -m pytest -m "not slow" --durations=100
```

If a test you added appears in this list, consider splitting it into a fast portion and a slow portion.

1.37.4 Docstrings

The docstrings follow Pep8, most of the numpydoc standard, More information about numpydoc can be found here

In addition to being documentation, the docstrings in chemicals serve the following purposes:

- Contain LaTeX math formulas for implemented formulas. This makes it easy for the reader and authors to follow code. This is especially important when the code can be optimized by hand significantly, and end up not looking like the math formulas.
- Contain doctests for every public method. These examples often make debugging really easy since they can just be copy-pasted into Jupyter or an IDE/debugger.
- Contain type information for each variable, which is automatically parsed by the unit handling framework around *pint*.
- Contain the units of each argument, which is used by the unit handling framework around *pint*.
- Contain docstrings for every argument - these are checked by the unit tests programatically to avoid forgetting to add a description, which the author did often before the checker was added.

No automated style tool is ran on the docstrings at present, but the following command was used once to format the docstrings with the tool docformatter

```
python3 -m docformatter --wrap-summaries=80 --wrap-descriptions=80 --in-place --
↪recursive .
```

This does not quite match numpydoc's recommended 75 character limit.

1.37.5 Doctest

As anyone who has used doctest before knows, floating-point calculations have trivially different results across platforms. An example cause of this is that most compilers have different sin/cos implementations which are not identical. However, docstrings are checked bit-for-bit, so consistent output is important. Python is better than most languages at maintaining the same results between versions but it is still an issue.

The docstrings are not considered sufficiently consistent to be part of the automated CI infrastructure. All functionality tested by docstrings should also be tested as part of the unit tests.

CPython 3.7 64 bit on Linux compiled with gcc 9.2 is currently the platform used to generate the final/official results of docstring examples. Docstrings are should be added by contributors for new functionality, but **don't worry about this floating point issue**. The principal author will make any necessary changes before each release.

1.37.6 Type Hints

The Python ecosystem is gradually adding support for type information, which may allow static analyzers to help find bugs in code even before it is ran. The author has not found these helpful in Python yet - the tools are too slow, missing features, and most libraries do not contain type information. However, type hints might still be useful for your program that uses *chemicals*!

For that reason *chemicals* includes a set of type hints as stub files (.pyi extension). These are not generated by hand - they use the cool MonkeyType library. An included script *make_test_stubs* interfaces with this library, which runs the test suite and at the end generates the type hints including the types of every argument to every function seen in the test suite. This is another reason comprehensive test suite coverage is required.

Monkeytype on the *chemicals* test suite takes ~5 minutes to run, and generates a ~1 GB database file which is deleted at the end of the run. Some manipulation of the result by hand may be required in the future, or MonkeyType may be replaced by making the type hints by hand. It is planned to incorporate the type stubs into the main file at some point in the future when the tooling is better.

If you are contributing, the main developer can do this step for your contribution.

1.37.7 Supported Python Versions

It is strongly recommended to use Chemicals with Python 3.6 or later.

Numpy's latest release supports Python 3.6 or later as of August 2020. Pint, the unit interface, supports Python 3.6 or later as of August 2020. If using the Numba interface of Chemicals, the latest version of Numba is required. New features added to Numba may be added to Chemicals quite quickly.

Chemicals may import in an earlier Python but that is not an indication you should use it in that case.

1.37.8 Packaging

The most up to date chemicals can be obtained on GitHub, and new releases are pushed to PyPi whenever a new release is made.

1.37.9 Code Formatting

Pep8 is loosely followed. Do your best to follow it if possible, otherwise don't worry about it. Please don't submit a PR for just style changes.

1.37.10 Documentation

Sphinx is used with readthedocs. Readthedocs is configured to build whatever is on the release branch. From the root of the *chemicals* project, the documentation can be built with the following command, which will output html files into a "_build" folder:

```
sphinx-build -b html docs _build
```

1.37.11 Sample Notebooks

The nbval pytest plugin can be used to check the results of running the notebooks against the stored notebooks.

On UNIX/Mac OS/WSL, the notebook results can be regenerated with the following shell command, from the directory with the notebooks:

```
for i in *.ipynb ; do python3 -m jupyter nbconvert --to notebook --inplace --execute "$i
↪" ; done
```

1.37.12 Continuous Integration

Github Actions is presently used. They test only code in the *release* branch. Some tests, like those that download data from the internet, are not ran by design on their platforms. The same goes for testing *numba* online - getting an up to date version of numba is challenging.

1.37.13 Load Speed

On CPython, *chemicals* will load Numpy on load if it is available and *SciPy* and *Pandas* when they are needed. Numpy loads in ~150 ms. chemicals alone loads in ~15 ms. It is intended for this to increase only slowly.

1.37.14 RAM Usage

Adding new data and methods is well worth the price of increasing RAM, but it is intended to keep RAM consumption small via lazy-loading all data sets.

1.37.15 Additional Material

More information about the interfaces with PyPy and Numba can be found on the fluids site.

1.38 Computing Properties of Water and Steam in Python

Water is a very special substance. It is abundant, cheap, hydrating, and great for many engineering applications. Whatever your modeling goal, there is a good change you will require properties of water at various conditions.

There is an international association, IAPWS, which publishes and coordinates some of the best research on the properties of water. There is a special equation of state just for water developed by them that very accurately computes the properties of water, called IAPWS-95. There is also a "shortcut" version called IAPWS-97 which is faster to solve but has reduced accuracy and various discontinuities.

There are quite a few implementations of IAPWS-95 and IAPWS-97 out there. Besides the many commercial implementations, the are the following excellent open source ones:

- iapws by Juan José Gómez Romera, GPL3 licensed, containing IAPWS-95 and IAPWS-97 among other standards. Implemented in Python.
- CoolProp by Ian Bell, MIT licensed and containing IAPWS-95 and IAPWS-97 along with their transport properties. Implemented in C++ with an excellent interface to Python among other languages.
- freesteam by John Pye, GPL3 licensed, containing most of IAPWS-97 and the transport properties. Implemented in C.

There are many more, but these are the best developed libraries that can be used from Python. Water is so common and present in so many calculations that for many applications it is important to make it as fast as possible. IAPWS-95 is conventionally slow; properties are requested at a specified temperature T and pressure P, but the equation of state's input variables are temperature and density! A numerical solver must be used in this case to find the density which yields the specified pressure. This density-solution procedure is normally the slowest part, although computing some properties requires many derivatives that can be slow also.

A good conventional density solver will take ~10-30 s on a modern computer. Only the CPU clockspeed really matters for this calculation time. It was discovered that with the use of Common subexpression elimination, the calculation could be speed up quite a lot. Additionally, if the IAPWS-95 density solution is initialized by the IAPWS-97 explicit calculation (applicable most of the time but not always), a few more iterations can be saved.

The net result of these optimizations is a greatly improved density solve time - normally 2.5-4 s when running with PyPy or Numba. The con to this approach is that the code is nearly unreadable, and it would not be possible to update the coefficients without rewriting the implementation. As IAPWS-95 is a static model which will be the best one available for many years to come, this is an acceptable trade off.

CHAPTER

TWO

INSTALLATION

Get the latest version of chemicals from https://pypi.python.org/pypi/chemicals/

If you have an installation of Python with pip, simple install it with:

```
$ pip install chemicals
```

If you are using conda, you can install chemicals from conda-forge channel:

```
$ conda install -c conda-forge chemicals
```

To get the git version, run:

```
$ git clone git://github.com/CalebBell/chemicals.git
```

CHAPTER

THREE

LATEST SOURCE CODE

The latest development version of chemicals's sources can be obtained at

https://github.com/CalebBell/chemicals

CHAPTER

FOUR

BUG REPORTS

To report bugs, please use the chemicals's Bug Tracker at:

https://github.com/CalebBell/chemicals/issues

CHAPTER

FIVE

LICENSE INFORMATION

See `LICENSE.txt` for information on the terms & conditions for usage of this software, and a DISCLAIMER OF ALL WARRANTIES.

Although not required by the chemicals license, if it is convenient for you, please cite chemicals if used in your work. Please also consider contributing any changes you make back, and benefit the community.

CHAPTER

SIX

CITATION

To cite chemicals in publications use:

```
Caleb Bell, Yoel Rene Cortes-Pena, and Contributors (2016-2023). Chemicals: Chemical␣
↪properties component of Chemical Engineering Design Library (ChEDL)
https://github.com/CalebBell/chemicals.
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Pitzer, K. S., D. Z. Lippmann, R. F. Curl, C. M. Huggins, and D. E. Petersen: The Volumetric and Thermodynamic Properties of Fluids. II. Compressibility Factor, Vapor Pressure and Entropy of Vaporization. J. Am. Chem. Soc., 77: 3433 (1955).

[2] Horstmann, Sven, Anna Jabłoniec, Jörg Krafczyk, Kai Fischer, and Jürgen Gmehling. "PSRK Group Contribution Equation of State: Comprehensive Revision and Extension IV, Including Critical Constants and A-Function Parameters for 1000 Components." Fluid Phase Equilibria 227, no. 2 (January 25, 2005): 157-64. doi:10.1016/j.fluid.2004.11.002.

[3] Passut, Charles A., and Ronald P. Danner. "Acentric Factor. A Valuable Correlating Parameter for the Properties of Hydrocarbons." Industrial & Engineering Chemistry Process Design and Development 12, no. 3 (July 1, 1973): 365-68. doi:10.1021/i260047a026.

[4] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[5] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Halm, Roland L., and Leonard I. Stiel. "A Fourth Parameter for the Vapor Pressure and Entropy of Vaporization of Polar Fluids." AIChE Journal 13, no. 2 (1967): 351-355. doi:10.1002/aic.690130228.

[2] D, Kukoljac Miloš, and Grozdanić Dušan K. "New Values of the Polarity Factor." Journal of the Serbian Chemical Society 65, no. 12 (January 1, 2000). http://www.shd.org.rs/JSCS/Vol65/No12-Pdf/JSCS12-07.pdf

[1] Lee, Byung Ik, and Michael G. Kesler. "A Generalized Thermodynamic Correlation Based on Three-Parameter Corresponding States." AIChE Journal 21, no. 3 (1975): 510-527. doi:10.1002/aic.690210313.

[1] Lemmon, Eric W., Richard T. Jacobsen, Steven G. Penoncello, and Daniel G. Friend. "Thermodynamic Properties of Air and Mixtures of Nitrogen, Argon, and Oxygen From 60 to 2000 K at Pressures to 2000 MPa." Journal of Physical and Chemical Reference Data 29, no. 3 (May 1, 2000): 331-85. https://doi.org/10.1063/1.1285884.

[1] Lemmon, Eric W., Richard T. Jacobsen, Steven G. Penoncello, and Daniel G. Friend. "Thermodynamic Properties of Air and Mixtures of Nitrogen, Argon, and Oxygen From 60 to 2000 K at Pressures to 2000 MPa." Journal of Physical and Chemical Reference Data 29, no. 3 (May 1, 2000): 331-85. https://doi.org/10.1063/1.1285884.

[1] Lemmon, Eric W., Richard T. Jacobsen, Steven G. Penoncello, and Daniel G. Friend. "Thermodynamic Properties of Air and Mixtures of Nitrogen, Argon, and Oxygen From 60 to 2000 K at Pressures to 2000 MPa." Journal of Physical and Chemical Reference Data 29, no. 3 (May 1, 2000): 331-85. https://doi.org/10.1063/1.1285884.

[1] Herrmann, Sebastian, Hans-Joachim Kretzschmar, and Donald P. Gatley. "Thermodynamic Properties of Real Moist Air, Dry Air, Steam, Water, and Ice (RP-1485)." HVAC&R Research 15, no. 5 (September 1, 2009): 961-986. https://doi.org/10.1080/10789669.2009.10390874.

[1] Herrmann, Sebastian, Hans-Joachim Kretzschmar, and Donald P. Gatley. “Thermodynamic Properties of Real Moist Air, Dry Air, Steam, Water, and Ice (RP-1485).” HVAC&R Research 15, no. 5 (September 1, 2009): 961-986. https://doi.org/10.1080/10789669.2009.10390874.

[1] Herrmann, Sebastian, Hans-Joachim Kretzschmar, and Donald P. Gatley. “Thermodynamic Properties of Real Moist Air, Dry Air, Steam, Water, and Ice (RP-1485).” HVAC&R Research 15, no. 5 (September 1, 2009): 961-986. https://doi.org/10.1080/10789669.2009.10390874.

[1] Fernández-Prini, Roberto, Jorge L. Alvarez, and Allan H. Harvey. “Henry’s Constants and Vapor-Liquid Distribution Constants for Gaseous Solutes in H2O and D2O at High Temperatures.” Journal of Physical and Chemical Reference Data 32, no. 2 (June 2003): 903-16. https://doi.org/10.1063/1.1564818.

[1] Fernández-Prini, Roberto, Jorge L. Alvarez, and Allan H. Harvey. “Henry’s Constants and Vapor-Liquid Distribution Constants for Gaseous Solutes in H2O and D2O at High Temperatures.” Journal of Physical and Chemical Reference Data 32, no. 2 (June 2003): 903-16. https://doi.org/10.1063/1.1564818.

[1] Green, D. W. Waste management. In Perry`s Chemical Engineers` Handbook, 9 ed.; McGraw-Hill Education, 2018

[1] Green, D. W. Waste management. In Perry`s Chemical Engineers` Handbook, 9 ed.; McGraw-Hill Education, 2018

[1] Perez, Peter L., and André L. Boehman. “Experimental Investigation of the Autoignition Behavior of Surrogate Gasoline Fuels in a Constant-Volume Combustion Bomb Apparatus and Its Relevance to HCCI Combustion.” Energy & Fuels 26, no. 10 (October 18, 2012): 6106-17. https://doi.org/10.1021/ef300503b.

[1] Perez, Peter L., and André L. Boehman. “Experimental Investigation of the Autoignition Behavior of Surrogate Gasoline Fuels in a Constant-Volume Combustion Bomb Apparatus and Its Relevance to HCCI Combustion.” Energy & Fuels 26, no. 10 (October 18, 2012): 6106-17. https://doi.org/10.1021/ef300503b.

[1] Lehn, Florian vom, Liming Cai, Rupali Tripathi, Rafal Broda, and Heinz Pitsch. “A Property Database of Fuel Compounds with Emphasis on Spark-Ignition Engine Applications.” Applications in Energy and Combustion Science 5 (March 1, 2021): 100018. https://doi.org/10.1016/j.jaecs.2020.100018.

[1] McKinsey. “Octane.” Accessed April 18, 2022. http://www.mckinseyenergyinsights.com/resources/refinery-reference-desk/octane/.

[1] Al Ibrahim, Emad, and Aamir Farooq. “Prediction of the Derived Cetane Number and Carbon/Hydrogen Ratio from Infrared Spectroscopic Data.” Energy & Fuels 35, no. 9 (May 6, 2021): 8141-52. https://doi.org/10.1021/acs.energyfuels.0c03899.

