thermo Documentation

Release 0.2.20

Caleb Bell

Apr 17, 2022

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION TO CUBIC EQUATIONS OF STATE

Cubic equations of state provide thermodynamically-consistent and relatively fast models for pure chemicals and mixtures. They are normally used to represent gases and liquids.

The generic three-parameter form is as follows:

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V^2 + \delta V + \epsilon}$$

This forms the basis of the implementation in *thermo*.

Two separate interfaces are provided, `thermo.eos` for pure component modeling and `thermo.eos_mix` for multicomponent modeling. Pure components are quite a bit faster than multicomponent mixtures, because the Van der Waals mixing rules conventionally used take N^2 operations to compute $\alpha(T)$:

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

The other slow parts which applies to both types are calculating some basic properties (the list is at `set_properties_from_solution`) that other properties may depend on, and calculating the molar volume given a pair of (*T*, *P*) inputs (an entire submodule `thermo.eos_volume` discusses and implements this topic). Both of those calculations are constant-time, so their overhead is the same for pure components and multicomponent mixtures.

1.1 Working With Pure Components

We can use the *GCEOS* (short for "General Cubic Equation Of State") interface with any component or implemented equation of state, but for simplicity n-hexane is used with the Peng-Robinson EOS. Its critical temperature is 507.6 K, critical pressure 3.025 MPa, and acentric factor is 0.2975.

The state must be specified along with the critical constants when initializing a *GCEOS* object; we use 400 K and 1e6 Pa here:

```
>>> from thermo import *
>>> eos = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=400., P=1E6)
>>> eos
PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=400.0, P=1000000.0)
```

The *__repr__* string is designed to show all the inputs to the object.

We can check the volume solutions with the `raw_volumes` attribute:

```
>>> eos.raw_volumes
(0.0001560731847856, 0.002141876816741, 0.000919295474982)
```

At this point there are three real volume, so there is a liquid-like and a vapor-like solution available. The `phase` attribute will have the value of 'l/g' in this state; otherwise it will be 'l' or 'g'.

```
>>> eos.phase
'l/g'
```

The basic properties calculated at initialization are directly attributes, and can be accessed as such. Liquid-like properties have "_l" at the end of their name, and "_g" is at the end of gas-like properties.

```
>>> eos.H_dep_l
-26111.877
>>> eos.S_dep_g
-6.4394518
>>> eos.dP_dT_l
288501.633
```

All calculations in *thermo.eos* and *thermo.eos_mix* are on a molar basis; molecular weight is never provided or needed. All outputs are in base SI units (K, Pa, m^3, mole, etc). This simplified development substantially. For working with mass-based units, use the *Phase* interface. The *thermo.eos* and *thermo.eos_mix* interfaces were developed prior to the *Phase* interface and does have some features not exposed in the *Phase* interface however.

Other properties are either implemented as methods that require arguments, or Python properties which act just like attributes but calculate the results on the fly. For example, the liquid-phase fugacity *fugacity_l* or the gas isobaric (constant-pressure) expansion coefficient are properties.

```
>>> eos.fugacity_l
421597.00785
>>> eos.beta_g
0.0101232239
```

There are an awful lot of these properties, because many of them are derivatives subject to similar conditions. A full list is in the documentation for *GCEOS*. There are fewer calls that take temperature, such as *Hvap* which calculates the heat of vaporization of the object at a specified temperature:

```
>>> eos.Hvap(300)
31086.2
```

Once an object has been created, it can be used to instantiate new *GCEOS* objects at different conditions, without re-specifying the critical constants and other parameters that may be needed.

```
>>> eos.to(T=300.0, P=1e5)
PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=300.0, P=100000.0)
>>> eos.to(V=1e2, P=1e5)
PR(Tc=507.6, Pc=3025000.0, omega=0.2975, P=100000.0, V=100.0)
>>> eos.to(V=1e2, T=300)
PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=300, V=100.0)
```

As was seen in the examples above, any two of T, P, V can be used to specify the state of the object. The input variables of the object are stored and can be checked with *state_specs* :

```
>>> eos.state_specs
{'T': 400.0, 'P': 1000000.0}
```

The individual parts of the generic cubic equation are stored as well. We can use them to check that the pressure equation is satisfied:

```
>>> from thermo.eos import R
>>> R*eos.T/(eos.V_l-eos.b) - eos.a_alpha/(eos.V_l**2 + eos.V_l*eos.delta + eos.epsilon)
1000000.000000
>>> R*eos.T/(eos.V_g-eos.b) - eos.a_alpha/(eos.V_g**2 + eos.V_g*eos.delta + eos.epsilon)
1000000.000000
```

Note that as floating points are not perfectly precise, some small error may be shown but great care has been taken to minimize this.

The value of the gas constant used is 8.31446261815324 J/(mol*K). This is near the full precision of floating point numbers, but not quite. It is now an exact value used as a “definition” in the SI system. Note that other implementations of equations of state may not use the full value of the gas constant, but the author strongly recommends anyone considering writing their own EOS implementation use the full gas constant. This will allow more interchangeable results.

1.2 Pure Component Equilibrium

Continuing with the same state and example as before, there were two solutions available from the equation of state. However, unless the exact temperature 400 K and pressure 1 MPa happens to be on the saturation line, there is always one more thermodynamically stable state. We need to use the departure Gibbs free energy to determine which state is more stable. For a pure component, the state which minimizes departure Gibbs free energy is the most stable state.

```
>>> eos = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=400., P=1E6)
>>> eos.G_dep_l, eos.G_dep_g
(-2872.498434, -973.5198207)
```

It is easy to see the liquid phase is more stable. This shortcut of using departure Gibbs free energy is valid only for pure components with all phases using the ideal-gas reference state. The full criterial is whichever state minimizes the actual Gibbs free energy.

The method *more_stable_phase* does this check and returns either ‘l’ or ‘g’:

```
>>> eos.more_stable_phase
'l'
```

For a pure component, there is a vapor-liquid equilibrium line right up to the critical point which defines the vapor pressure of the fluid. This can be calculated using the `Psat` method:

```
>>> eos.Psat(400.0)
466205.073739
```

The result is accurate to more than 10 digits, and is implemented using some fancy mathematical techniques that allow a direct calculation of the vapor pressure. A few more digits can be obtained by setting *polish* to True, which polishes the result with a newton solver to as much accuracy as a floating point number can provide:

```
>>> 1-eos.Psat(400, polish=True)/eos.Psat(400)
1.6e-14
```

A few more methods of interest are `V_l_sat` and `V_g_sat` which calculate the saturation liquid and molar volumes; `Tsat` which calculates the saturation temperature given a specified pressure, and `phi_sat` which computes the saturation fugacity coefficient given a temperature.

```
>>> eos.V_l_sat(298.15), eos.V_g_sat(500)
(0.0001303559, 0.0006827569)
>>> eos.Tsat(101325.0)
341.76265
>>> eos.phi_sat(425.0)
0.8349716
```

1.3 Working With Mixtures

Using mixture from `thermo.eos_mix` is first illustrated using an equimolar mixture of nitrogen-methane at 115 K and 1 MPa and the Peng-Robinson equation of state:

```
>>> eos = PRMIX(T=115.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], omegas=[0.04,
↪ 0.011], zs=[0.5, 0.5], kijs=[[0.0, 0.0289], [0.0289, 0.0]])
>>> eos.V_l, eos.V_g
(3.658707770e-05, 0.00070676607)
>>> eos.fugacities_l, eos.fugacities_g
([838516.99, 78350.27], [438108.61, 359993.48])
```

All of the properties available in `GCEOS` are also available for `GCEOSMIX` objects.