[2] Dahmen, Manuel, and Wolfgang Marquardt. “A Novel Group Contribution Method for the Prediction of the Derived Cetane Number of Oxygenated Hydrocarbons.” Energy & Fuels 29, no. 9 (September 17, 2015): 5781-5801. https://doi.org/10.1021/acs.energyfuels.5b01032.

[1] Lehn, Florian vom, Liming Cai, Rupali Tripathi, Rafal Broda, and Heinz Pitsch. “A Property Database of Fuel Compounds with Emphasis on Spark-Ignition Engine Applications.” Applications in Energy and Combustion Science 5 (March 1, 2021): 100018. https://doi.org/10.1016/j.jaecs.2020.100018.

[2] Kessler, Travis. CombustDB. Python. 2019. UMass Lowell Energy and Combustion Research Laboratory, 2021. https://github.com/ecrl/combustdb.

[3] Lehn, Florian vom, Benedict Brosius, Rafal Broda, Liming Cai, and Heinz Pitsch. “Using Machine Learning with Target-Specific Feature Sets for Structure-Property Relationship Modeling of Octane Numbers and Octane Sensitivity.” Fuel 281 (December 1, 2020): 118772. https://doi.org/10.1016/j.fuel.2020.118772.

[4] Kessler, Travis, and John Hunter Mack. “ECNet: Large Scale Machine Learning Projects for Fuel Property Prediction.” Journal of Open Source Software 2, no. 17 (2017): 401.

[1] Lehn, Florian vom, Liming Cai, Rupali Tripathi, Rafal Broda, and Heinz Pitsch. “A Property Database of Fuel Compounds with Emphasis on Spark-Ignition Engine Applications.” Applications in Energy and Combustion Science 5 (March 1, 2021): 100018. https://doi.org/10.1016/j.jaecs.2020.100018.

[2] Kessler, Travis. CombustDB. Python. 2019. UMass Lowell Energy and Combustion Research Laboratory, 2021. https://github.com/ecrl/combustdb.

[3] Lehn, Florian vom, Benedict Brosius, Rafal Broda, Liming Cai, and Heinz Pitsch. "Using Machine Learning with Target-Specific Feature Sets for Structure-Property Relationship Modeling of Octane Numbers and Octane Sensitivity." Fuel 281 (December 1, 2020): 118772. https://doi.org/10.1016/j.fuel.2020.118772.

[4] Kessler, Travis, and John Hunter Mack. "ECNet: Large Scale Machine Learning Projects for Fuel Property Prediction." Journal of Open Source Software 2, no. 17 (2017): 401.

[1] Dahmen, Manuel, and Wolfgang Marquardt. "A Novel Group Contribution Method for the Prediction of the Derived Cetane Number of Oxygenated Hydrocarbons." Energy & Fuels 29, no. 9 (September 17, 2015): 5781-5801. https://doi.org/10.1021/acs.energyfuels.5b01032.

[1] Ambrose, Douglas, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 1. An Introductory Survey." Journal of Chemical & Engineering Data 41, no. 1 (January 1, 1996): 154-154. doi:10.1021/je950378q.

[2] Ambrose, Douglas, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 2. Normal Alkanes." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 531-46. doi:10.1021/je00019a001.

[3] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 3. Aromatic Hydrocarbons." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 547-58. doi:10.1021/je00019a002.

[4] Gude, Michael, and Amyn S. Teja. "Vapor-Liquid Critical Properties of Elements and Compounds. 4. Aliphatic Alkanols." Journal of Chemical & Engineering Data 40, no. 5 (September 1, 1995): 1025-36. doi:10.1021/je00021a001.

[5] Daubert, Thomas E. "Vapor-Liquid Critical Properties of Elements and Compounds. 5. Branched Alkanes and Cycloalkanes." Journal of Chemical & Engineering Data 41, no. 3 (January 1, 1996): 365-72. doi:10.1021/je9501548.

[6] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 6. Unsaturated Aliphatic Hydrocarbons." Journal of Chemical & Engineering Data 41, no. 4 (January 1, 1996): 645-56. doi:10.1021/je9501999.

[7] Kudchadker, Arvind P., Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 7. Oxygen Compounds Other Than Alkanols and Cycloalkanols." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 457-79. doi:10.1021/je0001680.

[8] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 8. Organic Sulfur, Silicon, and Tin Compounds (C + H + S, Si, and Sn)." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 480-85. doi:10.1021/je000210r.

[9] Marsh, Kenneth N., Colin L. Young, David W. Morton, Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 9. Organic Compounds Containing Nitrogen." Journal of Chemical & Engineering Data 51, no. 2 (March 1, 2006): 305-14. doi:10.1021/je050221q.

[10] Marsh, Kenneth N., Alan Abramson, Douglas Ambrose, David W. Morton, Eugene Nikitin, Constantine Tsonopoulos, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 10. Organic Compounds Containing Halogens." Journal of Chemical & Engineering Data 52, no. 5 (September 1, 2007): 1509-38. doi:10.1021/je700336g.

[11] Ambrose, Douglas, Constantine Tsonopoulos, and Eugene D. Nikitin. "Vapor-Liquid Critical Properties of Elements and Compounds. 11. Organic Compounds Containing B + O; Halogens + N, + O, + O + S, + S, + Si; N + O; and O + S, + Si." Journal of Chemical & Engineering Data 54, no. 3 (March 12, 2009): 669-89. doi:10.1021/je800580z.

[12] Ambrose, Douglas, Constantine Tsonopoulos, Eugene D. Nikitin, David W. Morton, and Kenneth N. Marsh. "Vapor-Liquid Critical Properties of Elements and Compounds. 12. Review of Recent Data for Hydrocarbons and Non-Hydrocarbons." Journal of Chemical & Engineering Data, October 5, 2015, 151005081500002. doi:10.1021/acs.jced.5b00571.

[13] Mathews, Joseph F. "Critical Constants of Inorganic Substances." Chemical Reviews 72, no. 1 (February 1, 1972): 71-100. doi:10.1021/cr60275a004.

[14] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[15] Horstmann, Sven, Anna Jabłoniec, Jörg Krafczyk, Kai Fischer, and Jürgen Gmehling. "PSRK Group Contribution Equation of State: Comprehensive Revision and Extension IV, Including Critical Constants and A-Function Parameters for 1000 Components." Fluid Phase Equilibria 227, no. 2 (January 25, 2005): 157-64. doi:10.1016/j.fluid.2004.11.002.

[16] Passut, Charles A., and Ronald P. Danner. "Acentric Factor. A Valuable Correlating Parameter for the Properties of Hydrocarbons." Industrial & Engineering Chemistry Process Design and Development 12, no. 3 (July 1, 1973): 365-68. doi:10.1021/i260047a026.

[17] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[18] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[19] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[20] Piña-Martinez, Andrés, Romain Privat, and Jean-Noël Jaubert. "Use of 300,000 Pseudo-Experimental Data over 1800 Pure Fluids to Assess the Performance of Four Cubic Equations of State: SRK, PR, Tc-RK, and Tc-PR." AIChE Journal n/a, no. n/a (n.d.): e17518. https://doi.org/10.1002/aic.17518.

[21] Wilson, G. M., and L. V. Jasperson. "Critical Constants Tc, Pc, Estimation Based on Zero, First and Second Order Methods." In Proceedings of the AIChE Spring Meeting, 21, 1996.

[22] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Ambrose, Douglas, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 1. An Introductory Survey." Journal of Chemical & Engineering Data 41, no. 1 (January 1, 1996): 154-154. doi:10.1021/je950378q.

[2] Ambrose, Douglas, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 2. Normal Alkanes." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 531-46. doi:10.1021/je00019a001.

[3] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 3. Aromatic Hydrocarbons." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 547-58. doi:10.1021/je00019a002.

[4] Gude, Michael, and Amyn S. Teja. "Vapor-Liquid Critical Properties of Elements and Compounds. 4. Aliphatic Alkanols." Journal of Chemical & Engineering Data 40, no. 5 (September 1, 1995): 1025-36. doi:10.1021/je00021a001.

[5] Daubert, Thomas E. "Vapor-Liquid Critical Properties of Elements and Compounds. 5. Branched Alkanes and Cycloalkanes." Journal of Chemical & Engineering Data 41, no. 3 (January 1, 1996): 365-72. doi:10.1021/je9501548.

[6] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 6. Unsaturated Aliphatic Hydrocarbons." Journal of Chemical & Engineering Data 41, no. 4 (January 1, 1996): 645-56. doi:10.1021/je9501999.

[7] Kudchadker, Arvind P., Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 7. Oxygen Compounds Other Than Alkanols and Cycloalkanols." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 457-79. doi:10.1021/je0001680.

[8] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 8. Organic Sulfur, Silicon, and Tin Compounds (C + H + S, Si, and Sn)." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 480-85. doi:10.1021/je000210r.

[9] Marsh, Kenneth N., Colin L. Young, David W. Morton, Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 9. Organic Compounds Containing Nitrogen." Journal of Chemical & Engineering Data 51, no. 2 (March 1, 2006): 305-14. doi:10.1021/je050221q.

[10] Marsh, Kenneth N., Alan Abramson, Douglas Ambrose, David W. Morton, Eugene Nikitin, Constantine Tsonopoulos, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 10. Organic Compounds Containing Halogens." Journal of Chemical & Engineering Data 52, no. 5 (September 1, 2007): 1509-38. doi:10.1021/je700336g.

[11] Ambrose, Douglas, Constantine Tsonopoulos, and Eugene D. Nikitin. "Vapor-Liquid Critical Properties of Elements and Compounds. 11. Organic Compounds Containing B + O; Halogens + N, + O, + O + S, + S, + Si; N + O; and O + S, + Si." Journal of Chemical & Engineering Data 54, no. 3 (March 12, 2009): 669-89. doi:10.1021/je800580z.

[12] Ambrose, Douglas, Constantine Tsonopoulos, Eugene D. Nikitin, David W. Morton, and Kenneth N. Marsh. "Vapor-Liquid Critical Properties of Elements and Compounds. 12. Review of Recent Data for Hydrocarbons and Non-Hydrocarbons." Journal of Chemical & Engineering Data, October 5, 2015, 151005081500002. doi:10.1021/acs.jced.5b00571.

[13] Mathews, Joseph F. "Critical Constants of Inorganic Substances." Chemical Reviews 72, no. 1 (February 1, 1972): 71-100. doi:10.1021/cr60275a004.

[14] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[15] Horstmann, Sven, Anna Jabłoniec, Jörg Krafczyk, Kai Fischer, and Jürgen Gmehling. "PSRK Group Contribution Equation of State: Comprehensive Revision and Extension IV, Including Critical Constants and A-Function Parameters for 1000 Components." Fluid Phase Equilibria 227, no. 2 (January 25, 2005): 157-64. doi:10.1016/j.fluid.2004.11.002.

[16] Passut, Charles A., and Ronald P. Danner. "Acentric Factor. A Valuable Correlating Parameter for the Properties of Hydrocarbons." Industrial & Engineering Chemistry Process Design and Development 12, no. 3 (July 1, 1973): 365-68. doi:10.1021/i260047a026.

[17] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[18] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[19] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[20] Piña-Martinez, Andrés, Romain Privat, and Jean-Noël Jaubert. "Use of 300,000 Pseudo-Experimental Data over 1800 Pure Fluids to Assess the Performance of Four Cubic Equations of State: SRK, PR, Tc-RK, and Tc-PR." AIChE Journal n/a, no. n/a (n.d.): e17518. https://doi.org/10.1002/aic.17518.

[21] Wilson, G. M., and L. V. Jasperson. "Critical Constants Tc, Pc, Estimation Based on Zero, First and Second Order Methods." In Proceedings of the AIChE Spring Meeting, 21, 1996.

[22] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Ambrose, Douglas, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 1. An Introductory Survey." Journal of Chemical & Engineering Data 41, no. 1 (January 1, 1996): 154-154. doi:10.1021/je950378q.

[2] Ambrose, Douglas, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 2. Normal Alkanes." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 531-46. doi:10.1021/je00019a001.

[3] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 3. Aromatic Hydrocarbons." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 547-58. doi:10.1021/je00019a002.

[4] Gude, Michael, and Amyn S. Teja. "Vapor-Liquid Critical Properties of Elements and Compounds. 4. Aliphatic Alkanols." Journal of Chemical & Engineering Data 40, no. 5 (September 1, 1995): 1025-36. doi:10.1021/je00021a001.

[5] Daubert, Thomas E. "Vapor-Liquid Critical Properties of Elements and Compounds. 5. Branched Alkanes and Cycloalkanes." Journal of Chemical & Engineering Data 41, no. 3 (January 1, 1996): 365-72. doi:10.1021/je9501548.

[6] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 6. Unsaturated Aliphatic Hydrocarbons." Journal of Chemical & Engineering Data 41, no. 4 (January 1, 1996): 645-56. doi:10.1021/je9501999.

[7] Kudchadker, Arvind P., Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 7. Oxygen Compounds Other Than Alkanols and Cycloalkanols." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 457-79. doi:10.1021/je0001680.

[8] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 8. Organic Sulfur, Silicon, and Tin Compounds (C + H + S, Si, and Sn)." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 480-85. doi:10.1021/je000210r.

[9] Marsh, Kenneth N., Colin L. Young, David W. Morton, Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 9. Organic Compounds Containing Nitrogen." Journal of Chemical & Engineering Data 51, no. 2 (March 1, 2006): 305-14. doi:10.1021/je050221q.

[10] Marsh, Kenneth N., Alan Abramson, Douglas Ambrose, David W. Morton, Eugene Nikitin, Constantine Tsonopoulos, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 10. Organic Compounds Containing Halogens." Journal of Chemical & Engineering Data 52, no. 5 (September 1, 2007): 1509-38. doi:10.1021/je700336g.

[11] Ambrose, Douglas, Constantine Tsonopoulos, and Eugene D. Nikitin. "Vapor-Liquid Critical Properties of Elements and Compounds. 11. Organic Compounds Containing B + O; Halogens + N, + O, + O + S, + S, + Si; N + O; and O + S, + Si." Journal of Chemical & Engineering Data 54, no. 3 (March 12, 2009): 669-89. doi:10.1021/je800580z.

[12] Ambrose, Douglas, Constantine Tsonopoulos, Eugene D. Nikitin, David W. Morton, and Kenneth N. Marsh. "Vapor-Liquid Critical Properties of Elements and Compounds. 12. Review of Recent Data for Hydrocarbons and Non-Hydrocarbons." Journal of Chemical & Engineering Data, October 5, 2015, 151005081500002. doi:10.1021/acs.jced.5b00571.

[13] Mathews, Joseph F. "Critical Constants of Inorganic Substances." Chemical Reviews 72, no. 1 (February 1, 1972): 71-100. doi:10.1021/cr60275a004.