New `GCEOSMIX` objects can be created with the `to` method, which accepts new mole fractions *zs* as well as new state variables. If a new composition *zs* is not provided, the current composition is also used for the new object.

```
>>> eos.to(T=300.0, P=1e5)
PRMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011], kijs=[[0.0,␣
↪0.0289], [0.0289, 0.0]], zs=[0.5, 0.5], T=300.0, P=100000.0)
>>> eos.to(T=300.0, P=1e5, zs=[.1, .9])
PRMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011], kijs=[[0.0,␣
↪0.0289], [0.0289, 0.0]], zs=[0.1, 0.9], T=300.0, P=100000.0)
>>> eos.to(V=1, P=1e5, zs=[.4, .6])
PRMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011], kijs=[[0.0,␣
↪0.0289], [0.0289, 0.0]], zs=[0.4, 0.6], P=100000.0, V=1)
```

(continues on next page)

(continued from previous page)

```
>>> eos.to(V=1.0, T=300.0, zs=[.4, .6])
PRMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011], kijs=[[0.0,␣
↪0.0289], [0.0289, 0.0]], zs=[0.4, 0.6], T=300.0, V=1.0)
```

It is possible to create new *GCEOSMIX* objects with the *subset* method which uses only some of the initially specified components:

```
>>> kijs = [[0.0, 0.00076, 0.00171], [0.00076, 0.0, 0.00061], [0.00171, 0.00061, 0.0]]
>>> PR3 = PRMIX(Tcs=[469.7, 507.4, 540.3], zs=[0.8168, 0.1501, 0.0331], omegas=[0.249, 0.
↪305, 0.349], Pcs=[3.369E6, 3.012E6, 2.736E6], T=322.29, P=101325.0, kijs=kijs)
>>> PR3.subset([1,2])
PRMIX(Tcs=[507.4, 540.3], Pcs=[3012000.0, 2736000.0], omegas=[0.305, 0.349], kijs=[[0.0,␣
↪0.00061], [0.00061, 0.0]], zs=[0.8193231441048, 0.1806768558951], T=322.29, P=101325.0)
>>> PR3.subset([1,2], T=500.0, P=1e5, zs=[.2, .8])
PRMIX(Tcs=[507.4, 540.3], Pcs=[3012000.0, 2736000.0], omegas=[0.305, 0.349], kijs=[[0.0,␣
↪0.00061], [0.00061, 0.0]], zs=[0.2, 0.8], T=500.0, P=100000.0)
>>> PR3.subset([1,2], zs=[.2, .8])
PRMIX(Tcs=[507.4, 540.3], Pcs=[3012000.0, 2736000.0], omegas=[0.305, 0.349], kijs=[[0.0,␣
↪0.00061], [0.00061, 0.0]], zs=[0.2, 0.8], T=322.29, P=101325.0)
```

It is also possible to create pure *GCEOS* objects:

```
>>> PR3.pures()
[PR(Tc=469.7, Pc=3369000.0, omega=0.249, T=322.29, P=101325.0), PR(Tc=507.4, Pc=3012000.
↪0, omega=0.305, T=322.29, P=101325.0), PR(Tc=540.3, Pc=2736000.0, omega=0.349, T=322.
↪29, P=101325.0)]
```

Temperature, pressure, mole number, and mole fraction derivatives of the log fugacity coefficients are available as well with the methods *dlnphis_dT*, *dlnphis_dP*, *dlnphis_dns*, and *dlnphis_dzs*:

```
>>> PR3.dlnphis_dT('l')
[0.029486952019, 0.03514175794, 0.040281845273]
>>> PR3.dlnphis_dP('l')
[-9.8253779e-06, -9.8189093031e-06, -9.8122598e-06]
>>> PR3.dlnphis_dns(PR3.Z_l)
[[-0.0010590517, 0.004153228837, 0.007300114797], [0.0041532288, -0.016918292791, -0.
↪0257680231], [0.0073001147, -0.02576802316, -0.0632916462]]
>>> PR3.dlnphis_dzs(PR3.Z_l)
[[0.0099380692, 0.0151503498, 0.0182972357], [-0.038517738, -0.059589260, -0.068438990],␣
↪[-0.070571069, -0.103639207, -0.141162830]]
```

1.4 Other features

1.4.1 Hashing

It is possible to compare the two objects with each other to see if they have the same kijs, model parameters, and components by using the *model_hash* method:

```
>>> PR_case = PRMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], omegas=[0.
↪04, 0.011], zs=[0.5, 0.5], kijs=[[0,0.41],[0.41,0]])
>>> SRK_case = SRKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0.41],[0.41,0]])
```

```
>>> PR_case.model_hash() == SRK_case.model_hash()
False
```

It is possible to see if both the exact state and the model match between two different objects by using the *state_hash* method:

```
>>> PR_case2 = PRMIX(T=116, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], omegas=[0.
↪04, 0.011], zs=[0.5, 0.5], kijs=[[0,0.41],[0.41,0]])
>>> PR_case.model_hash() == PR_case2.model_hash()
True
>>> PR_case.state_hash() == PR_case2.state_hash()
False
```

And finally it is possible to see if two objects are exactly identical, including cached calculation results, by using the `__hash__` method:

```
>>> PR_case3 = PRMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], omegas=[0.
↪04, 0.011], zs=[0.5, 0.5], kijs=[[0,0.41],[0.41,0]])
>>> PR_case.state_hash() == PR_case3.state_hash()
True
>>> hash(PR_case) == hash(PR_case3)
True
>>> _ = PR_case.da_alpha_dT_ijs
>>> hash(PR_case) == hash(PR_case3)
False
```

1.4.2 Serialization

All cubic EOS models offer a *as_json* method and a *from_json* to serialize the object state for transport over a network, storing to disk, and passing data between processes.

```
>>> import json
>>> eos = PRSV2MIX(Tcs=[507.6], Pcs=[3025000], omegas=[0.2975], zs=[1], T=299., P=1E6,␣
↪kappa1s=[0.05104], kappa2s=[0.8634], kappa3s=[0.460])
>>> json_stuff = json.dumps(eos.as_json())
>>> new_eos = GCEOSMIX.from_json(json.loads(json_stuff))
>>> assert new_eos == eos
```

Other json libraries can be used besides the standard json library by design.

Storing and recreating objects with Python's `pickle.dumps` library is also tested; this can be faster than using JSON at the cost of being binary data.

1.5 Mixture Equilibrium

Unlike pure components, it is not straightforward to determine what the equilibrium state is for mixtures. Different algorithms are used such as sequential substitution and Gibbs minimization. All of those require initial guesses, which usually come from simpler thermodynamic models. While in practice it is possible to determine the equilibrium composition to an N-phase problem, in theory a global optimization algorithm must be used.

More details on this topic can be found in the *thermo.flash* module.

1.6 Using Units with Cubic Equations of State

There is a pint wrapper to use these objects as well.

```
>>> from thermo.units import *
>>> kwargs = dict(T=400.0*u.degC, P=30*u.psi, Tcs=[126.1, 190.6]*u.K, Pcs=[33.94E5, 46.
↪04E5]*u.Pa, omegas=[0.04, 0.011]*u.dimensionless, zs=[0.5, 0.5]*u.dimensionless,␣
↪kijs=[[0.0, 0.0289], [0.0289, 0.0]]*u.dimensionless)
>>> eos_units = PRMIX(**kwargs)
>>> eos_units.H_dep_g, eos_units.T
(<Quantity(-2.53858854, 'joule / mole')>, <Quantity(673.15, 'kelvin')>)
```

```
>>> base = IG(T=300.0*u.K, P=1e6*u.Pa)
>>> base.V_g
<Quantity(0.00249433879, 'meter ** 3 / mole')>
```

CHAPTER

TWO

INTRODUCTION TO ACTIVITY COEFFICIENT MODELS

Vapor-liquid and liquid-liquid equilibria systems can have all sorts of different behavior. Raoult's law can describe only temperature and pressure dependence, so a correction factor that adds dependence on composition called the "activity coefficient" is often used. This is a separate approach to using an equation of state, but because direct vapor pressure correlations are used with the activity coefficients, a higher-accuracy result can be obtained for phase equilibria.

While these models are often called "activity coefficient models", they are in fact actually a prediction for excess Gibbs energy. The activity coefficients that are used for phase equilibria are derived from the partial mole number derivative of excess Gibbs energy according to the following expression:

$$\gamma_i = \exp\left(\frac{\frac{\partial n_i G^E}{\partial n_i}}{RT}\right)$$

There are 5 basic activity coefficient models in thermo:

- `NRTL`
- `Wilson`
- `UNIQUAC`
- `RegularSolution`
- `UNIFAC`

Each of these models are object-oriented, and inherit from a base class `GibbsExcess` that provides many common methods. A further dummy class that predicts zero excess Gibbs energy and activity coefficients of 1 is available as `IdealSolution`.

The excess Gibbs energy model is typically fairly simple. A number of derivatives are needed to calculate other properties like activity coefficient so those expressions can seem more complicated than the model really is. In the literature it is common for a model to be shown directly in activity coefficient form without discussion of the Gibbs excess energy model. To illustrate the difference, here is the `NRTL` model Gibbs energy expression and its activity coefficient model:

$$g^E = RT\sum_i x_i \frac{\sum_j \tau_{ji} G_{ji} x_j}{\sum_j G_{ji} x_j}$$

$$\ln(\gamma_i) = \frac{\sum_{j=1}^{n} x_j \tau_{ji} G_{ji}}{\sum_{k=1}^{n} x_k G_{ki}} + \sum_{j=1}^{n} \frac{x_j G_{ij}}{\sum_{k=1}^{n} x_k G_{kj}} \left(\tau_{ij} - \frac{\sum_{m=1}^{n} x_m \tau_{mj} G_{mj}}{\sum_{k=1}^{n} x_k G_{kj}} \right)$$

The models *NRTL*, *Wilson*, and *UNIQUAC* are the most commonly used. Each of them is regression-based - all coefficients must be found in the literature or regressed yourself. Each of these models has extensive temperature dependence parameters in addition to the composition dependence. The temperature dependencies implemented should allow parameters from most other sources to be used here with them.

The model *RegularSolution* is based on the concept of a `solubility parameter`; with liquid molar volumes and solubility parameters it is a predictive model. It does not show temperature dependence. Additional regression coefficients can be used with that model also.

The *UNIFAC* model is a predictive group-contribution scheme. In it, each molecule is fragmented into different sections. These sections have interaction parameters with other sections. Usually the fragmentation is not done by hand. One online tool for doing this is the DDBST Online Group Assignment Tool.

2.1 Object Structure

The *GibbsExcess* object doesn't know anything about phase equilibria, vapor pressure, or flash routines; it is limited in scope to dealing with excess Gibbs energy. Because of that modularity, an initialized *GibbsExcess* object is designed to be passed in an argument to a cubic equations of state that use excess Gibbs energy such as *PSRK*.

The other place these objects are used are in *GibbsExcessLiquid* objects, which brings the pieces together to construct a thermodynamically (mostly) consistent phase that the *flash algorithms* can work with.

This modularity allows new Gibbs excess models to be written and used anywhere - so the *PSRK* model will happily allow a UNIFAC object configured like VTPR.

2.2 UNIFAC Example

The UNIFAC model is a group contribution based predictive model that is works using "fragmentations" of each molecule into a number of different "groups" and their "counts",

The DDBST has published numerous sample problems using UNIFAC; a simple binary system from example P05.22a in[2] with n-hexane and butanone-2 is shown below:

```
>>> from thermo.unifac import UFIP, UFSG, UNIFAC
>>> GE = UNIFAC.from_subgroups(chemgroups=[{1:2, 2:4}, {1:1, 2:1, 18:1}], T=60+273.15,␣
↪xs=[0.5, 0.5], version=0, interaction_data=UFIP, subgroups=UFSG)
```

The solution given by the DDBST has the activity coefficient values [1.428, 1.365], which match those calculated by the UNIFAC object:

```
>>> GE.gammas()
[1.4276025835, 1.3646545010]
```

Many other properties are also implemented, a few of which are shown below:

[2] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

```
>>> GE.GE(), GE.dGE_dT(), GE.d2GE_dT2()
(923.641197, 0.206721488, -0.00380070204)
>>> GE.HE(), GE.SE(), GE.dHE_dT(), GE.dSE_dT()
(854.77193363, -0.2067214889, 1.266203886, 0.0038007020460)
```

Note that the *UFIP* and *UFSG* variables contain the actual interaction parameters; none are hardcoded with the class, so the class could be used for regression. The *version* parameter controls which variant of UNIFAC to use, as there are quite a few. The different UNIFAC models implemented include original UNIFAC, Dortmund UNIFAC, PSRK, VTPR, Lyngby/Larsen, and UNIFAC KT. Interaction parameters for all models are included as well, but the *version* argument is not connected to the data files.

For convenience, a number of molecule fragmentations are distributed with the UNIFAC code. All fragmentations were obtained through the DDBST online portal, where molecular structure files can be submitted. This has the advantage that what is submitted is unambiguous; there are no worries about CAS numbers like how graphite and diamond have a different CAS number while being the same element or Air having a CAS number despite being a mixture. Accordingly, The index in these distributed data files are InChI keys, which can be obtained from `chemicals.identifiers` or in various places online.

```
>>> import thermo.unifac
>>> thermo.unifac.load_group_assignments_DDBST()
>>> len(thermo.unifac.DDBST_UNIFAC_assignments)
28846
>>> len(thermo.unifac.DDBST_MODIFIED_UNIFAC_assignments)
29271
>>> len(thermo.unifac.DDBST_PSRK_assignments)
30034
>>> from chemicals import search_chemical
>>> search_chemical('toluene').InChI_key
'YXFVVABEGXRONW-UHFFFAOYSA-N'
>>> thermo.unifac.DDBST_MODIFIED_UNIFAC_assignments['YXFVVABEGXRONW-UHFFFAOYSA-N']
{9: 5, 11: 1}
```

Please note that the identifying integer in these {group: count} elements are not necessarily the same in different UNIFAC versions, making them a royal pain.

2.3 Notes on Performance

Initializing the object for the first time is a not a high performance operation as certain checks need to be done and data structures set up. Some pieces of the equations of the Gibbs excess model may depend only on temperature or composition, instead of depending on both. Each model implements the method *to_T_xs* which should be used to create a new object at the new temperature and/or composition. The design of the object is to lazy-calculate properties, and to be immutable: calculations at new temperatures and compositions are done in a new object.

Note also that the *__repr__* string for each model is designed to allow lossless reconstruction of the model. This is very useful when building test cases.

```
>>> GE.to_T_xs(T=400.0, xs=[.1, .9])
UNIFAC(T=400.0, xs=[0.1, 0.9], rs=[4.4998000000000005, 3.2479], qs=[3.856, 2.876], Qs=[0.
↪848, 0.54, 1.488], vs=[[2, 1], [4, 1], [0, 1]], psi_abc=([[0.0, 0.0, 476.4], [0.0, 0.0,
↪ 476.4], [26.76, 26.76, 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]]), version=0)
```

When working with small numbers of components (5 or under), PyPy offers the best performance and using the model with Python lists as inputs is the fastest way to perform the calculations even in CPython.

If working with many components or if Numpy arrays are desired as inputs and outputs, numpy arrays can be provided as inputs. This will have a negative impact on performance unless the *numba* interface is used:

```
>>> import numpy as np
>>> import thermo.numba
>>> N = 3
>>> T = 25.0 + 273.15
>>> xs = np.array([0.7273, 0.0909, 0.1818])
>>> rs = np.array([.92, 2.1055, 3.1878])
>>> qs = np.array([1.4, 1.972, 2.4])
>>> tausA = tausC = tausD = tausE = tausF = np.array([[0.0]*N for i in range(N)])
>>> tausB = np.array([[0, -526.02, -309.64], [318.06, 0, 91.532], [-1325.1, -302.57, 0]])
>>> ABCDEF = (tausA, tausB, tausC, tausD, tausE, tausF)
>>> from thermo import UNIQUAC
>>> GE2 = UNIQUAC(T=T, xs=xs, rs=rs, qs=qs, ABCDEF=ABCDEF)
>>> GE2.gammas()
array([ 1.57039333,  0.29482416, 18.11432905])
```

The *numba* interface will speed code up and allow calculations with dozens of components. The *numba* interface requires all inputs to be numpy arrays and all of its outputs are also numba arrays.

```
>>> GE3 = thermo.numba.UNIQUAC(T=T, xs=xs, rs=rs, qs=qs, ABCDEF=ABCDEF)
>>> GE3.gammas()
array([ 1.57039333,  0.29482416, 18.11432905])
```

As an example of the performance benefits, a 200-component UNIFAC gamma calculation takes 10.6 ms in CPython and 318 µs when accelerated by Numba. In this case PyPy takes at 664 µs.