[14] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[15] Horstmann, Sven, Anna Jabłoniec, Jörg Krafczyk, Kai Fischer, and Jürgen Gmehling. "PSRK Group Contribution Equation of State: Comprehensive Revision and Extension IV, Including Critical Constants and A-Function Parameters for 1000 Components." Fluid Phase Equilibria 227, no. 2 (January 25, 2005): 157-64. doi:10.1016/j.fluid.2004.11.002.

[16] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[17] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[18] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[19] Piña-Martinez, Andrés, Romain Privat, and Jean-Noël Jaubert. "Use of 300,000 Pseudo-Experimental Data over 1800 Pure Fluids to Assess the Performance of Four Cubic Equations of State: SRK, PR, Tc-RK, and Tc-PR." AIChE Journal n/a, no. n/a (n.d.): e17518. https://doi.org/10.1002/aic.17518.

[20] Fedors, R. F. "A Method to Estimate Critical Volumes." AIChE Journal 25, no. 1 (1979): 202-202. https://doi.org/10.1002/aic.690250129.

[21] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Mersmann, Alfons, and Matthias Kind. "Correlation for the Prediction of Critical Molar Volume." Industrial & Engineering Chemistry Research, October 16, 2017. https://doi.org/10.1021/acs.iecr.7b03171.

[2] Mersmann, Alfons, and Matthias Kind. "Prediction of Mechanical and Thermal Properties of Pure Liquids, of Critical Data, and of Vapor Pressure." Industrial & Engineering Chemistry Research, January 31, 2017. https://doi.org/10.1021/acs.iecr.6b04323.

[1] Ambrose, Douglas, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 1. An Introductory Survey." Journal of Chemical & Engineering Data 41, no. 1 (January 1, 1996): 154-154. doi:10.1021/je950378q.

[2] Ambrose, Douglas, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 2. Normal Alkanes." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 531-46. doi:10.1021/je00019a001.

[3] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 3. Aromatic Hydrocarbons." Journal of Chemical & Engineering Data 40, no. 3 (May 1, 1995): 547-58. doi:10.1021/je00019a002.

[4] Gude, Michael, and Amyn S. Teja. "Vapor-Liquid Critical Properties of Elements and Compounds. 4. Aliphatic Alkanols." Journal of Chemical & Engineering Data 40, no. 5 (September 1, 1995): 1025-36. doi:10.1021/je00021a001.

[5] Daubert, Thomas E. "Vapor-Liquid Critical Properties of Elements and Compounds. 5. Branched Alkanes and Cycloalkanes." Journal of Chemical & Engineering Data 41, no. 3 (January 1, 1996): 365-72. doi:10.1021/je9501548.

[6] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 6. Unsaturated Aliphatic Hydrocarbons." Journal of Chemical & Engineering Data 41, no. 4 (January 1, 1996): 645-56. doi:10.1021/je9501999.

[7] Kudchadker, Arvind P., Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 7. Oxygen Compounds Other Than Alkanols and Cycloalkanols." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 457-79. doi:10.1021/je0001680.

[8] Tsonopoulos, Constantine, and Douglas Ambrose. "Vapor-Liquid Critical Properties of Elements and Compounds. 8. Organic Sulfur, Silicon, and Tin Compounds (C + H + S, Si, and Sn)." Journal of Chemical & Engineering Data 46, no. 3 (May 1, 2001): 480-85. doi:10.1021/je000210r.

[9] Marsh, Kenneth N., Colin L. Young, David W. Morton, Douglas Ambrose, and Constantine Tsonopoulos. "Vapor-Liquid Critical Properties of Elements and Compounds. 9. Organic Compounds Containing Nitrogen." Journal of Chemical & Engineering Data 51, no. 2 (March 1, 2006): 305-14. doi:10.1021/je050221q.

[10] Marsh, Kenneth N., Alan Abramson, Douglas Ambrose, David W. Morton, Eugene Nikitin, Constantine Tsonopoulos, and Colin L. Young. "Vapor-Liquid Critical Properties of Elements and Compounds. 10. Organic Compounds Containing Halogens." Journal of Chemical & Engineering Data 52, no. 5 (September 1, 2007): 1509-38. doi:10.1021/je700336g.

[11] Ambrose, Douglas, Constantine Tsonopoulos, and Eugene D. Nikitin. "Vapor-Liquid Critical Properties of Elements and Compounds. 11. Organic Compounds Containing B + O; Halogens + N, + O, + O + S, + S, + Si; N + O; and O + S, + Si." Journal of Chemical & Engineering Data 54, no. 3 (March 12, 2009): 669-89. doi:10.1021/je800580z.

[12] Ambrose, Douglas, Constantine Tsonopoulos, Eugene D. Nikitin, David W. Morton, and Kenneth N. Marsh. "Vapor-Liquid Critical Properties of Elements and Compounds. 12. Review of Recent Data for Hydrocarbons and Non-Hydrocarbons." Journal of Chemical & Engineering Data, October 5, 2015, 151005081500002. doi:10.1021/acs.jced.5b00571.

[13] Mathews, Joseph F. "Critical Constants of Inorganic Substances." Chemical Reviews 72, no. 1 (February 1, 1972): 71-100. doi:10.1021/cr60275a004.

[14] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[15] Horstmann, Sven, Anna Jabłoniec, Jörg Krafczyk, Kai Fischer, and Jürgen Gmehling. "PSRK Group Contribution Equation of State: Comprehensive Revision and Extension IV, Including Critical Constants and A-Function Parameters for 1000 Components." Fluid Phase Equilibria 227, no. 2 (January 25, 2005): 157-64. doi:10.1016/j.fluid.2004.11.002.

[16] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[17] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[18] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[19] Piña-Martinez, Andrés, Romain Privat, and Jean-Noël Jaubert. "Use of 300,000 Pseudo-Experimental Data over 1800 Pure Fluids to Assess the Performance of Four Cubic Equations of State: SRK, PR, Tc-RK, and Tc-PR." AIChE Journal n/a, no. n/a (n.d.): e17518. https://doi.org/10.1002/aic.17518.

[20] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Ihmels, E. Christian. "The Critical Surface." Journal of Chemical & Engineering Data 55, no. 9 (September 9, 2010): 3474-80. doi:10.1021/je100167w.

[1] Meissner, H. P., and E. M. Redding. "Prediction of Critical Constants." Industrial & Engineering Chemistry 34, no. 5 (May 1, 1942): 521-26. doi:10.1021/ie50389a003.

[1] Grigoras, Stelian. "A Structural Approach to Calculate Physical Properties of Pure Organic Substances: The Critical Temperature, Critical Volume and Related Properties." Journal of Computational Chemistry 11, no. 4 (May 1, 1990): 493-510. doi:10.1002/jcc.540110408

[1] Hekayati, Javad, and Sona Raeissi. "Estimation of the Critical Properties of Compounds Using Volume-Based Thermodynamics." AIChE Journal n/a, no. n/a (n.d.): e17004. https://doi.org/10.1002/aic.17004.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Li, C. C. "Critical Temperature Estimation for Simple Mixtures." The Canadian Journal of Chemical Engineering 49, no. 5 (October 1, 1971): 709-10. doi:10.1002/cjce.5450490529.

[1] Chueh, P. L., and J. M. Prausnitz. "Vapor-Liquid Equilibria at High Pressures: Calculation of Critical Temperatures, Volumes, and Pressures of Nonpolar Mixtures." AIChE Journal 13, no. 6 (November 1, 1967): 1107-13. doi:10.1002/aic.690130613.

[2] Najafi, Hamidreza, Babak Maghbooli, and Mohammad Amin Sobati. "Prediction of True Critical Temperature of Multi-Component Mixtures: Extending Fast Estimation Methods." Fluid Phase Equilibria 392 (April 25, 2015): 104-26. doi:10.1016/j.fluid.2015.02.001.

[1] Grieves, Robert B., and George Thodos. "The Critical Temperatures of Multicomponent Hydrocarbon Systems." AIChE Journal 8, no. 4 (September 1, 1962): 550-53. doi:10.1002/aic.690080426.

[2] Najafi, Hamidreza, Babak Maghbooli, and Mohammad Amin Sobati. "Prediction of True Critical Temperature of Multi-Component Mixtures: Extending Fast Estimation Methods." Fluid Phase Equilibria 392 (April 25, 2015): 104-26. doi:10.1016/j.fluid.2015.02.001.

[1] Teja, Amyn S., Kul B. Garg, and Richard L. Smith. "A Method for the Calculation of Gas-Liquid Critical Temperatures and Pressures of Multicomponent Mixtures." Industrial & Engineering Chemistry Process Design and Development 22, no. 4 (1983): 672-76.

[2] Najafi, Hamidreza, Babak Maghbooli, and Mohammad Amin Sobati. "Prediction of True Critical Temperature of Multi-Component Mixtures: Extending Fast Estimation Methods." Fluid Phase Equilibria 392 (April 25, 2015): 104-26. doi:10.1016/j.fluid.2015.02.001.

[1] Chueh, P. L., and J. M. Prausnitz. "Vapor-Liquid Equilibria at High Pressures: Calculation of Critical Temperatures, Volumes, and Pressures of Nonpolar Mixtures." AIChE Journal 13, no. 6 (November 1, 1967): 1107-13. doi:10.1002/aic.690130613.

[2] Najafi, Hamidreza, Babak Maghbooli, and Mohammad Amin Sobati. "Prediction of True Critical Volume of Multi-Component Mixtures: Extending Fast Estimation Methods." Fluid Phase Equilibria 386 (January 25, 2015): 13-29. doi:10.1016/j.fluid.2014.11.008.

[1] Teja, Amyn S., Kul B. Garg, and Richard L. Smith. "A Method for the Calculation of Gas-Liquid Critical Temperatures and Pressures of Multicomponent Mixtures." Industrial & Engineering Chemistry Process Design and Development 22, no. 4 (1983): 672-76.

[2] Najafi, Hamidreza, Babak Maghbooli, and Mohammad Amin Sobati. "Prediction of True Critical Temperature of Multi-Component Mixtures: Extending Fast Estimation Methods." Fluid Phase Equilibria 392 (April 25, 2015): 104-26. doi:10.1016/j.fluid.2015.02.001.

[1] NIST Computational Chemistry Comparison and Benchmark Database NIST Standard Reference Database Number 101 Release 17b, September 2015, Editor: Russell D. Johnson III http://cccbdb.nist.gov/

[2] Muller, Karsten, Liudmila Mokrushina, and Wolfgang Arlt. "Second- Order Group Contribution Method for the Determination of the Dipole Moment." Journal of Chemical & Engineering Data 57, no. 4 (April 12, 2012): 1231-36. doi:10.1021/je2013395.

[3] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[4] Turney, Justin M., Andrew C. Simmonett, Robert M. Parrish, Edward G. Hohenstein, Francesco A. Evangelista, Justin T. Fermann, Benjamin J. Mintz, et al. "Psi4: An Open-Source Ab Initio Electronic Structure Program." WIREs Computational Molecular Science 2, no. 4 (2012): 556-65. https://doi.org/10.1002/wcms.93.

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[2] Aly, Fouad A., and Lloyd L. Lee. "Self-Consistent Equations for Calculating the Ideal Gas Heat Capacity, Enthalpy, and Entropy." Fluid Phase Equilibria 6, no. 3 (January 1, 1981): 169-79. doi:10.1016/0378-3812(81)85002-9.

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] Design Institute for Physical Properties, 1996. DIPPR Project 801 DIPPR/AIChE

[1] N M O'Boyle, M Banck, C A James, C Morley, T Vandermeersch, and G R Hutchison. "Open Babel: An open chemical toolbox." J. Cheminf. (2011), 3, 33. DOI:10.1186/1758-2946-3-33

[1] Hill, Edwin A.""ON A SYSTEM OF INDEXING CHEMICAL LITERATURE; ADOPTED BY THE CLASSIFICATION DIVISION OF THE U. S. PATENT OFFICE.1." Journal of the American Chemical Society 22, no. 8 (August 1, 1900): 478-94. doi:10.1021/ja02046a005.

[1] RDKit: Open-source cheminformatics; http://www.rdkit.org

[1] Laštovka, Václav, Nasser Sallamie, and John M. Shaw. "A Similarity Variable for Estimating the Heat Capacity of Solid Organic Compounds: Part I. Fundamentals." Fluid Phase Equilibria 268, no. 1-2 (June 25, 2008): 51-60. doi:10.1016/j.fluid.2008.03.019.

[1] Brown, William H., and Thomas Poon. Introduction to Organic Chemistry. 4th edition. Hoboken, NJ: Wiley, 2010.

[1] RDKit: Open-source cheminformatics; http://www.rdkit.org

[1] RDKit: Open-source cheminformatics; http://www.rdkit.org

[1] IPCC. "2.10.2 Direct Global Warming Potentials - AR4 WGI Chapter 2: Changes in Atmospheric Constituents and in Radiative Forcing." 2007. https://www.ipcc.ch/publications_and_data/ar4/wg1/en/ch2s2-10-2.html.

[2] IPCC. "Climate Change 2013: The Physical Science Basis. - AR5 WGI Chapter 8: Anthropogenic and Natural Radiative Forcing." 2013. https://www.ipcc.ch/site/assets/uploads/2018/02/WG1AR5_Chapter08_FINAL.pdf

[3] Hodnebrog, Ø., B. Aamaas, J. S. Fuglestvedt, G. Marston, G. Myhre, C. J. Nielsen, M. Sandstad, K. P. Shine, and T. J. Wallington. "Updated Global Warming Potentials and Radiative Efficiencies of Halocarbons and Other Weak Atmospheric Absorbers." Reviews of Geophysics 58, no. 3 (2020): e2019RG000691. https://doi.org/10.1029/2019RG000691.

[1] US EPA, OAR. "Ozone-Depleting Substances." Accessed April 26, 2016. https://www.epa.gov/ozone-layer-protection/ozone-depleting-substances.

[2] WMO (World Meteorological Organization), 2011: Scientific Assessment of Ozone Depletion: 2010. Global Ozone Research and Monitoring Project-Report No. 52, Geneva, Switzerland, 516 p. https://www.wmo.int/pages/prog/arep/gaw/ozone_2010/documents/Ozone-Assessment-2010-complete.pdf

[1] Syrres. 2006. KOWWIN Data, SrcKowData2.zip. http://esc.syrres.com/interkow/Download/SrcKowData2.zip

[2] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[1] Kandula, Vamshi Krishna, John C. Telotte, and F. Carl Knopf. "It`s Not as Easy as It Looks: Revisiting Peng—Robinson Equation of State Convergence Issues for Dew Point, Bubble Point and Flash Calculations." International Journal of Mechanical Engineering Education 41, no. 3 (July 1, 2013): 188-202. https://doi.org/10.7227/IJMEE.41.3.2.

[1] Gmehling, Jurgen, Barbel Kolbe, Michael Kleiber, and Jurgen Rarey. Chemical Thermodynamics for Process Simulation. 1st edition. Weinheim: Wiley-VCH, 2012.

[2] Skogestad, Sigurd. Chemical and Energy Process Engineering. 1st edition. Boca Raton, FL: CRC Press, 2008.