When the same benchmark is performed with 10 components, the calculation takes 387 µs in CPython, 88.6 µs with numba, and 36.2 µs with PyPy.

It can be quite important to use the `to_T_xs` method re-use parts of the calculation; for UNIFAC, several terms depends only on temperature. If the 200 component calculation is repeated with those already calculated, the timings are 3.26 ms in CPython, 127 µs with numba, and 125 µs with PyPy.

2.4 Other features

The limiting infinite-dilution activity coefficients can be obtained with a call to `gammas_infinite_dilution`

```
>>> GE.gammas_infinite_dilution()
[3.5659995166, 4.32849696]
```

All activity coefficient models offer a `as_json` method and a `from_json` to serialize the object state for transport over a network, storing to disk, and passing data between processes.

```
>>> from thermo import IdealSolution
>>> import json
>>> model = IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
>>> json_view = model.as_json()
>>> json_str = json.dumps(json_view)
```

(continues on next page)

(continued from previous page)

```
>>> model_copy = IdealSolution.from_json(json.loads(json_str))
>>> assert model_copy == model
```

Other json libraries can be used besides the standard json library by design.

Storing and recreating objects with Python's `pickle.dumps` library is also tested; this can be faster than using JSON at the cost of being binary data.

All models have a `__hash__` method that can be used to compare different models to see if they are absolutely identical (including which values have been calculated already).

They also have a `model_hash` method that can be used to compare different models to see if they have identical model parameters.

They also have a `state_hash` method that can be used to compare different models to see if they have identical temperature, composition, and model parameters.

2.5 Activity Coefficient Identities

A set of useful equations are as follows. For more information, the reader is directed to[1],[?],[3],[4], and[5]; no one source contains all this information.

$$h^E = -T\frac{\partial g^E}{\partial T} + g^E$$

$$\frac{\partial h^E}{\partial T} = -T\frac{\partial^2 g^E}{\partial T^2}$$

$$\frac{\partial h^E}{\partial x_i} = -T\frac{\partial^2 g^E}{\partial T\partial x_i} + \frac{\partial g^E}{\partial x_i}$$

$$s^E = \frac{h^E - g^E}{T}$$

$$\frac{\partial s^E}{\partial T} = \frac{1}{T}\left(\frac{-\partial g^E}{\partial T} + \frac{\partial h^E}{\partial T} - \frac{(G+H)}{T}\right)$$

$$\frac{\partial S^E}{\partial x_i} = \frac{1}{T}\left(\frac{\partial h^E}{\partial x_i} - \frac{\partial g^E}{\partial x_i}\right)$$

$$\frac{\partial \gamma_i}{\partial n_i} = \gamma_i\left(\frac{\frac{\partial^2 G^E}{\partial x_i\partial x_j}}{RT}\right)$$

$$\frac{\partial \gamma_i}{\partial T} = \left(\frac{\frac{\partial^2 nG^E}{\partial T\partial n_i}}{RT} - \frac{\frac{\partial n_i G^E}{\partial n_i}}{RT^2}\right)\exp\left(\frac{\frac{\partial n_i G^E}{\partial n_i}}{RT}\right)$$

[1] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Nevers, Noel de. Physical and Chemical Equilibrium for Chemical Engineers. 2nd edition. Wiley, 2012.

[4] Elliott, J., and Carl Lira. Introductory Chemical Engineering Thermodynamics. 2nd edition. Upper Saddle River, NJ: Prentice Hall, 2012.

[5] Walas, Dr Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

2.6 References

CHAPTER

THREE

INTRODUCTION TO PROPERTY OBJECTS

For every chemical property, there are lots and lots of methods. The methods can be grouped by which phase they apply to, although some methods are valid for both liquids and gases.

Properties calculations be separated into three categories:

- Properties of chemicals that depend on **temperature**. Some properties have weak dependence on pressure, like surface tension, and others have no dependence on pressure like vapor pressure by definition.
- Properties of chemicals that depend on **temperature** and **pressure**. Some properties have weak dependence on pressure like thermal conductivity, while other properties depend on pressure fundamentally, like gas volume.
- Properties of mixtures, that depend on **temperature** and **pressure** and **composition**. Some properties like gas mixture heat capacity require the pressure as an input but do not use it.

These properties are implemented in an object oriented way, with the actual functional algorithms themselves having been separated out into the chemicals library. The goal of these objects is to make it easy to experiment with different methods.

The base classes for the three respective types of properties are:

- `TDependentProperty`
- `TPDependentProperty`
- `MixtureProperty`

The specific classes for the three respective types of properties are:

- `HeatCapacityGas`, `HeatCapacityLiquid`, `HeatCapacitySolid`, `VolumeSolid`, `VaporPressure`, `SublimationPressure`, `EnthalpyVaporization`, `EnthalpySublimation`, `Permittivity`, `SurfaceTension`.
- `VolumeGas`, `VolumeLiquid`, `ViscosityGas`, `ViscosityLiquid`, `ThermalConductivityGas`, `ThermalConductivityLiquid`
- `HeatCapacityGasMixture`, `HeatCapacityLiquidMixture`, `HeatCapacitySolidMixture`, `VolumeGasMixture`, `VolumeLiquidMixture`, `VolumeSolidMixture`, `ViscosityLiquidMixture`, `ViscosityGasMixture`, `ThermalConductivityLiquidMixture`, `ThermalConductivityGasMixture`, `SurfaceTensionMixture`

3.1 Temperature Dependent Properties

The following examples introduce how to use some of the methods of the `TDependentProperty` objects. The API documentation for `TDependentProperty` as well as each specific property such as `VaporPressure` should be consulted for full details.

3.1.1 Creating Objects

All arguments and information the property object requires must be provided in the constructor of the object. If a piece of information is not provided, whichever methods require it will not be available for that object.

```
>>> from thermo import VaporPressure, HeatCapacityGas
>>> ethanol_psat = VaporPressure(Tb=351.39, Tc=514.0, Pc=6137000.0, omega=0.635, CASRN=
↪'64-17-5')
```

Various data files will be searched to see if information such as Antoine coefficients is available for the compound during the initialization. This behavior can be avoided by setting the optional *load_data* argument to False. Loading data requires *pandas*, uses more RAM, and is a once-per-process procedure that takes 20-1000 ms per property. For some applications it may be advantageous to provide your own data instead of using the provided data files.

```
>>> useless_psat = VaporPressure(CASRN='64-17-5', load_data=False)
```

3.1.2 Temperature-dependent Methods

As many methods may be available, a single method is always selected automatically during initialization. This method can be inspected with the *method* property; if no methods are available, *method* will be None. *method* is also a valid parameter when constructing the object, but if the method specified is not available an exception will be raised.

```
>>> ethanol_psat.method, useless_psat.method
('WAGNER_MCGARRY', None)
```

All available methods can be found by inspecting the `all_methods` attribute:

```
>>> ethanol_psat.all_methods
{'ANTOINE_POLING', 'EDALAT', 'WAGNER_POLING', 'SANJARI', 'COOLPROP', 'LEE_KESLER_PSAT',
↪'DIPPR_PERRY_8E', 'VDI_PPDS', 'WAGNER_MCGARRY', 'VDI_TABULAR', 'AMBROSE_WALTON',
↪'BOILING_CRITICAL'}
```

Changing the method is as easy as setting a new value to the attribute:

```
>>> ethanol_psat.method = 'ANTOINE_POLING'
>>> ethanol_psat.method
'ANTOINE_POLING'
>>> ethanol_psat.method = 'WAGNER_MCGARRY'
```

3.1.3 Calculating Properties

Calculation of the property at a specific temperature is as easy as calling the object which triggers the *__call__* method:

```
>>> ethanol_psat(300.0)
8753.8160
```

This is actually a cached wrapper around the specific call, *T_dependent_property*:

```
>>> ethanol_psat.T_dependent_property(300.0)
8753.8160
```

The caching of *__call__* is quite basic - the previously specified temperature is stored, and if the new T is the same as the previous T the previously calculated result is returned.

There is a lower-level interface for calculating properties with a specified method by name, *calculate*. *T_dependent_property* is a wrapper around *calculate* that includes validation of the result.

```
>>> ethanol_psat.calculate(T=300.0, method='WAGNER_MCGARRY')
8753.8160
>>> ethanol_psat.calculate(T=300.0, method='DIPPR_PERRY_8E')
8812.9812
```

3.1.4 Limits and Extrapolation

Each correlation is associated with temperature limits. These can be inspected as part of the *T_limits* attribute which is loaded on creation of the property object.

```
>>> ethanol_psat.T_limits
{'WAGNER_MCGARRY': (293.0, 513.92), 'WAGNER_POLING': (159.05, 513.92), 'ANTOINE_POLING':␣
↪(276.5, 369.54), 'DIPPR_PERRY_8E': (159.05, 514.0), 'COOLPROP': (159.1, 514.71), 'VDI_
↪TABULAR': (300.0, 513.9), 'VDI_PPDS': (159.05, 513.9), 'BOILING_CRITICAL': (0.01, 514.
↪0), 'LEE_KESLER_PSAT': (0.01, 514.0), 'AMBROSE_WALTON': (0.01, 514.0), 'SANJARI': (0.
↪01, 514.0), 'EDALAT': (0.01, 514.0)}
```

Because there is often a need to obtain a property outside the range of the correlation, there are some extrapolation methods available; depending on the method these may be enabled by default. The full list of extrapolation methods can be see *here*.

For vapor pressure, there are actually two separate extrapolation techniques used, one for the low-pressure and thermodynamically reasonable region and another for extrapolating even past the critical point. This can be useful for obtaining initial estimates of phase equilibrium.

The low-pressure region uses $\log(P_{sat}) = A - B/T$, where the coefficients A and B are calculated from the low-temperature limit and its temperature derivative. The default high-temperature extrapolation is $P_{sat} = \exp\left(A + B/T + C\log(T)\right)$. The coefficients are also determined from the high-temperature limits and its first two temperature derivatives.

When extrapolation is turned on, it is used automatically if a property is requested out of range:

```
>>> ethanol_psat(100.0), ethanol_psat(1000)
(1.047582e-11, 1779196575.4962692)
```

The default extrapolation methods may be changed in the future, but can be manually specified also by changing the value of the *extrapolation* attribute. For example, if the *linear* extrapolation method is set, extrapolation will be linear instead of using those fit equations. Because not all properties are suitable for linear extrapolation, some methods have a default *transform* to make the property behave as linearly as possible. This is also used in tabular interpolation:

```
>>> ethanol_psat.extrapolation = 'linear'
>>> ethanol_psat(100.0), ethanol_psat(1000)
(1.0475e-11, 385182009.4)
```

The low-temperature linearly extrapolated value is actually the same as before, because it performs a 1/T transform and a log(P) transform on the output, which results in the fit being the same as the default equation for vapor pressure.

To better understand what methods are available, the *valid_methods* method checks all available correlations against their temperature limits.

```
>>> ethanol_psat.valid_methods(100)
['AMBROSE_WALTON', 'LEE_KESLER_PSAT', 'EDALAT', 'BOILING_CRITICAL', 'SANJARI']
```

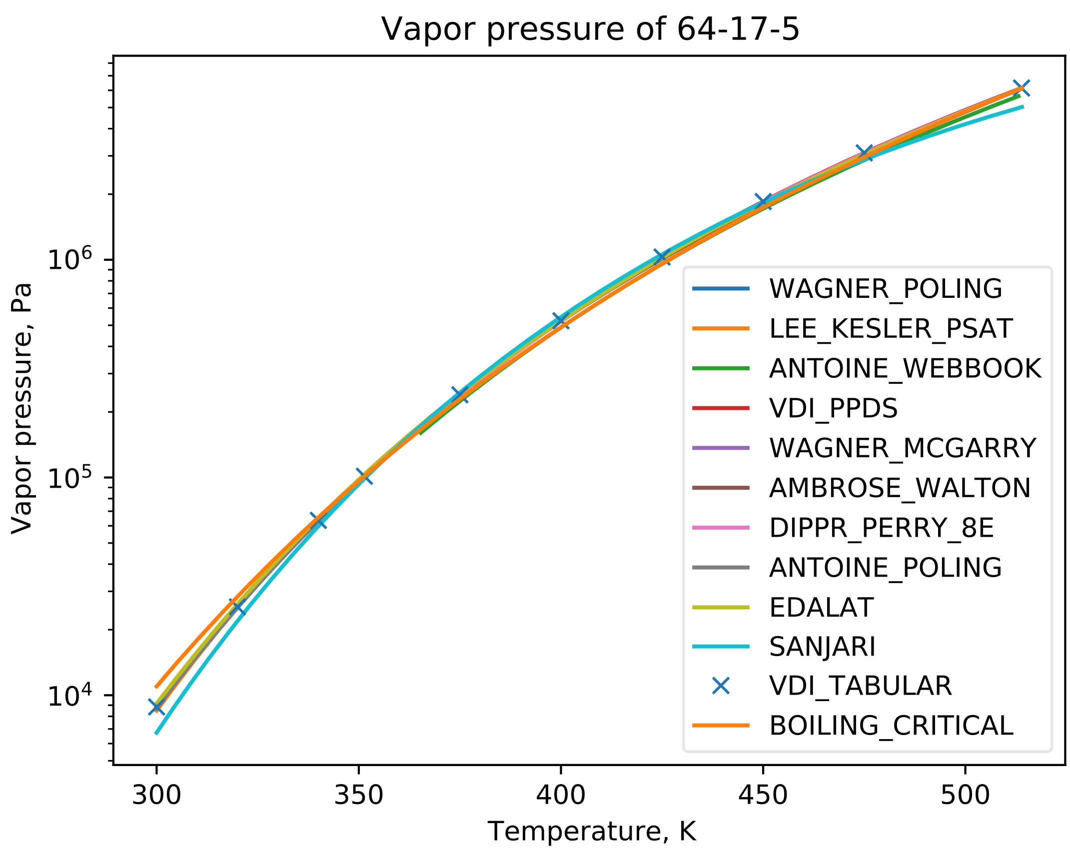
If the temperature is not provided, all available methods are returned; the returned value favors the methods by the ranking defined in thermo, with the currently selected method as the first item.

```
>>> ethanol_psat.valid_methods()
['WAGNER_MCGARRY', 'WAGNER_POLING', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'COOLPROP', 'ANTOINE_
↪POLING', 'VDI_TABULAR', 'AMBROSE_WALTON', 'LEE_KESLER_PSAT', 'EDALAT', 'BOILING_
↪CRITICAL', 'SANJARI']
```

3.1.5 Plotting

It is also possible to compare the correlations graphically with the method *plot_T_dependent_property*.

```
>>> ethanol_psat.plot_T_dependent_property(Tmin=300)
```



By default all methods are shown in the plot, but a smaller selection of methods can be specified. The following example compares 30 points in the temperature range 400 K to 500 K, with three of the best methods.