[1] Wilson, Grant M. "A Modified Redlich-Kwong Equation of State, Application to General Physical Data Calculations." In 65th National AIChE Meeting, Cleveland, OH, 1969.

[2] Peng, Ding-Yu, and Donald B. Robinson. "Two and Three Phase Equilibrium Calculations for Systems Containing Water." The Canadian Journal of Chemical Engineering, December 1, 1976. https://doi.org/10.1002/cjce.5450540620.

[1] Peng, Ding-Yu, and Donald B. Robinson. "Two and Three Phase Equilibrium Calculations for Systems Containing Water." The Canadian Journal of Chemical Engineering, December 1, 1976. https://doi.org/10.1002/cjce.5450540620.

[1] Kabo, G. J., and G. N. Roganov. Thermodynamics of Organic Compounds in the Gas State, Volume II: V. 2. College Station, Tex: CRC Press, 1994.

[1] Kabo, G. J., and G. N. Roganov. Thermodynamics of Organic Compounds in the Gas State, Volume II: V. 2. College Station, Tex: CRC Press, 1994.

[1] Kabo, G. J., and G. N. Roganov. Thermodynamics of Organic Compounds in the Gas State, Volume II: V. 2. College Station, Tex: CRC Press, 1994.

[1] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/TDE_Help/Eqns-Pure-Cp0/PPDS2Cp0.htm.

[1] Lastovka, Vaclav, and John M. Shaw. "Predictive Correlations for Ideal Gas Heat Capacities of Pure Hydrocarbons and Petroleum Fractions." Fluid Phase Equilibria 356 (October 25, 2013): 338-370. doi:10.1016/j.fluid.2013.07.023.

[1] Lastovka, Vaclav, and John M. Shaw. "Predictive Correlations for Ideal Gas Heat Capacities of Pure Hydrocarbons and Petroleum Fractions." Fluid Phase Equilibria 356 (October 25, 2013): 338-370. doi:10.1016/j.fluid.2013.07.023.

[1] Lastovka, Vaclav, and John M. Shaw. "Predictive Correlations for Ideal Gas Heat Capacities of Pure Hydrocarbons and Petroleum Fractions." Fluid Phase Equilibria 356 (October 25, 2013): 338-370. doi:10.1016/j.fluid.2013.07.023.

[1] Lastovka, Vaclav, and John M. Shaw. "Predictive Correlations for Ideal Gas Heat Capacities of Pure Hydrocarbons and Petroleum Fractions." Fluid Phase Equilibria 356 (October 25, 2013): 338-370. doi:10.1016/j.fluid.2013.07.023.

[1] Lastovka, Vaclav, and John M. Shaw. "Predictive Correlations for Ideal Gas Heat Capacities of Pure Hydrocarbons and Petroleum Fractions." Fluid Phase Equilibria 356 (October 25, 2013): 338-370. doi:10.1016/j.fluid.2013.07.023.

[1] Lastovka, Vaclav, and John M. Shaw. "Predictive Correlations for Ideal Gas Heat Capacities of Pure Hydrocarbons and Petroleum Fractions." Fluid Phase Equilibria 356 (October 25, 2013): 338-370. doi:10.1016/j.fluid.2013.07.023.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-CsatL/PPDS15-Csat.htm.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-CsatL/CSExpansion.htm

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[3] J.S. Rowlinson, Liquids and Liquid Mixtures, 2nd Ed., Butterworth, London (1969).

[1] Dadgostar, Nafiseh, and John M. Shaw. "A Predictive Correlation for the Constant-Pressure Specific Heat Capacity of Pure and Ill-Defined Liquid Hydrocarbons." Fluid Phase Equilibria 313 (January 15, 2012): 211-226. doi:10.1016/j.fluid.2011.09.015.

[1] Dadgostar, Nafiseh, and John M. Shaw. "A Predictive Correlation for the Constant-Pressure Specific Heat Capacity of Pure and Ill-Defined Liquid Hydrocarbons." Fluid Phase Equilibria 313 (January 15, 2012): 211-226. doi:10.1016/j.fluid.2011.09.015.

[1] Dadgostar, Nafiseh, and John M. Shaw. "A Predictive Correlation for the Constant-Pressure Specific Heat Capacity of Pure and Ill-Defined Liquid Hydrocarbons." Fluid Phase Equilibria 313 (January 15, 2012): 211-226. doi:10.1016/j.fluid.2011.09.015.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Laštovka, Václav, Michal Fulem, Mildred Becerra, and John M. Shaw. "A Similarity Variable for Estimating the Heat Capacity of Solid Organic Compounds: Part II. Application: Heat Capacity Calculation for Ill-Defined Organic Solids." Fluid Phase Equilibria 268, no. 1-2 (June 25, 2008): 134-41. doi:10.1016/j.fluid.2008.03.018.

[1] Laštovka, Václav, Michal Fulem, Mildred Becerra, and John M. Shaw. "A Similarity Variable for Estimating the Heat Capacity of Solid Organic Compounds: Part II. Application: Heat Capacity Calculation for Ill-Defined Organic Solids." Fluid Phase Equilibria 268, no. 1-2 (June 25, 2008): 134-41. doi:10.1016/j.fluid.2008.03.018.

[1] Laštovka, Václav, Michal Fulem, Mildred Becerra, and John M. Shaw. "A Similarity Variable for Estimating the Heat Capacity of Solid Organic Compounds: Part II. Application: Heat Capacity Calculation for Ill-Defined Organic Solids." Fluid Phase Equilibria 268, no. 1-2 (June 25, 2008): 134-41. doi:10.1016/j.fluid.2008.03.018.

[1] Kabo, G. J., and G. N. Roganov. Thermodynamics of Organic Compounds in the Gas State, Volume II: V. 2. College Station, Tex: CRC Press, 1994.

[2] Zabransky, M., V. Ruzicka Jr, V. Majer, and Eugene S. Domalski. Heat Capacity of Liquids: Critical Review and Recommended Values. 2 Volume Set. Washington, D.C.: Amer Inst of Physics, 1996.

[3] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[4] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[5] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Wagner, Wolfgang, and Andreas Pruß. "The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use." Journal of Physical and Chemical Reference Data 31, no. 2 (2002): 387-535.

[1] Wagner, Wolfgang, and Andreas Pruß. "The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use." Journal of Physical and Chemical Reference Data 31, no. 2 (2002): 387-535.

[1] Wagner, Wolfgang, and Andreas Pruß. "The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use." Journal of Physical and Chemical Reference Data 31, no. 2 (2002): 387-535.

[1] Cooper, JR, and RB Dooley. "Revised Release on the IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam." The International Association for the Properties of Water and Steam 1 (2007): 48.

[1] Cooper, JR, and RB Dooley. "Revised Release on the IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam." The International Association for the Properties of Water and Steam 1 (2007): 48.

[1] Cooper, JR, and RB Dooley. "Revised Release on the IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam." The International Association for the Properties of Water and Steam 1 (2007): 48.

[1] Wagner, Wolfgang, and Andreas Pruß. "The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use." Journal of Physical and Chemical Reference Data 31, no. 2 (2002): 387-535.

[1] IAPWS, Secretariat, B Dooley, and EPRI. "Revised Supplementary Release on Saturation Properties of Ordinary Water Substance", 1992.

[2] Wagner, Wolfgang, and A. Pruss. "International Equations for the Saturation Properties of Ordinary Water Substance. Revised According to the International Temperature Scale of 1990. Addendum to J. Phys. Chem. Ref.

Data 16, 893 (1987)." Journal of Physical and Chemical Reference Data 22, no. 3 (May 1, 1993): 783-87. https://doi.org/10.1063/1.555926.

[1] IAPWS, Secretariat, B Dooley, and EPRI. "Revised Supplementary Release on Saturation Properties of Ordinary Water Substance", 1992.

[2] Wagner, Wolfgang, and A. Pruss. "International Equations for the Saturation Properties of Ordinary Water Substance. Revised According to the International Temperature Scale of 1990. Addendum to J. Phys. Chem. Ref. Data 16, 893 (1987)." Journal of Physical and Chemical Reference Data 22, no. 3 (May 1, 1993): 783-87. https://doi.org/10.1063/1.555926.

[1] Brock, James R., and R. Byron Bird. "Surface Tension and the Principle of Corresponding States." AIChE Journal 1, no. 2 (June 1, 1955): 174-77. doi:10.1002/aic.690010208

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Curl, R. F., and Kenneth Pitzer. "Volumetric and Thermodynamic Properties of Fluids-Enthalpy, Free Energy, and Entropy." Industrial & Engineering Chemistry 50, no. 2 (February 1, 1958): 265-74. doi:10.1021/ie50578a047

[3] Pitzer, K. S.: Thermodynamics, 3d ed., New York, McGraw-Hill, 1995, p. 521.

[1] Sastri, S. R. S., and K. K. Rao. "A Simple Method to Predict Surface Tension of Organic Liquids." The Chemical Engineering Journal and the Biochemical Engineering Journal 59, no. 2 (October 1995): 181-86. doi:10.1016/0923-0467(94)02946-6.

[1] Zuo, You-Xiang, and Erling H. Stenby. "Corresponding-States and Parachor Models for the Calculation of Interfacial Tensions." The Canadian Journal of Chemical Engineering 75, no. 6 (December 1, 1997): 1130-37. doi:10.1002/cjce.5450750617

[1] Hakim, D. I., David Steinberg, and L. I. Stiel. "Generalized Relationship for the Surface Tension of Polar Fluids." Industrial & Engineering Chemistry Fundamentals 10, no. 1 (February 1, 1971): 174-75. doi:10.1021/i160037a032.

[1] Miqueu, C, D Broseta, J Satherley, B Mendiboure, J Lachaise, and A Graciaa. "An Extended Scaled Equation for the Temperature Dependence of the Surface Tension of Pure Compounds Inferred from an Analysis of Experimental Data." Fluid Phase Equilibria 172, no. 2 (July 5, 2000): 169-82. doi:10.1016/S0378-3812(00)00384-8.

[1] Aleem, W., N. Mellon, S. Sufian, M. I. A. Mutalib, and D. Subbarao. "A Model for the Estimation of Surface Tension of Pure Hydrocarbon Liquids." Petroleum Science and Technology 33, no. 23-24 (December 17, 2015): 1908-15. doi:10.1080/10916466.2015.1110593.

[1] Mersmann, Alfons, and Matthias Kind. "Prediction of Mechanical and Thermal Properties of Pure Liquids, of Critical Data, and of Vapor Pressure." Industrial & Engineering Chemistry Research, January 31, 2017. https://doi.org/10.1021/acs.iecr.6b04323.

[1] Gharagheizi, Farhad, Ali Eslamimanesh, Mehdi Sattari, Amir H. Mohammadi, and Dominique Richon. "Development of Corresponding States Model for Estimation of the Surface Tension of Chemical Compounds." AIChE Journal 59, no. 2 (2013): 613-21. https://doi.org/10.1002/aic.13824.

[1] Gharagheizi, Farhad, Ali Eslamimanesh, Mehdi Sattari, Amir H. Mohammadi, and Dominique Richon. "Development of Corresponding States Model for Estimation of the Surface Tension of Chemical Compounds." AIChE Journal 59, no. 2 (2013): 613-21. https://doi.org/10.1002/aic.13824.

[1] Winterfeld, P. H., L. E. Scriven, and H. T. Davis. "An Approximate Theory of Interfacial Tensions of Multicomponent Systems: Applications to Binary Liquid-Vapor Tensions." AIChE Journal 24, no. 6 (November 1, 1978): 1010-14. doi:10.1002/aic.690240610.

[2] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] Weinaug, Charles F., and Donald L. Katz. "Surface Tensions of Methane-Propane Mixtures." Industrial & Engineering Chemistry 35, no. 2 (February 1, 1943): 239-246. https://doi.org/10.1021/ie50398a028.

[2] Pedersen, Karen Schou, Aage Fredenslund, and Per Thomassen. Properties of Oils and Natural Gases. Vol. 5. Gulf Pub Co, 1989.

[1] Diguilio, Ralph, and Amyn S. Teja. "Correlation and Prediction of the Surface Tensions of Mixtures." The Chemical Engineering Journal 38, no. 3 (July 1988): 205-8. doi:10.1016/0300-9467(88)80079-0.

[1] IAPWS. 2014. Revised Release on Surface Tension of Ordinary Water Substance

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Kalantari Meybodi, Mahdi, Amin Daryasafar, and Masoud Karimi. "Determination of Hydrocarbon-Water Interfacial Tension Using a New Empirical Correlation." Fluid Phase Equilibria 415 (May 15, 2016): 42-50. doi:10.1016/j.fluid.2016.01.037.

[1] Diky, Vladimir, Robert D. Chirico, Chris D. Muzny, Andrei F. Kazakov, Kenneth Kroenlein, Joseph W. Magee, Ilmutdin Abdulagatov, and Michael Frenkel. "ThermoData Engine (TDE): Software Implementation of the Dynamic Data Evaluation Concept." Journal of Chemical Information and Modeling 53, no. 12 (2013): 3418-30. doi:10.1021/ci4005699.

[1] Somayajulu, G. R. "A Generalized Equation for Surface Tension from the Triple Point to the Critical Point." International Journal of Thermophysics 9, no. 4 (July 1988): 559-66. doi:10.1007/BF00503154.

[1] Jasper, Joseph J. "The Surface Tension of Pure Liquid Compounds." Journal of Physical and Chemical Reference Data 1, no. 4 (October 1, 1972): 841-1010. doi:10.1063/1.3253106.

[2] Speight, James. Lange's Handbook of Chemistry. 16 edition. McGraw-Hill Professional, 2005.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-SurfaceTension/PPDS14.htm.

[2] Frenkel, Michael, Robert D. Chirico, Vladimir Diky, Xinjian Yan, Qian Dong, and Chris Muzny. "ThermoData Engine (TDE): Software Implementation of the Dynamic Data Evaluation Concept." Journal of Chemical Information and Modeling 45, no. 4 (July 1, 2005): 816-38. https://doi.org/10.1021/ci050067b.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-SurfaceTension/HVPExpansion-SurfaceTension.htm

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-SurfaceTension/ISTExpansion-SurfaceTension.htm

[1] Somayajulu, G. R. "A Generalized Equation for Surface Tension from the Triple Point to the Critical Point." International Journal of Thermophysics 9, no. 4 (July 1988): 559-66. doi:10.1007/BF00503154.

[2] Mulero, A., M. I. Parra, and I. Cachadina. "The Somayajulu Correlation for the Surface Tension Revisited." Fluid Phase Equilibria 339 (February 15, 2013): 81-88. doi:10.1016/j.fluid.2012.11.038.

[3] Jasper, Joseph J. "The Surface Tension of Pure Liquid Compounds." Journal of Physical and Chemical Reference Data 1, no. 4 (October 1, 1972): 841-1010. doi:10.1063/1.3253106.

[4] Speight, James. Lange's Handbook of Chemistry. 16 edition. McGraw-Hill Professional, 2005.