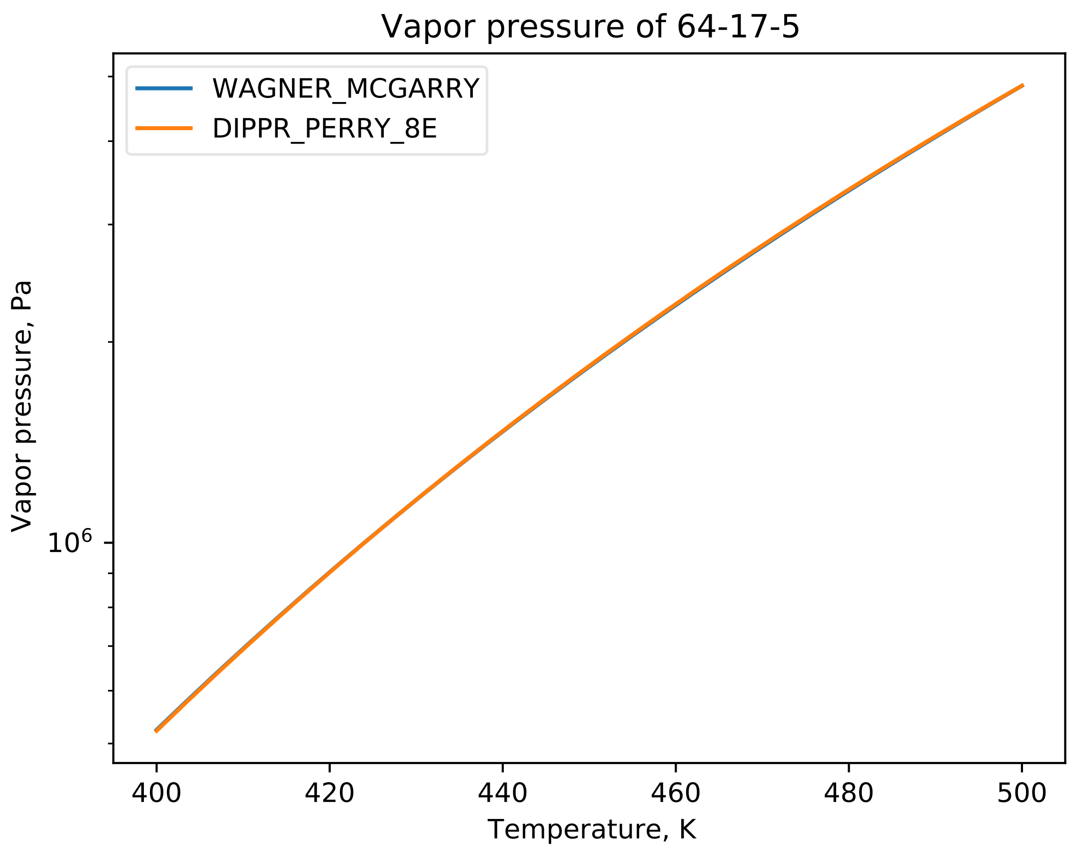
```
>>> ethanol_psat.plot_T_dependent_property(Tmin=400, Tmax=500, methods=['COOLPROP',
↪'WAGNER_MCGARRY', 'DIPPR_PERRY_8E'], pts=30)
```

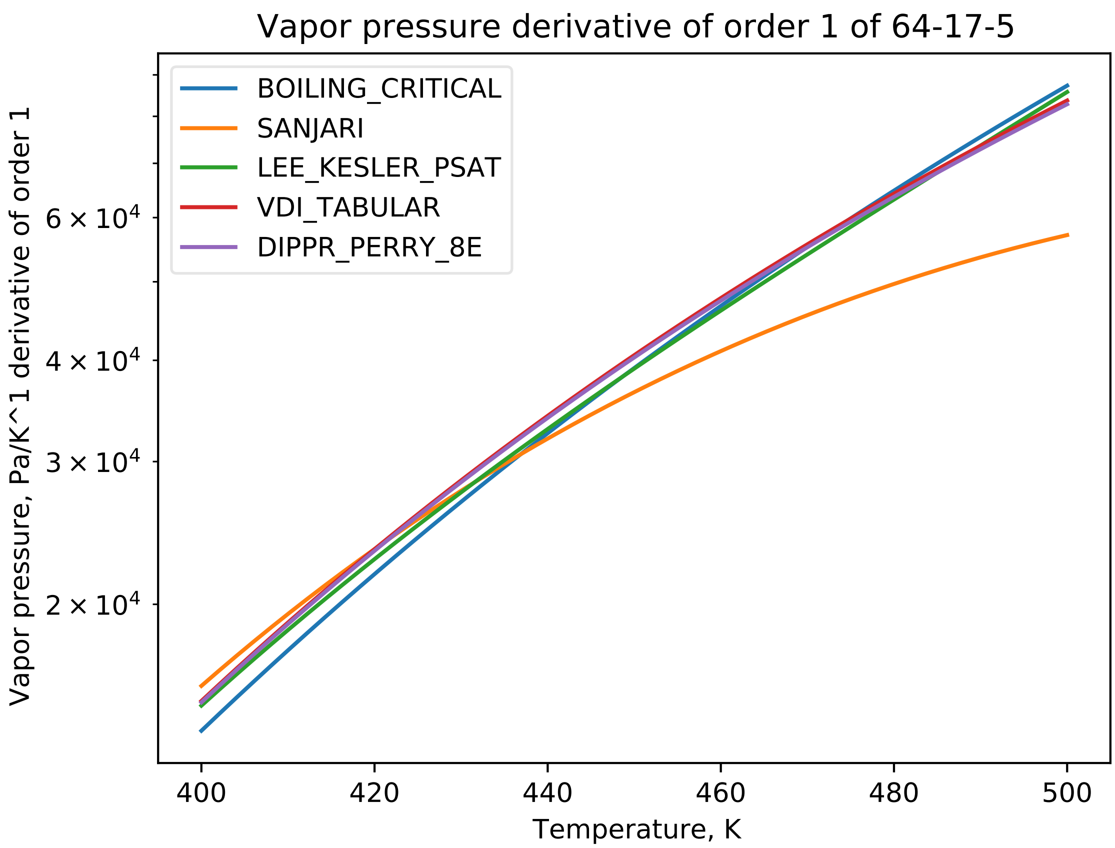
It is also possible to plot the nth derivative of the methods with the *order* parameter. The following plot shows the first derivative of vapor pressure of three estimation methods, a tabular source being interpolated, and 'DIPPR_PERRY_8E' as a reference method.

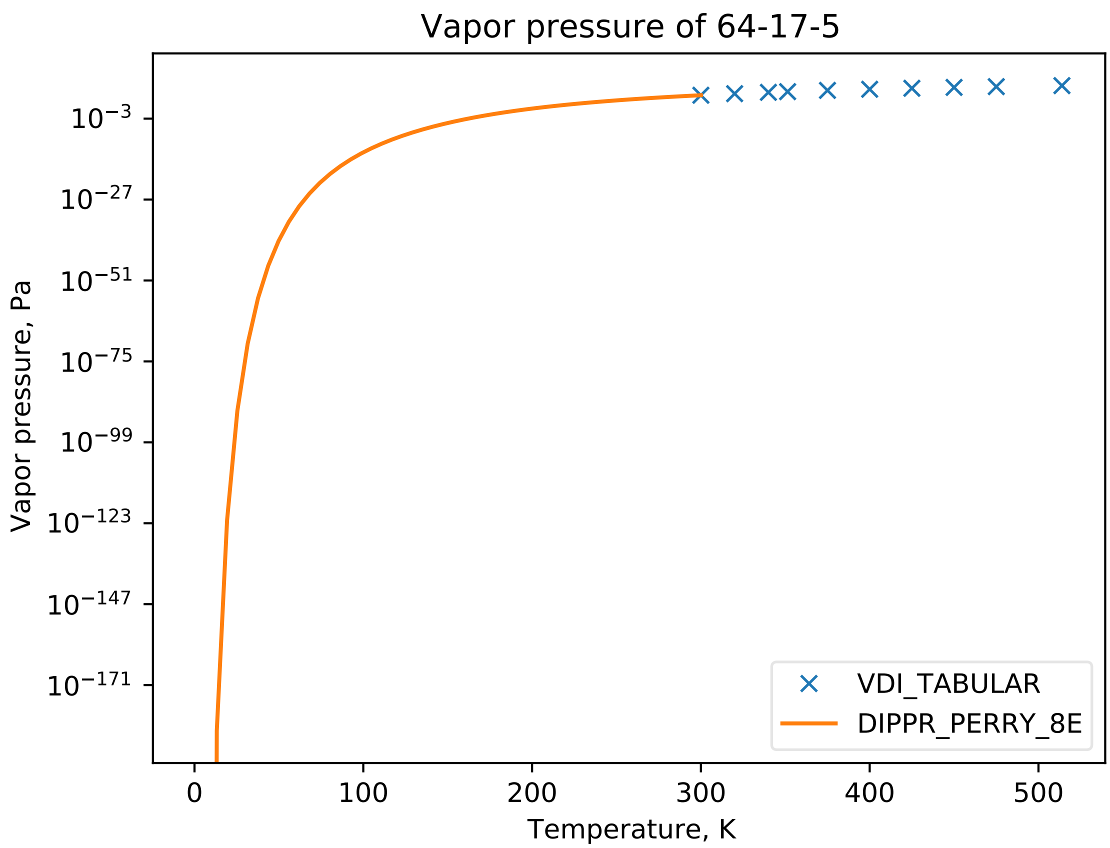
```
>>> ethanol_psat.plot_T_dependent_property(Tmin=400, Tmax=500, methods=['BOILING_CRITICAL
↪', 'SANJARI', 'LEE_KESLER_PSAT', 'VDI_TABULAR', 'DIPPR_PERRY_8E'], pts=50, order=1)
```