[5] Mulero, A., I. Cachadiña, and M. I. Parra. "Recommended Correlations for the Surface Tension of Common Fluids." Journal of Physical and Chemical Reference Data 41, no. 4 (December 1, 2012): 043105. doi:10.1063/1.4768782.

[6] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[2] Magalhães, Ana L., Patrícia F. Lito, Francisco A. Da Silva, and Carlos M. Silva. "Simple and Accurate Correlations for Diffusion Coefficients of Solutes in Liquids and Supercritical Fluids over Wide Ranges of Temperature and Density." The Journal of Supercritical Fluids 76 (April 2013): 94-114. doi:10.1016/j.supflu.2013.02.002.

[1] Flynn, L.W., M.S. thesis, Northwestern Univ., Evanston, Ill. (1960).

[2] Stiel, L. I., and George Thodos. "Lennard-Jones Force Constants Predicted from Critical Properties." Journal of Chemical & Engineering Data 7, no. 2 (April 1, 1962): 234-36. doi:10.1021/je60013a023

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Stiel, L. I., and George Thodos. "Lennard-Jones Force Constants Predicted from Critical Properties." Journal of Chemical & Engineering Data 7, no. 2 (April 1, 1962): 234-36. doi:10.1021/je60013a023

[1] Tee, L. S., Sukehiro Gotoh, and W. E. Stewart. "Molecular Parameters for Normal Fluids. Lennard-Jones 12-6 Potential." Industrial & Engineering Chemistry Fundamentals 5, no. 3 (August 1, 1966): 356-63. doi:10.1021/i160019a011

[1] Tee, L. S., Sukehiro Gotoh, and W. E. Stewart. "Molecular Parameters for Normal Fluids. Lennard-Jones 12-6 Potential." Industrial & Engineering Chemistry Fundamentals 5, no. 3 (August 1, 1966): 356-63. doi:10.1021/i160019a011

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[2] Magalhães, Ana L., Patrícia F. Lito, Francisco A. Da Silva, and Carlos M. Silva. "Simple and Accurate Correlations for Diffusion Coefficients of Solutes in Liquids and Supercritical Fluids over Wide Ranges of Temperature and Density." The Journal of Supercritical Fluids 76 (April 2013): 94-114. doi:10.1016/j.supflu.2013.02.002.

[1] Flynn, L.W., M.S. thesis, Northwestern Univ., Evanston, Ill. (1960).

[2] Stiel, L. I., and George Thodos. "Lennard-Jones Force Constants Predicted from Critical Properties." Journal of Chemical & Engineering Data 7, no. 2 (April 1, 1962): 234-36. doi:10.1021/je60013a023

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Stiel, L. I., and George Thodos. "Lennard-Jones Force Constants Predicted from Critical Properties." Journal of Chemical & Engineering Data 7, no. 2 (April 1, 1962): 234-36. doi:10.1021/je60013a023

[1] Tee, L. S., Sukehiro Gotoh, and W. E. Stewart. "Molecular Parameters for Normal Fluids. Lennard-Jones 12-6 Potential." Industrial & Engineering Chemistry Fundamentals 5, no. 3 (August 1, 1966): 356-63. doi:10.1021/i160019a011

[1] Tee, L. S., Sukehiro Gotoh, and W. E. Stewart. "Molecular Parameters for Normal Fluids. Lennard-Jones 12-6 Potential." Industrial & Engineering Chemistry Fundamentals 5, no. 3 (August 1, 1966): 356-63. doi:10.1021/i160019a011

[1] Silva, Carlos M., Hongqin Liu, and Eugenia A. Macedo. "Models for Self-Diffusion Coefficients of Dense Fluids, Including Hydrogen-Bonding Substances." Chemical Engineering Science 53, no. 13 (July 1, 1998): 2423-29. doi:10.1016/S0009-2509(98)00037-2

[1] Bird, R. Byron, Warren E. Stewart, and Edwin N. Lightfoot. Transport Phenomena, Revised 2nd Edition. New York: John Wiley & Sons, Inc., 2006

[1] Gesellschaft, VDI, ed. VDI Heat Atlas. 2E. Berlin : Springer, 2010.

[1] Turney, Justin M., Andrew C. Simmonett, Robert M. Parrish, Edward G. Hohenstein, Francesco A. Evangelista, Justin T. Fermann, Benjamin J. Mintz, et al. "Psi4: An Open-Source Ab Initio Electronic Structure Program." WIREs Computational Molecular Science 2, no. 4 (2012): 556-65. https://doi.org/10.1002/wcms.93.

[2] Kooijman, Harry A., and Ross Taylor. The ChemSep Book. Books on Demand Norderstedt, Germany, 2000.

[1] Turney, Justin M., Andrew C. Simmonett, Robert M. Parrish, Edward G. Hohenstein, Francesco A. Evangelista, Justin T. Fermann, Benjamin J. Mintz, et al. "Psi4: An Open-Source Ab Initio Electronic Structure Program." WIREs Computational Molecular Science 2, no. 4 (2012): 556-65. https://doi.org/10.1002/wcms.93.

[1] IAPWS. 1997. Release on the Static Dielectric Constant of Ordinary Water Substance for Temperatures from 238 K to 873 K and Pressures up to 1000 MPa.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[2] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[3] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[4] "CAS Common Chemistry". https://commonchemistry.cas.org/.

[5] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[6] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[7] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Bradley, Jean-Claude, Antony Williams, and Andrew Lang. "Jean-Claude Bradley Open Melting Point Dataset", May 20, 2014. https://figshare.com/articles/Jean_Claude_Bradley_Open_Melting_Point_Datset/1031637.

[2] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[3] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[4] "CAS Common Chemistry". https://commonchemistry.cas.org/.

[5] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[6] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[2] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[3] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[4] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Riedel, L. "Eine Neue Universelle Dampfdruckformel Untersuchungen Uber Eine Erweiterung Des Theorems Der Ubereinstimmenden Zustande. Teil I." Chemie Ingenieur Technik 26, no. 2 (February 1, 1954): 83-89. doi:10.1002/cite.330260206.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Chen, N. H. "Generalized Correlation for Latent Heat of Vaporization." Journal of Chemical & Engineering Data 10, no. 2 (April 1, 1965): 207-10. doi:10.1021/je60025a047

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] LIU, ZHI-YONG. "Estimation of Heat of Vaporization of Pure Liquid at Its Normal Boiling Temperature." Chemical Engineering Communications 184, no. 1 (February 1, 2001): 221-28. doi:10.1080/00986440108912849.

[1] Vetere, Alessandro. "Methods to Predict the Vaporization Enthalpies at the Normal Boiling Temperature of Pure Compounds Revisited." Fluid Phase Equilibria 106, no. 1-2 (May 1, 1995): 1-10. doi:10.1016/0378-3812(94)02627-D.

[2] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Pitzer, Kenneth S. "The Volumetric and Thermodynamic Properties of Fluids. I. Theoretical Basis and Virial Coefficients." Journal of the American Chemical Society 77, no. 13 (July 1, 1955): 3427-33. doi:10.1021/ja01618a001

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Sivaraman, Alwarappa, Joe W. Magee, and Riki Kobayashi. "Generalized Correlation of Latent Heats of Vaporization of Coal-Liquid Model Compounds between Their Freezing Points and Critical Points." Industrial & Engineering Chemistry Fundamentals 23, no. 1 (February 1, 1984): 97-100. doi:10.1021/i100013a017.

[1] Morgan, David L., and Riki Kobayashi. "Extension of Pitzer CSP Models for Vapor Pressures and Heats of Vaporization to Long-Chain Hydrocarbons." Fluid Phase Equilibria 94 (March 15, 1994): 51-87. doi:10.1016/0378-3812(94)87051-9.

[1] Velasco, S., M. J. Santos, and J. A. White. "Extended Corresponding States Expressions for the Changes in Enthalpy, Compressibility Factor and Constant-Volume Heat Capacity at Vaporization." The Journal of Chemical Thermodynamics 85 (June 2015): 68-76. doi:10.1016/j.jct.2015.01.011.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Watson, KM. "Thermodynamics of the Liquid State." Industrial & Engineering Chemistry 35, no. 4 (1943): 398-406.

[2] Martin, Joseph J., and John B. Edwards. "Correlation of Latent Heats of Vaporization." AIChE Journal 11, no. 2 (1965): 331-33. https://doi.org/10.1002/aic.690110226.

[1] Alibakhshi, Amin. "Enthalpy of Vaporization, Its Temperature Dependence and Correlation with Surface Tension: A Theoretical Approach." Fluid Phase Equilibria 432 (January 25, 2017): 62-69. https://doi.org/10.1016/j.fluid.2016.10.013.

[1] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[2] "Enthalpy of Vaporization: PPDS12." https://trc.nist.gov/TDE/TDE_Help/Eqns-Pure-Hvap/PPDS12.htm.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, 8E. McGraw-Hill Professional, 2007.

[2] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[3] Alibakhshi, Amin. "Enthalpy of Vaporization, Its Temperature Dependence and Correlation with Surface Tension: A Theoretical Approach." Fluid Phase Equilibria 432 (January 25, 2017): 62-69. https://doi.org/10.1016/j.fluid.2016.10.013.

[1] Rachford, H. H. Jr, and J. D. Rice. "Procedure for Use of Electronic Digital Computers in Calculating Flash Vaporization Hydrocarbon Equilibrium." Journal of Petroleum Technology 4, no. 10 (October 1, 1952): 19-3. doi:10.2118/952327-G.

[2] Li, Yinghui, Russell T. Johns, and Kaveh Ahmadi. "A Rapid and Robust Alternative to Rachford-Rice in Flash Calculations." Fluid Phase Equilibria 316 (February 25, 2012): 85-97. doi:10.1016/j.fluid.2011.12.005.

[3] Whitson, Curtis H., and Michael L. Michelsen. "The Negative Flash." Fluid Phase Equilibria, Proceedings of the Fifth International Conference, 53 (December 1, 1989): 51-71. doi:10.1016/0378-3812(89)80072-X.

[1] Leibovici, Claude F., and Dan Vladimir Nichita. "Iterative Solutions for iai-ci=1 Equations." Chemical Engineering Research and Design 88, no. 5 (May 1, 2010): 602-5. https://doi.org/10.1016/j.cherd.2009.10.012.

[2] Li, Yinghui, Russell T. Johns, and Kaveh Ahmadi. "A Rapid and Robust Alternative to Rachford-Rice in Flash Calculations." Fluid Phase Equilibria 316 (February 25, 2012): 85-97. doi:10.1016/j.fluid.2011.12.005.

[3] Billingsley, D. S. "Iterative Solution for iai-ci Equations." Computers & Chemical Engineering 26, no. 3 (March 15, 2002): 457-60. https://doi.org/10.1016/S0098-1354(01)00767-0.

[1] Li, Yinghui, Russell T. Johns, and Kaveh Ahmadi. "A Rapid and Robust Alternative to Rachford-Rice in Flash Calculations." Fluid Phase Equilibria 316 (February 25, 2012): 85-97. doi:10.1016/j.fluid.2011.12.005.

[1] Leibovici, ClaudeF., and Jean Neoschil. "A New Look at the Rachford-Rice Equation." Fluid Phase Equilibria 74 (July 15, 1992): 303-8. https://doi.org/10.1016/0378-3812(92)85069-K.

[1] Weigle, Brett D. "A Generalized Polynomial Form of the Objective Function in Flash Calculations." Pennsylvania State University, 1992.

[2] Li, Yinghui, Russell T. Johns, and Kaveh Ahmadi. "A Rapid and Robust Alternative to Rachford-Rice in Flash Calculations." Fluid Phase Equilibria 316 (February 25, 2012): 85-97. doi:10.1016/j.fluid.2011.12.005.

[3] Warren, John H. "Explicit Determination of the Vapor Fraction in Flash Calculations." Pennsylvania State University, 1991.

[1] Leibovici, ClaudeF., and Jean Neoschil. "A New Look at the Rachford-Rice Equation." Fluid Phase Equilibria 74 (July 15, 1992): 303-8. https://doi.org/10.1016/0378-3812(92)85069-K.

[1] Okuno, Ryosuke, Russell Johns, and Kamy Sepehrnoori. "A New Algorithm for Rachford-Rice for Multiphase Compositional Simulation." SPE Journal 15, no. 02 (June 1, 2010): 313-25. https://doi.org/10.2118/117752-PA.

[2] Li, Zhidong, and Abbas Firoozabadi. "Initialization of Phase Fractions in Rachford-Rice Equations for Robust and Efficient Three-Phase Split Calculation." Fluid Phase Equilibria 332 (October 25, 2012): 21-27. https://doi.org/10.1016/j.fluid.2012.06.021.

[3] Gao, Ran, Xiaolong Yin, and Zhiping Li. "Hybrid Newton-Successive Substitution Method for Multiphase Rachford-Rice Equations." Entropy 20, no. 6 (June 2018): 452. https://doi.org/10.3390/e20060452.

[4] Leibovici, Claude F., and Jean Neoschil. "A Solution of Rachford-Rice Equations for Multiphase Systems." Fluid Phase Equilibria 112, no. 2 (December 1, 1995): 217-21. https://doi.org/10.1016/0378-3812(95)02797-I.

[1] Gao, Ran, Xiaolong Yin, and Zhiping Li. "Hybrid Newton-Successive Substitution Method for Multiphase Rachford-Rice Equations." Entropy 20, no. 6 (June 2018): 452. https://doi.org/10.3390/e20060452.

[1] Weigle, Brett D. "A Generalized Polynomial Form of the Objective Function in Flash Calculations." Pennsylvania State University, 1992.

[2] Warren, John H. "Explicit Determination of the Vapor Fraction in Flash Calculations." Pennsylvania State University, 1991.

[3] Monroy-Loperena, Rosendo, and Felipe D. Vargas-Villamil. "On the Determination of the Polynomial Defining of Vapor-Liquid Split of Multicomponent Mixtures." Chemical Engineering Science 56, no. 20 (October 1, 2001): 5865-68. https://doi.org/10.1016/S0009-2509(01)00267-6.

[1] Rachford, H. H. Jr, and J. D. Rice. "Procedure for Use of Electronic Digital Computers in Calculating Flash Vaporization Hydrocarbon Equilibrium." Journal of Petroleum Technology 4, no. 10 (October 1, 1952): 19-3. doi:10.2118/952327-G.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[2] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Ruscic, Branko, Reinhardt E. Pinzon, Gregor von Laszewski, Deepti Kodeboyina, Alexander Burcat, David Leahy, David Montoy, and Albert F. Wagner. "Active Thermochemical Tables: Thermochemistry for the 21st Century." Journal of Physics: Conference Series 16, no. 1 (January 1, 2005): 561. doi:10.1088/1742-6596/16/1/078.

[2] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[3] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Ruscic, Branko, Reinhardt E. Pinzon, Gregor von Laszewski, Deepti Kodeboyina, Alexander Burcat, David Leahy, David Montoy, and Albert F. Wagner. "Active Thermochemical Tables: Thermochemistry for the 21st Century." Journal of Physics: Conference Series 16, no. 1 (January 1, 2005): 561. doi:10.1088/1742-6596/16/1/078.

[2] Frenkel`, M. L, Texas Engineering Experiment Station, and Thermodynamics Research Center. Thermodynamics of Organic Compounds in the Gas State. College Station, Tex.: Thermodynamics Research Center, 1994.

[3] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[4] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[5] Joback, K.G., and R.C. Reid. "Estimation of Pure-Component Properties from Group-Contributions." Chemical Engineering Communications 57, no. 1-6 (July 1, 1987): 233-43. doi:10.1080/00986448708960487.

[6] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[2] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[2] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[3] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] "Standard Gibbs Free Energy of Formation Calculations Chemistry Tutorial." Accessed March, 2019. https://www.ausetute.com.au/gibbsform.html.

[1] Sen, S. K., Hans Agarwal, and Sagar Sen. "Chemical Equation Balancing: An Integer Programming Approach." Mathematical and Computer Modelling 44, no. 7 (October 1, 2006): 678-91. https://doi.org/10.1016/j.mcm.2006.02.004.

[2] URAVNOTE, NOVOODKRITI PARADOKSI V. TEORIJI, and ENJA KEMIJSKIH REAKCIJ. "New Discovered Paradoxes in Theory of Balancing Chemical Reactions." Materiali in Tehnologije 45, no. 6 (2011): 503-22.

[1] Sen, S. K., Hans Agarwal, and Sagar Sen. "Chemical Equation Balancing: An Integer Programming Approach." Mathematical and Computer Modelling 44, no. 7 (October 1, 2006): 678-91. https://doi.org/10.1016/j.mcm.2006.02.004.

[2] URAVNOTE, NOVOODKRITI PARADOKSI V. TEORIJI, and ENJA KEMIJSKIH REAKCIJ. "New Discovered Paradoxes in Theory of Balancing Chemical Reactions." Materiali in Tehnologije 45, no. 6 (2011): 503-22.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. Boca Raton, FL: CRC press, 2014.

[2] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[1] IAPWS, 1997. Release on the Refractive Index of Ordinary Water Substance as a Function of Wavelength, Temperature and Pressure.

[1] "RefractometerData Book-Refractive Index and Brix | ATAGO CO., LTD." Accessed June 13, 2020. https://www.atago.net/en/databook-refractometer_relationship.php.

[1] "RefractometerData Book-Refractive Index and Brix | ATAGO CO., LTD." Accessed June 13, 2020. https://www.atago.net/en/databook-refractometer_relationship.php.

[1] Panuganti, Sai R., Fei Wang, Walter G. Chapman, and Francisco M. Vargas. "A Simple Method for Estimation of Dielectric Constants and Polarizabilities of Nonpolar and Slightly Polar Hydrocarbons." International Journal of Thermophysics 37, no. 7 (June 6, 2016): 1-24. doi:10.1007/s10765-016-2075-8.

[1] Panuganti, Sai R., Fei Wang, Walter G. Chapman, and Francisco M. Vargas. "A Simple Method for Estimation of Dielectric Constants and Polarizabilities of Nonpolar and Slightly Polar Hydrocarbons." International Journal of Thermophysics 37, no. 7 (June 6, 2016): 1-24. doi:10.1007/s10765-016-2075-8.

[1] Panuganti, Sai R., Fei Wang, Walter G. Chapman, and Francisco M. Vargas. "A Simple Method for Estimation of Dielectric Constants and Polarizabilities of Nonpolar and Slightly Polar Hydrocarbons." International Journal of Thermophysics 37, no. 7 (June 6, 2016): 1-24. doi:10.1007/s10765-016-2075-8.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-RefractiveIndex/RIXExpansion.htm.

[1] International Agency for Research on Cancer. Agents Classified by the IARC Monographs, Volumes 1-115. Lyon, France: IARC; 2020 Available from: http://monographs.iarc.fr/ENG/Classification/

[2] NTP (National Toxicology Program). 2021. Report on Carcinogens, Fifteenth Edition.; Research Triangle Park, NC: U.S. Department of Health and Human Services, Public Health Service. https://doi.org/10.22427/NTP-OTHER-1003

[1] IEC. "IEC 60079-20-1:2010 Explosive atmospheres - Part 20-1: Material characteristics for gas and vapour classification - Test methods and data." https://webstore.iec.ch/publication/635. See also https://law.resource.org/pub/in/bis/S05/is.iec.60079.20.1.2010.pdf

[2] National Fire Protection Association. NFPA 497: Recommended Practice for the Classification of Flammable Liquids, Gases, or Vapors and of Hazardous. NFPA, 2008.

[3] Serat, Fatima Zohra, Ali Mustapha Benkouider, Ahmed Yahiaoui, and Farid Bagui. "Nonlinear Group Contribution Model for the Prediction of Flash Points Using Normal Boiling Points." Fluid Phase Equilibria 449 (October 15, 2017): 52-59. doi:10.1016/j.fluid.2017.06.008.

[4] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[1] IEC. "IEC 60079-20-1:2010 Explosive atmospheres - Part 20-1: Material characteristics for gas and vapour classification - Test methods and data." https://webstore.iec.ch/publication/635. See also https://law.resource.org/pub/in/bis/S05/is.iec.60079.20.1.2010.pdf

[2] National Fire Protection Association. NFPA 497: Recommended Practice for the Classification of Flammable Liquids, Gases, or Vapors and of Hazardous. NFPA, 2008.

[3] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[1] IEC. "IEC 60079-20-1:2010 Explosive atmospheres - Part 20-1: Material characteristics for gas and vapour classification - Test methods and data." https://webstore.iec.ch/publication/635. See also https://law.resource.org/pub/in/bis/S05/is.iec.60079.20.1.2010.pdf

[2] National Fire Protection Association. NFPA 497: Recommended Practice for the Classification of Flammable Liquids, Gases, or Vapors and of Hazardous. NFPA, 2008.

[3] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[1] Suzuki, Takahiro. "Note: Empirical Relationship between Lower Flammability Limits and Standard Enthalpies of Combustion of Organic Compounds." Fire and Materials 18, no. 5 (September 1, 1994): 333-36. doi:10.1002/fam.810180509.

[1] Crowl, Daniel A., and Joseph F. Louvar. Chemical Process Safety: Fundamentals with Applications. 2E. Upper Saddle River, N.J: Prentice Hall, 2001.

[2] Jones, G. W. "Inflammation Limits and Their Practical Application in Hazardous Industrial Operations." Chemical Reviews 22, no. 1 (February 1, 1938): 1-26. doi:10.1021/cr60071a001

[1] Standardization, International Organization for. ISO 10156: 2017 : Gas Cylinders - Gases and Gas Mixtures - Determination of Fire Potential and Oxidizing Ability for the Selection of Cylinder Valve Outlets, 2017.

[1] IEC. "IEC 60079-20-1:2010 Explosive atmospheres - Part 20-1: Material characteristics for gas and vapour classification - Test methods and data." https://webstore.iec.ch/publication/635. See also https://law.resource.org/pub/in/bis/S05/is.iec.60079.20.1.2010.pdf

[2] National Fire Protection Association. NFPA 497: Recommended Practice for the Classification of Flammable Liquids, Gases, or Vapors and of Hazardous. NFPA, 2008.

[3] Wikidata. Wikidata. Accessed via API. https://www.wikidata.org/

[1] Suzuki, Takahiro, and Kozo Koide. "Short Communication: Correlation between Upper Flammability Limits and Thermochemical Properties of Organic Compounds." Fire and Materials 18, no. 6 (November 1, 1994): 393-97. doi:10.1002/fam.810180608.

[1] Crowl, Daniel A., and Joseph F. Louvar. Chemical Process Safety: Fundamentals with Applications. 2E. Upper Saddle River, N.J: Prentice Hall, 2001.

[2] Jones, G. W. "Inflammation Limits and Their Practical Application in Hazardous Industrial Operations." Chemical Reviews 22, no. 1 (February 1, 1938): 1-26. doi:10.1021/cr60071a001

[1] Crowl, Daniel A., and Joseph F. Louvar. Chemical Process Safety: Fundamentals with Applications. 2E. Upper Saddle River, N.J: Prentice Hall, 2001.

[1] ACGIH. Industrial Ventilation: A Manual of Recommended Practice, 23rd Edition. American Conference of Governmental and Industrial Hygenists, 2004.

[1] ACGIH. Industrial Ventilation: A Manual of Recommended Practice, 23rd Edition. American Conference of Governmental and Industrial Hygenists, 2004.

[1] NFPA (National Fire Prevention Association). NFPA 30: Flammable and Combustible Liquids Code, 2008. National Fire Protection Association (NFPA), 2007.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, 8E. McGraw-Hill Professional, 2007.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Barton, Allan F. M. CRC Handbook of Solubility Parameters and Other Cohesion Parameters, Second Edition. CRC Press, 1991.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Wier, Ron D., and Robert N. Goldberg. "On the Conversion of Thermodynamic Properties to the Basis of the International Temperature Scale of 1990." The Journal of Chemical Thermodynamics 28, no. 3 (March 1996): 261-76. doi:10.1006/jcht.1996.0026.

[2] Goldberg, Robert N., and R. D. Weir. "Conversion of Temperatures and Thermodynamic Properties to the Basis of the International Temperature Scale of 1990 (Technical Report)." Pure and Applied Chemistry 64, no. 10 (1992): 1545-1562. doi:10.1351/pac199264101545.

[1] Bedford, R. E., G. Bonnier, H. Maas, and F. Pavese. "Techniques for Approximating the International Temperature Scale of 1990." Bureau International Des Poids et Mesures, Sfievres, 1990.

[2] Wier, Ron D., and Robert N. Goldberg. "On the Conversion of Thermodynamic Properties to the Basis of the International Temperature Scale of 1990." The Journal of Chemical Thermodynamics 28, no. 3 (March 1996): 261-76. doi:10.1006/jcht.1996.0026.

[3] Goldberg, Robert N., and R. D. Weir. "Conversion of Temperatures and Thermodynamic Properties to the Basis of the International Temperature Scale of 1990 (Technical Report)." Pure and Applied Chemistry 64, no. 10 (1992): 1545-1562. doi:10.1351/pac199264101545.

[4] Code10.info. "Conversions among International Temperature Scales." Accessed May 22, 2016. http://www.code10.info/index.php%3Foption%3Dcom_content%26view%3Darticle%26id%3D83:conversions-among-international-temperature-scales%26catid%3D60:temperature%26Itemid%3D83.

[1] Scheffy, W. J., and E. F. Johnson. "Thermal Conductivities of Liquids at High Temperatures." Journal of Chemical & Engineering Data 6, no. 2 (April 1, 1961): 245-49. doi:10.1021/je60010a019

[1] Riedel, L.: Chem. Ing. Tech., 21, 349 (1949); 23: 59, 321, 465 (1951)

[2] Maejima, T., private communication, 1973

[3] Properties of Gases and Liquids", 3rd Ed., McGraw-Hill, 1977

[1] Lakshmi, D. S., and D. H. L. Prasad. "A Rapid Estimation Method for Thermal Conductivity of Pure Liquids." The Chemical Engineering Journal 48, no. 3 (April 1992): 211-14. doi:10.1016/0300-9467(92)80037-B

[1] Gharagheizi, Farhad, Poorandokht Ilani-Kashkouli, Mehdi Sattari, Amir H. Mohammadi, Deresh Ramjugernath, and Dominique Richon. "Development of a General Model for Determination of Thermal Conductivity of Liquid Chemical Compounds at Atmospheric Pressure." AIChE Journal 59, no. 5 (May 1, 2013): 1702-8. doi:10.1002/aic.13938

[1] Nicola, Giovanni Di, Eleonora Ciarrocchi, Mariano Pierantozzi, and Roman Stryjek. "A New Equation for the Thermal Conductivity of Organic Compounds." Journal of Thermal Analysis and Calorimetry 116, no. 1 (April 1, 2014): 135-40. doi:10.1007/s10973-013-3422-7

[1] Di Nicola, Giovanni, Eleonora Ciarrocchi, Gianluca Coccia, and Mariano Pierantozzi. "Correlations of Thermal Conductivity for Liquid Refrigerants at Atmospheric Pressure or near Saturation." International Journal of Refrigeration. 2014. doi:10.1016/j.ijrefrig.2014.06.003

[1] Bahadori, Alireza, and Saeid Mokhatab. "Estimating Thermal Conductivity of Hydrocarbons." Chemical Engineering 115, no. 13 (December 2008): 52-54

[1] Mersmann, Alfons, and Matthias Kind. "Prediction of Mechanical and Thermal Properties of Pure Liquids, of Critical Data, and of Vapor Pressure." Industrial & Engineering Chemistry Research, January 31, 2017. https://doi.org/10.1021/acs.iecr.6b04323.

[1] Missenard, F. A., Thermal Conductivity of Organic Liquids of a Series or a Group of Liquids , Rev. Gen.Thermodyn., 101 649 (1970).

[2] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] Missenard, F. A., Thermal Conductivity of Organic Liquids of a Series or a Group of Liquids , Rev. Gen.Thermodyn., 101 649 (1970).

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. The Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[2] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[3] Focke, Walter W. "Correlating Thermal-Conductivity Data for Ternary Liquid Mixtures." International Journal of Thermophysics 29, no. 4 (August 1, 2008): 1342-60. https://doi.org/10.1007/s10765-008-0465-2.

[1] Li, C. C. "Thermal Conductivity of Liquid Mixtures." AIChE Journal 22, no. 5 (1976): 927-30. https://doi.org/10.1002/aic.690220520.

[2] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. The Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[2] Filippov, L. P.: Vest. Mosk. Univ., Ser. Fiz. Mat. Estestv. Nauk, (8I0E): 67-69A955); Chem. Abstr., 50: 8276 A956). Filippov, L. P., and N. S. Novoselova: Vestn. Mosk. Univ., Ser. F iz. Mat. Estestv.Nauk, CI0B): 37-40A955); Chem. Abstr., 49: 11366 A955).

[1] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Bromley, LeRoy A., Berkeley. University of California, and U.S. Atomic Energy Commission. Thermal Conductivity of Gases at Moderate Pressures. UCRL;1852. Berkeley, CA: University of California Radiation Laboratory, 1952.

[2] Stiel, Leonard I., and George Thodos. "The Thermal Conductivity of Nonpolar Substances in the Dense Gaseous and Liquid Regions." AIChE Journal 10, no. 1 (January 1, 1964): 26-30. doi:10.1002/aic.690100114

[3] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] Chung, Ting Horng, Lloyd L. Lee, and Kenneth E. Starling. "Applications of Kinetic Gas Theories and Multiparameter Correlation for Prediction of Dilute Gas Viscosity and Thermal Conductivity." Industrial & Engineering Chemistry Fundamentals 23, no. 1 (February 1, 1984): 8-13. doi:10.1021/i100013a002

[2] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Ely, James F., and H. J. M. Hanley. "Prediction of Transport Properties. 2. Thermal Conductivity of Pure Fluids and Mixtures." Industrial & Engineering Chemistry Fundamentals 22, no. 1 (February 1, 1983): 90-97. doi:10.1021/i100009a016.