Plots show how the extrapolation methods work. By default plots do not show extrapolated values from methods, but this can be forced by setting *only_valid* to False. It is easy to see that extrapolation is designed to show the correct trend, but that individual methods will have very different extrapolations.

```
>>> ethanol_psat.plot_T_dependent_property(Tmin=1, Tmax=300, methods=['VDI_TABULAR',
↪'DIPPR_PERRY_8E', 'COOLPROP'], pts=50, only_valid=False)
```

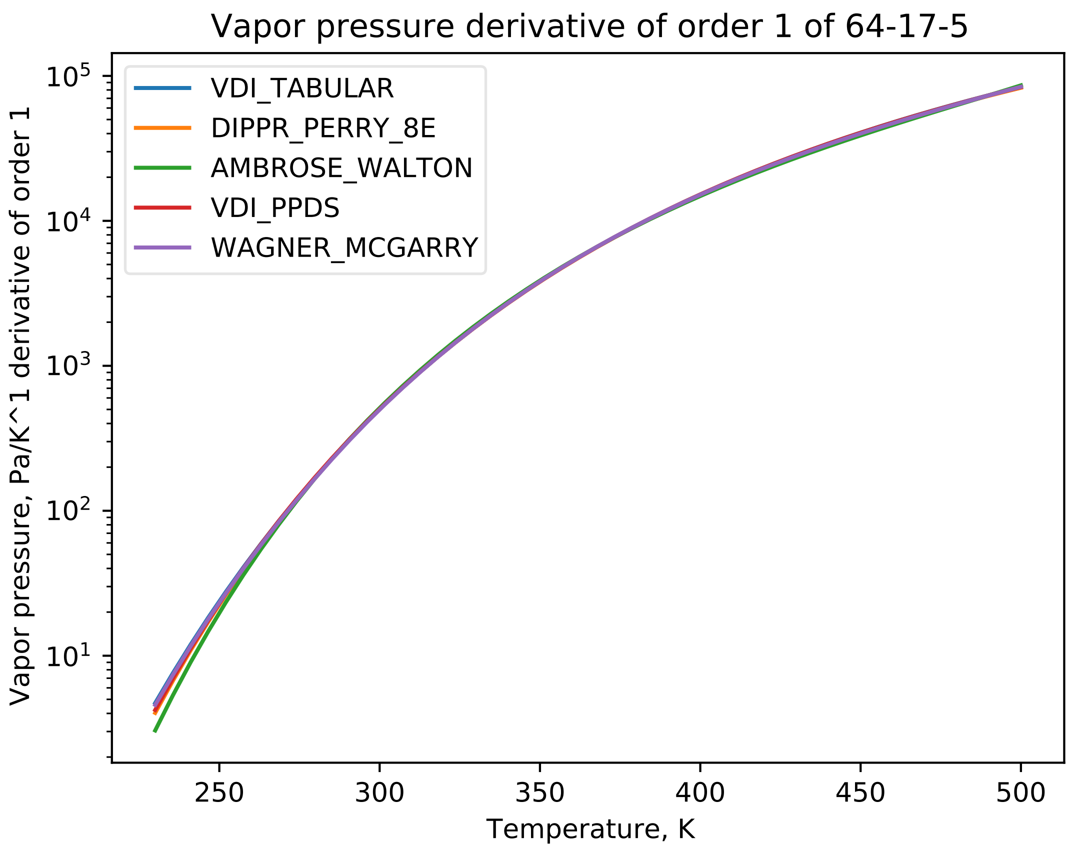

Vapor pressure of 64-17-5
WAGNER_MCGARRY
DIPPR_PERRY_8E
Vapor pressure, Pa
10^6
400
420
440
460
480
500
Temperature, K


Vapor pressure derivative of order 1 of 64-17-5
BOILING_CRITICAL
SANJARI
LEE_KESLER_PSAT
VDI_TABULAR
DIPPR_PERRY_8E
Vapor pressure, Pa/K^1 derivative of order 1
6 × 10^4
4 × 10^4
3 × 10^4
2 × 10^4
400
420
440
460
480
500
Temperature, K


Vapor pressure of 64-17-5
Vapor pressure, Pa
Temperature, K
VDI_TABULAR
DIPPR_PERRY_8E
10^−3
10^−27
10^−51
10^−75
10^−99
10^−123
10^−147
10^−171
0
100
200
300
400
500

It may also be helpful to see the derivative with respect to temperature of methods. This can be done with the *order* keyword:

```
>>> ethanol_psat.plot_T_dependent_property(Tmin=1, Tmax=300, methods=['VDI_TABULAR',
↪'DIPPR_PERRY_8E', 'COOLPROP'], pts=50, only_valid=False, order=1)
```



Higher order derivatives are also supported; most derivatives are numerically calculated, so there may be some noise. The derivative plot is particularly good at illustrating what happens at the critical point, when extrapolation takes over from the actual formulas.

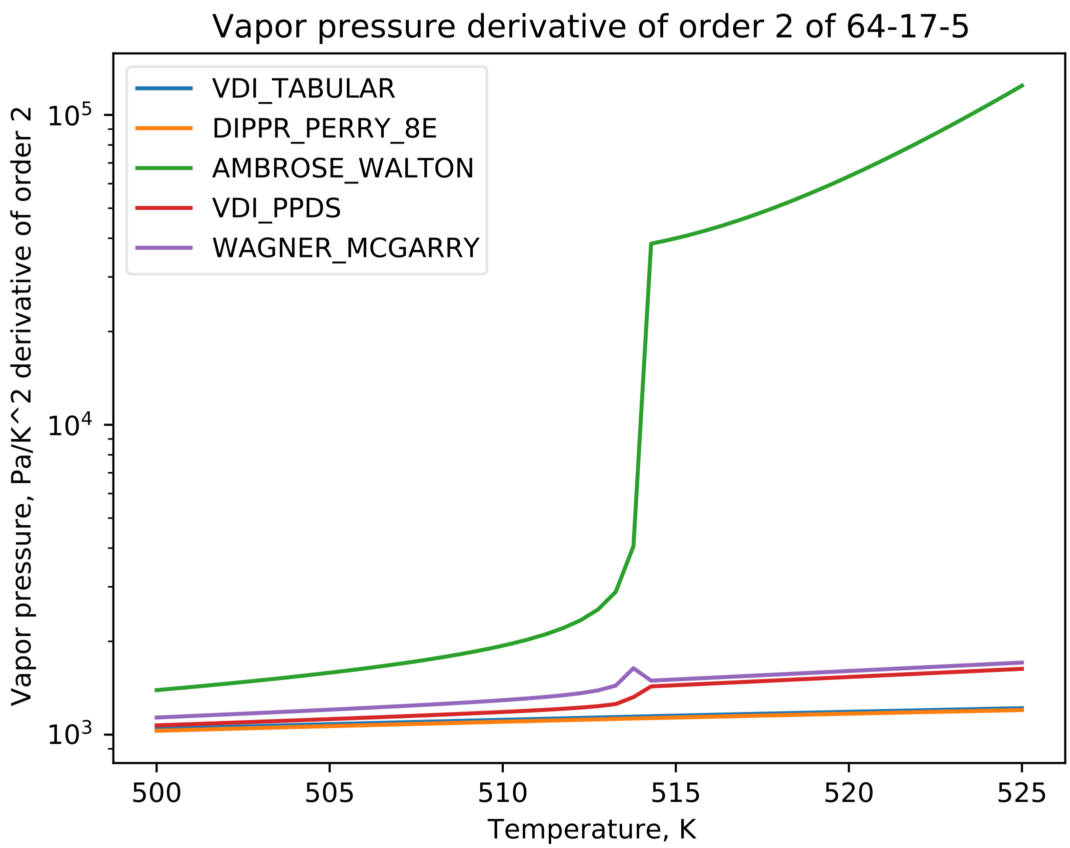
```
>>> ethanol_psat.plot_T_dependent_property(Tmin=500, Tmax=525, methods=['VDI_TABULAR',
↪'DIPPR_PERRY_8E', 'AMBROSE_WALTON', 'VDI_PPDS', 'WAGNER_MCGARRY'], pts=50, only_
↪valid=False, order=2)
```

3.1.6 Calculating Temperature From Properties

There is also functionality for reversing the calculation - finding out which temperature produces a specific property value. The method is *solve_property*. For vapor pressure, we can use this technique to find out the normal boiling point as follows:

```
>>> ethanol_psat.solve_property(101325)
351.43136
```

The experimentally reported value is 351.39 K.