[2] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Gharagheizi, Farhad, Poorandokht Ilani-Kashkouli, Mehdi Sattari, Amir H. Mohammadi, Deresh Ramjugernath, and Dominique Richon. "Development of a General Model for Determination of Thermal Conductivity of Liquid Chemical Compounds at Atmospheric Pressure." AIChE Journal 59, no. 5 (May 1, 2013): 1702-8. doi:10.1002/aic.13938

[1] Bahadori, Alireza, and Saeid Mokhatab. "Estimating Thermal Conductivity of Hydrocarbons." Chemical Engineering 115, no. 13 (December 2008): 52-54

[1] Stiel, Leonard I., and George Thodos. "The Thermal Conductivity of Nonpolar Substances in the Dense Gaseous and Liquid Regions." AIChE Journal 10, no. 1 (January 1, 1964): 26-30. doi:10.1002/aic.690100114.

[2] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Ely, James F., and H. J. M. Hanley. "Prediction of Transport Properties. 2. Thermal Conductivity of Pure Fluids and Mixtures." Industrial & Engineering Chemistry Fundamentals 22, no. 1 (February 1, 1983): 90-97. doi:10.1021/i100009a016.

[2] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Chung, Ting Horng, Mohammad Ajlan, Lloyd L. Lee, and Kenneth E. Starling. "Generalized Multiparameter Correlation for Nonpolar and Polar Fluid Transport Properties." Industrial & Engineering Chemistry Research 27, no. 4 (April 1, 1988): 671-79. doi:10.1021/ie00076a024.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Lindsay, Alexander L., and LeRoy A. Bromley. "Thermal Conductivity of Gas Mixtures." Industrial & Engineering Chemistry 42, no. 8 (August 1, 1950): 1508-11. doi:10.1021/ie50488a017.

[2] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[3] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Huber, M. L., R. A. Perkins, D. G. Friend, J. V. Sengers, M. J. Assael, I. N. Metaxa, K. Miyagawa, R. Hellmann, and E. Vogel. "New International Formulation for the Thermal Conductivity of H2O." Journal of Physical and Chemical Reference Data 41, no. 3 (September 1, 2012): 033102. doi:10.1063/1.4738955.

[1] Lemmon, E. W., and R. T. Jacobsen. "Viscosity and Thermal Conductivity Equations for Nitrogen, Oxygen, Argon, and Air." International Journal of Thermophysics 25, no. 1 (January 1, 2004): 21-69. https://doi.org/10.1023/B:IJOT.0000022327.04529.f3.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-ThermalCondSatL/PPDS8.htm

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-ThermalCondG/PPDS3-ThermCondGas.htm

[1] Kooijman, Harry A., and Ross Taylor. The ChemSep Book. Books on Demand Norderstedt, Germany, 2000.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[2] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[1] Staveley, L. A. K., L. Q. Lobo, and J. C. G. Calado. "Triple-Points of Low Melting Substances and Their Use in Cryogenic Work." Cryogenics 21, no. 3 (March 1981): 131-144. doi:10.1016/0011-2275(81)90264-2.

[2] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[3] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] Staveley, L. A. K., L. Q. Lobo, and J. C. G. Calado. "Triple-Points of Low Melting Substances and Their Use in Cryogenic Work." Cryogenics 21, no. 3 (March 1981): 131-144. doi:10.1016/0011-2275(81)90264-2.

[2] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[3] Huber, Marcia L., Eric W. Lemmon, Ian H. Bell, and Mark O. McLinden. "The NIST REFPROP Database for Highly Accurate Properties of Industrially Important Fluids." Industrial & Engineering Chemistry Research 61, no. 42 (October 26, 2022): 15449-72. https://doi.org/10.1021/acs.iecr.2c01427.

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[3] Gmehling, Jurgen, Barbel Kolbe, Michael Kleiber, and Jurgen Rarey. Chemical Thermodynamics for Process Simulation. 1st edition. Weinheim: Wiley-VCH, 2012.

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[2] Pratt, R. M. "Thermodynamic Properties Involving Derivatives: Using the Peng-Robinson Equation of State." Chemical Engineering Education 35, no. 2 (March 1, 2001): 112-115.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, 8E. McGraw-Hill Professional, 2007.

[3] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Pini, Matteo. "NicePropp: An Interactive Python-Based Educational Tool for Non-Ideal Compressible Fluid Dynamics." SoftwareX 17 (2022): 100897.

[2] Kouremenos, D. A., and K. A. Antonopoulos. "Isentropic Exponents of Real Gases and Application for the Air at Temperatures from 150 K to 450 K." Acta Mechanica 65, no. 1 (January 1, 1987): 81-99. https://doi.org/10.1007/BF01176874.

[1] Pini, Matteo. "NiceProp: An Interactive Python-Based Educational Tool for Non-Ideal Compressible Fluid Dynamics." SoftwareX 17 (2022): 100897.

[2] Kouremenos, D. A., and K. A. Antonopoulos. "Isentropic Exponents of Real Gases and Application for the Air at Temperatures from 150 K to 450 K." Acta Mechanica 65, no. 1 (January 1, 1987): 81-99. https://doi.org/10.1007/BF01176874.

[1] Pini, Matteo. "NiceProp: An Interactive Python-Based Educational Tool for Non-Ideal Compressible Fluid Dynamics." SoftwareX 17 (2022): 100897.

[2] Kouremenos, D. A., and K. A. Antonopoulos. "Isentropic Exponents of Real Gases and Application for the Air at Temperatures from 150 K to 450 K." Acta Mechanica 65, no. 1 (January 1, 1987): 81-99. https://doi.org/10.1007/BF01176874.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Venkatarathnam, G., and L. R. Oellrich. "Identification of the Phase of a Fluid Using Partial Derivatives of Pressure, Volume, and Temperature without Reference to Saturation Properties: Applications in Phase Equilibria Calculations." Fluid Phase Equilibria 301, no. 2 (February 25, 2011): 225-33. doi:10.1016/j.fluid.2010.12.001.

[2] Jayanti, Pranava Chaitanya, and G. Venkatarathnam. "Identification of the Phase of a Substance from the Derivatives of Pressure, Volume and Temperature, without Prior Knowledge of Saturation Properties: Extension to Solid Phase." Fluid Phase Equilibria 425 (October 15, 2016): 269-277. doi:10.1016/j.fluid.2016.06.001.

[1] Venkatarathnam, G., and L. R. Oellrich. "Identification of the Phase of a Fluid Using Partial Derivatives of Pressure, Volume, and Temperature without Reference to Saturation Properties: Applications in Phase Equilibria Calculations." Fluid Phase Equilibria 301, no. 2 (February 25, 2011): 225-33. doi:10.1016/j.fluid.2010.12.001.

[2] Jayanti, Pranava Chaitanya, and G. Venkatarathnam. "Identification of the Phase of a Substance from the Derivatives of Pressure, Volume and Temperature, without Prior Knowledge of Saturation Properties: Extension to Solid Phase." Fluid Phase Equilibria 425 (October 15, 2016): 269-277. doi:10.1016/j.fluid.2016.06.001.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, 8E. McGraw-Hill Professional, 2007.

[2] Johnson III, Russell D. "NIST 101. Computational Chemistry Comparison and Benchmark Database," 1999. https://cccbdb.nist.gov

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Gmehling, Jurgen, Barbel Kolbe, Michael Kleiber, and Jurgen Rarey. Chemical Thermodynamics for Process Simulation. 1st edition. Weinheim: Wiley-VCH, 2012.

[2] Pratt, R. M. "Thermodynamic Properties Involving Derivatives: Using the Peng-Robinson Equation of State." Chemical Engineering Education 35, no. 2 (March 1, 2001): 112-115.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, 8E. McGraw-Hill Professional, 2007.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Antoine, C. 1888. Tensions des Vapeurs: Nouvelle Relation Entre les Tensions et les Tempé. Compt.Rend. 107:681-684.

[3] Yaws, Carl L. The Yaws Handbook of Vapor Pressure: Antoine Coefficients. 1 edition. Houston, Tex: Gulf Publishing Company, 2007.

[1] Wagner, W. "New Vapour Pressure Measurements for Argon and Nitrogen and a New Method for Establishing Rational Vapour Pressure Equations." Cryogenics 13, no. 8 (August 1973): 470-82. doi:10.1016/0011-2275(73)90003-9

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] PPDS2 Temperature-Dependent Equation Forms. National Engineering Laboratory, 2004 https://web.archive.org/web/20050510061545/http://www.ppds.co.uk/library/pdf/PPDS_EquationForms.pdf

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] McGarry, Jack. "Correlation and Prediction of the Vapor Pressures of Pure Liquids over Large Pressure Ranges." Industrial & Engineering Chemistry Process Design and Development 22, no. 2 (April 1, 1983): 313-22. doi:10.1021/i200021a023.

[3] PPDS2 Temperature-Dependent Equation Forms. National Engineering Laboratory, 2004 https://web.archive.org/web/20050510061545/http://www.ppds.co.uk/library/pdf/PPDS_EquationForms.pdf

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Yaws, Carl L. Chemical Properties Handbook: Physical, Thermodynamic, Environmental, Transport, Safety, and Health Related Properties for Organic and Inorganic Chemicals. McGraw-Hill, 2001.

[2] "ThermoData Engine (TDE103a V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103a/Eqns-Pure-PhaseBoundaryLG/Yaws-VaporPressure.htm.

[1] "ThermoData Engine (TDE103a V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-PhaseBoundaryLG/PVExpansion.htm

[1] Lee, Byung Ik, and Michael G. Kesler. "A Generalized Thermodynamic Correlation Based on Three-Parameter Corresponding States." AIChE Journal 21, no. 3 (1975): 510-527. doi:10.1002/aic.690210313.

[2] Reid, Robert C..; Prausnitz, John M.;; Poling, Bruce E. The Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Ambrose, D., and J. Walton. "Vapour Pressures up to Their Critical Temperatures of Normal Alkanes and 1-Alkanols." Pure and Applied Chemistry 61, no. 8 (1989): 1395-1403. doi:10.1351/pac198961081395.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Reid, Robert C..; Prausnitz, John M.;; Poling, Bruce E. The Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Sanjari, Ehsan, Mehrdad Honarmand, Hamidreza Badihi, and Ali Ghaheri. "An Accurate Generalized Model for Predict Vapor Pressure of Refrigerants." International Journal of Refrigeration 36, no. 4 (June 2013): 1327-32. doi:10.1016/j.ijrefrig.2013.01.007.

[1] Edalat, M., R. B. Bozar-Jomehri, and G. A. Mansoori. "Generalized Equation Predicts Vapor Pressure of Hydrocarbons." Oil and Gas Journal; 91:5 (February 1, 1993).

[1] Goodman, B. T., W. V. Wilding, J. L. Oscarson, and R. L. Rowley. "Use of the DIPPR Database for the Development of QSPR Correlations: Solid Vapor Pressure and Heat of Sublimation of Organic Compounds." International Journal of Thermophysics 25, no. 2 (March 1, 2004): 337-50. https://doi.org/10.1023/B:IJOT.0000028471.77933.80.

[2] Feistel, Rainer, and Wolfgang Wagner. "Sublimation Pressure and Sublimation Enthalpy of H2O Ice Ih between 0 and 273.16K." Geochimica et Cosmochimica Acta 71, no. 1 (January 1, 2007): 36-45. https://doi.org/10.1016/j.gca.2006.08.034.

[1] Kretzschmar, Hans-Joachim, and Wolfgang Wagner. International Steam Tables: Properties of Water and Steam Based on the Industrial Formulation IAPWS-IF97. Springer, 2019.

[1] Kretzschmar, Hans-Joachim, and Wolfgang Wagner. International Steam Tables: Properties of Water and Steam Based on the Industrial Formulation IAPWS-IF97. Springer, 2019.

[1] Kretzschmar, Hans-Joachim, and Wolfgang Wagner. International Steam Tables: Properties of Water and Steam Based on the Industrial Formulation IAPWS-IF97. Springer, 2019.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] McGarry, Jack. "Correlation and Prediction of the Vapor Pressures of Pure Liquids over Large Pressure Ranges." Industrial & Engineering Chemistry Process Design and Development 22, no. 2 (April 1, 1983): 313-22. doi:10.1021/i200021a023.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[4] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[5] Alcock, C. B., V. P. Itkin, and M. K. Horrigan. "Vapour Pressure Equations for the Metallic Elements: 298-2500K." Canadian Metallurgical Quarterly 23, no. 3 (July 1, 1984): 309-13. https://doi.org/10.1179/cmq.1984.23.3.309.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Prausnitz, John M., Rudiger N. Lichtenthaler, and Edmundo Gomes de Azevedo. Molecular Thermodynamics of Fluid-Phase Equilibria. 3rd edition. Upper Saddle River, N.J: Prentice Hall, 1998.

[2] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Prausnitz, John M., Rudiger N. Lichtenthaler, and Edmundo Gomes de Azevedo. Molecular Thermodynamics of Fluid-Phase Equilibria. 3rd edition. Upper Saddle River, N.J: Prentice Hall, 1998.

[2] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Pitzer, Kenneth S., and R. F. Curl. "The Volumetric and Thermodynamic Properties of Fluids. III. Empirical Equation for the Second Virial Coefficient1." Journal of the American Chemical Society 79, no. 10 (May 1, 1957): 2369-70. doi:10.1021/ja01567a007.

[1] Smith, H. C. Van Ness Joseph M. Introduction to Chemical Engineering Thermodynamics 4E 1987.

[1] Tsonopoulos, Constantine. "An Empirical Correlation of Second Virial Coefficients." AIChE Journal 20, no. 2 (March 1, 1974): 263-72. doi:10.1002/aic.690200209.

[1] Tsonopoulos, C., and J. L. Heidman. "From the Virial to the Cubic Equation of State." Fluid Phase Equilibria 57, no. 3 (1990): 261-76. doi:10.1016/0378-3812(90)85126-U

[2] Tsonopoulos, Constantine, and John H. Dymond. "Second Virial Coefficients of Normal Alkanes, Linear 1-Alkanols (and Water), Alkyl Ethers, and Their Mixtures." Fluid Phase Equilibria, International Workshop on

Vapour-Liquid Equilibria and Related Properties in Binary and Ternary Mixtures of Ethers, Alkanes and Alkanols, 133, no. 1-2 (June 1997): 11-34. doi:10.1016/S0378-3812(97)00058-7.

[1] O`Connell, J. P., and J. M. Prausnitz. "Empirical Correlation of Second Virial Coefficients for Vapor-Liquid Equilibrium Calculations." Industrial & Engineering Chemistry Process Design and Development 6, no. 2 (April 1, 1967): 245-50. https://doi.org/10.1021/i260022a016.

[1] Xiang, H. W. "The New Simple Extended Corresponding-States Principle: Vapor Pressure and Second Virial Coefficient." Chemical Engineering Science 57, no. 8 (April 2002): 1439049. https://doi.org/10.1016/S0009-2509(02)00017-9.

[1] Meng, Long, Yuan-Yuan Duan, and Lei Li. "Correlations for Second and Third Virial Coefficients of Pure Fluids." Fluid Phase Equilibria 226 (December 10, 2004): 109-20. https://doi.org/10.1016/j.fluid.2004.09.023.

[1] Meng, Long, Yuan-Yuan Duan, and Lei Li. "Correlations for Second and Third Virial Coefficients of Pure Fluids." Fluid Phase Equilibria 226 (December 10, 2004): 109-20. https://doi.org/10.1016/j.fluid.2004.09.023.

[1] Orbey, Hasan, and J. H. Vera. "Correlation for the Third Virial Coefficient Using Tc, Pc and as Parameters." AIChE Journal 29, no. 1 (January 1, 1983): 107-13. https://doi.org/10.1002/aic.690290115.

[1] Liu, D. X., and H. W. Xiang. "Corresponding-States Correlation and Prediction of Third Virial Coefficients for a Wide Range of Substances." International Journal of Thermophysics 24, no. 6 (November 1, 2003): 1667-80. https://doi.org/10.1023/B:IJOT.0000004098.98614.38.

[1] Tarakad, Ramanathan R., and Ronald P. Danner. "An Improved Corresponding States Method for Polar Fluids: Correlation of Second Virial Coefficients." AIChE Journal 23, no. 5 (1977): 685-95. https://doi.org/10.1002/aic.690230510.

[2] Meng, Long, and Yuan-Yuan Duan. "Prediction of the Second Cross Virial Coefficients of Nonpolar Binary Mixtures." Fluid Phase Equilibria 238 (December 1, 2005): 229-38. https://doi.org/10.1016/j.fluid.2005.10.007.

[1] Tarakad, Ramanathan R., and Ronald P. Danner. "An Improved Corresponding States Method for Polar Fluids: Correlation of Second Virial Coefficients." AIChE Journal 23, no. 5 (1977): 685-95. https://doi.org/10.1002/aic.690230510.

[2] Meng, Long, and Yuan-Yuan Duan. "Prediction of the Second Cross Virial Coefficients of Nonpolar Binary Mixtures." Fluid Phase Equilibria 238 (December 1, 2005): 229-38. https://doi.org/10.1016/j.fluid.2005.10.007.

[1] Tarakad, Ramanathan R., and Ronald P. Danner. "An Improved Corresponding States Method for Polar Fluids: Correlation of Second Virial Coefficients." AIChE Journal 23, no. 5 (1977): 685-95. https://doi.org/10.1002/aic.690230510.

[2] Meng, Long, and Yuan-Yuan Duan. "Prediction of the Second Cross Virial Coefficients of Nonpolar Binary Mixtures." Fluid Phase Equilibria 238 (December 1, 2005): 229-38. https://doi.org/10.1016/j.fluid.2005.10.007.

[1] Tarakad, Ramanathan R., and Ronald P. Danner. "An Improved Corresponding States Method for Polar Fluids: Correlation of Second Virial Coefficients." AIChE Journal 23, no. 5 (1977): 685-95. https://doi.org/10.1002/aic.690230510.

[2] Meng, Long, and Yuan-Yuan Duan. "Prediction of the Second Cross Virial Coefficients of Nonpolar Binary Mixtures." Fluid Phase Equilibria 238 (December 1, 2005): 229-38. https://doi.org/10.1016/j.fluid.2005.10.007.

[1] Estela-Uribe, J. F., and J. Jaramillo. "Generalised Virial Equation of State for Natural Gas Systems." Fluid Phase Equilibria 231, no. 1 (April 1, 2005): 84-98. https://doi.org/10.1016/j.fluid.2005.01.005.

[2] Lee, Byung Ik, and Michael G. Kesler. "A Generalized Thermodynamic Correlation Based on Three-Parameter Corresponding States." AIChE Journal 21, no. 3 (1975): 510-27. https://doi.org/10.1002/aic.690210313.

[1] Meng, Long, and Yuan-Yuan Duan. "Prediction of the Second Cross Virial Coefficients of Nonpolar Binary Mixtures." Fluid Phase Equilibria 238 (December 1, 2005): 229-38. https://doi.org/10.1016/j.fluid.2005.10.007.

[1] Letsou, Athena, and Leonard I. Stiel. "Viscosity of Saturated Nonpolar Liquids at Elevated Pressures." AIChE Journal 19, no. 2 (1973): 409-11. doi:10.1002/aic.690190241.

[1] Przedziecki, J. W., and T. Sridhar. "Prediction of Liquid Viscosities." AIChE Journal 31, no. 2 (February 1, 1985): 333-35. doi:10.1002/aic.690310225.

[1] Lucas, Klaus. "Ein Einfaches Verfahren Zur Berechnung Der Viskositat von Gasen Und Gasgemischen." Chemie Ingenieur Technik 46, no. 4 (February 1, 1974): 157-157. doi:10.1002/cite.330460413.

[2] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Yoon, Poong, and George Thodos. "Viscosity of Nonpolar Gaseous Mixtures at Normal Pressures." AIChE Journal 16, no. 2 (1970): 300-304. doi:10.1002/aic.690160225.

[1] Stiel, Leonard I., and George Thodos. "The Viscosity of Nonpolar Gases at Normal Pressures." AIChE Journal 7, no. 4 (1961): 611-15. doi:10.1002/aic.690070416.

[1] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Gharagheizi, Farhad, Ali Eslamimanesh, Mehdi Sattari, Amir H. Mohammadi, and Dominique Richon. "Corresponding States Method for Determination of the Viscosity of Gases at Atmospheric Pressure." Industrial & Engineering Chemistry Research 51, no. 7 (February 22, 2012): 3179-85. doi:10.1021/ie202591f.

[1] Herning, F. and Zipperer, L,: "Calculation of the Viscosity of Technical Gas Mixtures from the Viscosity of Individual Gases, german", Gas u. Wasserfach (1936) 79, No. 49, 69.

[1] Brokaw, R. S. "Predicting Transport Properties of Dilute Gases." Industrial & Engineering Chemistry Process Design and Development 8, no. 2 (April 1, 1969): 240-53. doi:10.1021/i260030a015.

[2] Brokaw, R. S. Viscosity of Gas Mixtures, NASA-TN-D-4496, 1968.

[3] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] Wilke, C. R. "A Viscosity Equation for Gas Mixtures." The Journal of Chemical Physics 18, no. 4 (April 1, 1950): 517-19. https://doi.org/10.1063/1.1747673.

[1] Wilke, C. R. "A Viscosity Equation for Gas Mixtures." The Journal of Chemical Physics 18, no. 4 (April 1, 1950): 517-19. https://doi.org/10.1063/1.1747673.

[1] Wilke, C. R. "A Viscosity Equation for Gas Mixtures." The Journal of Chemical Physics 18, no. 4 (April 1, 1950): 517-19. https://doi.org/10.1063/1.1747673.

[1] Huber, M. L., R. A. Perkins, A. Laesecke, D. G. Friend, J. V. Sengers, M. J. Assael, I. N. Metaxa, E. Vogel, R. Mares, and K. Miyagawa. "New International Formulation for the Viscosity of H2O." Journal of Physical and Chemical Reference Data 38, no. 2 (June 1, 2009): 101-25. doi:10.1063/1.3088050.

[1] Lemmon, E. W., and R. T. Jacobsen. "Viscosity and Thermal Conductivity Equations for Nitrogen, Oxygen, Argon, and Air." International Journal of Thermophysics 25, no. 1 (January 1, 2004): 21-69. https://doi.org/10.1023/B:IJOT.0000022327.04529.f3.

[1] Twu, Chorng H. "Internally Consistent Correlation for Predicting Liquid Viscosities of Petroleum Fractions." Industrial & Engineering Chemistry Process Design and Development 24, no. 4 (October 1, 1985): 1287-93. https://doi.org/10.1021/i200031a064.

[1] Lohrenz, John, Bruce G. Bray, and Charles R. Clark. "Calculating Viscosities of Reservoir Fluids From Their Compositions." Journal of Petroleum Technology 16, no. 10 (October 1, 1964): 1,171-1,176. https://doi.org/10.2118/915-PA.

[2] Whitson, Curtis H., and Michael R. Brulé. Phase Behavior. Henry L. Doherty Memorial Fund of AIME, Society of Petroleum Engineers, 2000.

[1] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[1] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-ViscosityG/PPDS5-ViscosityGas.htm.

[1] Viswanath, Dabir S., and G. Natarajan. Databook On The Viscosity Of Liquids. New York: Taylor & Francis, 1989

[1] Viswanath, Dabir S., and G. Natarajan. Databook On The Viscosity Of Liquids. New York: Taylor & Francis, 1989

[1] Viswanath, Dabir S., and G. Natarajan. Databook On The Viscosity Of Liquids. New York: Taylor & Francis, 1989

[1] Yaws, Carl L. Thermophysical Properties of Chemicals and Hydrocarbons, Second Edition. 2 edition. Amsterdam Boston: Gulf Professional Publishing, 2014.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-ViscositySatL/ViscosityL.htm.

[1] Hydraulic Institute. Hydraulic Institute Engineering Data Book. Cleveland, Ohio: Hydraulic Institute, 1990.

[2] Gardner/Sward. Paint Testing Manual. Physical and Chemical Examination of Paints, Varnishes, Lacquers, and Colors. 13th Edition. ASTM, 1972.

[3] Euverard, M. R., The Efflux Type Viscosity Cup. National Paint, Varnish, and Lacquer Association, 1948.

[4] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[5] ASTM. Standard Practice for Conversion of Kinematic Viscosity to Saybolt Universal Viscosity or to Saybolt Furol Viscosity. D 2161 - 93.

[1] ASTM D2270-10(2016) Standard Practice for Calculating Viscosity Index from Kinematic Viscosity at 40 °C and 100 °C, ASTM International, West Conshohocken, PA, 2016, http://dx.doi.org/10.1520/D2270-10R16

[1] Viswanath, Dabir S., and G. Natarajan. Databook On The Viscosity Of Liquids. New York: Taylor & Francis, 1989

[2] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[3] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] Rackett, Harold G. "Equation of State for Saturated Liquids." Journal of Chemical & Engineering Data 15, no. 4 (1970): 514-517. doi:10.1021/je60047a012

[1] Hankinson, Risdon W., and George H. Thomson. "A New Correlation for Saturated Densities of Liquids and Their Mixtures." AIChE Journal 25, no. 4 (1979): 653-663. doi:10.1002/aic.690250412

[1] Yen, Lewis C., and S. S. Woods. "A Generalized Equation for Computer Calculation of Liquid Densities." AIChE Journal 12, no. 1 (1966): 95-99. doi:10.1002/aic.690120119

[1] Gunn, R. D., and Tomoyoshi Yamada. "A Corresponding States Correlation of Saturated Liquid Volumes." AIChE Journal 17, no. 6 (1971): 1341-45. doi:10.1002/aic.690170613

[2] Yamada, Tomoyoshi, and Robert D. Gunn. "Saturated Liquid Molar Volumes. Rackett Equation." Journal of Chemical & Engineering Data 18, no. 2 (1973): 234-36. doi:10.1021/je60057a006

[1] Hales, J. L, and R Townsend. "Liquid Densities from 293 to 490 K of Nine Aromatic Hydrocarbons." The Journal of Chemical Thermodynamics 4, no. 5 (1972): 763-72. doi:10.1016/0021-9614(72)90050-X

[1] Bhirud, Vasant L. "Saturated Liquid Densities of Normal Fluids." AIChE Journal 24, no. 6 (November 1, 1978): 1127-31. doi:10.1002/aic.690240630

[1] Campbell, Scott W., and George Thodos. "Prediction of Saturated Liquid Densities and Critical Volumes for Polar and Nonpolar Substances." Journal of Chemical & Engineering Data 30, no. 1 (January 1, 1985): 102-11. doi:10.1021/je00039a032.

[1] Mchaweh, A., A. Alsaygh, Kh. Nasrifar, and M. Moshfeghian. "A Simplified Method for Calculating Saturated Liquid Densities." Fluid Phase Equilibria 224, no. 2 (October 1, 2004): 157-67. doi:10.1016/j.fluid.2004.06.054

[1] Thomson, G. H., K. R. Brobst, and R. W. Hankinson. "An Improved Correlation for Densities of Compressed Liquids and Liquid Mixtures." AIChE Journal 28, no. 4 (July 1, 1982): 671-76. doi:10.1002/aic.690280420

[1] Rackett, Harold G. "Equation of State for Saturated Liquids." Journal of Chemical & Engineering Data 15, no. 4 (1970): 514-517. doi:10.1021/je60047a012

[2] Danner, Ronald P, and Design Institute for Physical Property Data. Manual for Predicting Chemical Process Design Data. New York, N.Y, 1982.

[1] Hankinson, Risdon W., and George H. Thomson. "A New Correlation for Saturated Densities of Liquids and Their Mixtures." AIChE Journal 25, no. 4 (1979): 653-663. doi:10.1002/aic.690250412

[1] Goodman, Benjamin T., W. Vincent Wilding, John L. Oscarson, and Richard L. Rowley. "A Note on the Relationship between Organic Solid Density and Liquid Density at the Triple Point." Journal of Chemical & Engineering Data 49, no. 6 (2004): 1512-14. doi:10.1021/je034220e.

[1] Frenkel, Michael, Robert D. Chirico, Vladimir Diky, Xinjian Yan, Qian Dong, and Chris Muzny. "ThermoData Engine (TDE): Software Implementation of the Dynamic Data Evaluation Concept." Journal of Chemical Information and Modeling 45, no. 4 (July 1, 2005): 816-38. https://doi.org/10.1021/ci050067b.

[2] Yaws, Carl L. "Liquid Density of the Elements: A Comprehensive Tabulation for All the Important Elements from Ag to Zr." Chemical Engineering 114, no. 12 (2007): 44-47.

[1] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/Help/TDE103b/Eqns-Pure-DensityLG/VDNSExpansion.htm.

[1] "ThermoData Engine (TDE103b V10.1) User`s Guide." https://trc.nist.gov/TDE/TDE_Help/Eqns-Pure-DensityLG/PPDS17.htm.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics, 95E. [Boca Raton, FL]: CRC press, 2014.

[1] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, 8E. McGraw-Hill Professional, 2007.

[2] Mchaweh, A., A. Alsaygh, Kh. Nasrifar, and M. Moshfeghian. "A Simplified Method for Calculating Saturated Liquid Densities." Fluid Phase Equilibria 224, no. 2 (October 1, 2004): 157-67. doi:10.1016/j.fluid.2004.06.054

[3] Hankinson, Risdon W., and George H. Thomson. "A New Correlation for Saturated Densities of Liquids and Their Mixtures." AIChE Journal 25, no. 4 (1979): 653-663. doi:10.1002/aic.690250412

[4] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[5] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

PYTHON MODULE INDEX

C

INDEX

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y

Z