Vapor pressure derivative of order 2 of 64-17-5
VDI_TABULAR
DIPPR_PERRY_8E
AMBROSE_WALTON
VDI_PPDS
WAGNER_MCGARRY
10^5
10^4
10^3
Vapor pressure, Pa/K^2 derivative of order 2
500
505
510
515
520
525
Temperature, K

3.1.7 Property Derivatives

Functionality for calculating the derivative of the property is also implemented as *T_dependent_property_derivative* :

```
>>> ethanol_psat.T_dependent_property_derivative(300)
498.882
```

The derivatives are numerical unless a special implementation has been added to the property's *calculate_derivative* method.

Higher order derivatives are available as well with the *order* argument. All higher-order derivatives are numerical, and they tend to have reduced numerical precision due to floating point limitations.

```
>>> ethanol_psat.T_dependent_property_derivative(300.0, order=2)
24.74
>>> ethanol_psat.T_dependent_property_derivative(300.0, order=3)
2.75
```

3.1.8 Property Integrals

Functionality for integrating over a property is implemented as *T_dependent_property_integral*.

$$\text{integral} = \int_{T_1}^{T_2} \text{property } dT$$

When the property is heat capacity, this calculation represents a change in enthalpy:

$$\Delta H = \int_{T_1}^{T_2} C_p \, dT$$

```
>>> CH4_Cp = HeatCapacityGas(CASRN='74-82-8')
>>> CH4_Cp.method = 'POLING_POLY'
>>> CH4_Cp.T_dependent_property_integral(300, 500)
8158.64
```

Besides enthalpy, a commonly used integral is that of the property divided by T:

$$\text{integral} = \int_{T_1}^{T_2} \frac{\text{property}}{T} dT$$

When the property is heat capacity, this calculation represents a change in entropy:

$$\Delta S = \int_{T_1}^{T_2} \frac{C_p}{T} dT$$

This integral, property over T, is implemented as *T_dependent_property_integral_over_T* :

```
>>> CH4_Cp.T_dependent_property_integral_over_T(300, 500)
20.6088
```

Where speed has been important so far, these integrals have been implemented analytically in a property object's *calculate_integral* and *calculate_integral_over_T* method; otherwise the integration is performed numerically.

3.1.9 Using Tabular Data

A common scenario is that there are no correlations available for a compound, and that estimation methods are not applicable. However, there may be a few experimental data points available in the literature. In this case, the data can be specified and used directly with the *add_tabular_data* method. Extrapolation can often show the correct trends for these properties from even a few data points.

In the example below, we take 5 data points on the vapor pressure of water from 300 K to 350 K, and use them to extrapolate and estimate the triple temperature and critical temperature (assuming we know the triple and critical pressures).

```
>>> from thermo import *
>>> import numpy as np
>>> w = VaporPressure(Tb=373.124, Tc=647.14, Pc=22048320.0, omega=0.344, CASRN='7732-18-5
↪', extrapolation='AntoineAB')
>>> Ts = np.linspace(300, 350, 5).tolist()
>>> Ps = [3533.9, 7125., 13514., 24287., 41619.]
>>> w.add_tabular_data(Ts=Ts, properties=Ps)
>>> w.solve_property(610.707), w.solve_property(22048320)
(272.83, 617.9)
```

The experimental values are 273.15 K and 647.14 K.

3.1.10 Adding New Methods

While a great many property methods have been implemented, there is always the case where a new one must be added. To support that, the method *add_method* will add a user-specified method and switch the method selected to the newly added method.

As an example, we can compare the default vapor pressure formulation for n-hexane against a set of Antoine coefficients on the NIST WebBook.

```
>>> from chemicals import *
>>> from thermo import *
>>> obj = VaporPressure(CASRN= '110-54-3')
>>> obj(200)
20.742
>>> f = lambda T: Antoine(T=T, A=3.45604+5, B=1044.038, C=-53.893)
>>> obj.add_method(f=f, name='WebBook', Tmin=177.70, Tmax=264.93)
>>> obj.method
'WebBook'
>>> obj.extrapolation = 'AntoineAB'
>>> obj(200.0)
20.432
```

We can, again, extrapolate quite easily and estimate the triple temperature and critical temperature from these correlations (if we know the triple pressure and critical pressure).

```
>>> obj.solve_property(1.378), obj.solve_property(3025000.0)
(179.43, 508.04)
```

Optionally, some derivatives and integrals can be provided for new methods as well. This avoids having to compute derivatives or integrals numerically. SymPy may be helpful to find these analytical derivatives or integrals in many cases, as in the following example:

```
>>> from sympy import symbols, lambdify, diff
>>> T = symbols('T')
>>> A, B, C = 3.45604+5, 1044.038, -53.893
>>> expr = 10**(A - B/(T + C))
>>> f = lambdify(T, expr)
>>> f_der = lambdify(T, diff(expr, T))
>>> f_der2 = lambdify(T, diff(expr, T, 2))
>>> f_der3 = lambdify(T, diff(expr, T, 3))
>>> obj.add_method(f=f, f_der=f_der, f_der2=f_der2, f_der3=f_der3, name='WebBookSymPy',␣
↪Tmin=177.70, Tmax=264.93)
>>> obj.method, obj(200), obj.T_dependent_property_derivative(200.0, order=2)
('WebBookSymPy', 20.43298036, 0.2276285)
```

Note that adding methods like this breaks the ability to export as json and the repr of the object is no longer complete.

3.1.11 Adding New Correlation Coefficient Methods

While adding entirely new methods is useful, it is more common to want to use different coefficients in an existing equation. A number of different equations are recognized, and accept/require the parameters as per their function name in e.g. `chemicals.vapor_pressure.Antoine`. More than one set of coefficients can be added for each model. After adding a new correlation the method is set to that method.

```
>>> obj = VaporPressure()
>>> obj.add_correlation(name='WebBook', model='Antoine', Tmin=177.70, Tmax=264.93,  A=3.
↪45604+5, B=1044.038, C=-53.893)
>>> obj(200)
20.43298036711
```

It is also possible to specify the parameters in the constructor of the object as well:

```
>>> obj = VaporPressure(Antoine_parameters={'WebBook': {'A': 8.45604, 'B': 1044.038, 'C':
↪ -53.893, 'Tmin': 177.7, 'Tmax': 264.93}})
>>> obj(200)
20.43298036711
```

More than one set of parameters and more than one model may be specified this way; the model name is the same, with '_parameters' appended to it.

For a full list of supported correlations (and their names), see *add_correlation*.

3.1.12 Fitting Correlation Coefficients

Thermo contains functionality for performing regression to obtain equation coefficients from experimental data.

Data is obtained from the DDBST for the vapor pressure of acetone (http://www.ddbst.com/en/EED/PCP/VAP_C4.php), and coefficients are regressed for several methods. There is data from five sources on that page, but no uncertainties are available; the fit will treat each data point equally.

```
>>> Ts = [203.65, 209.55, 212.45, 234.05, 237.04, 243.25, 249.35, 253.34, 257.25, 262.12,
↪ 264.5, 267.05, 268.95, 269.74, 272.95, 273.46, 275.97, 276.61, 277.23, 282.03, 283.06,
↪ 288.94, 291.49, 293.15, 293.15, 293.85, 294.25, 294.45, 294.6, 294.63, 294.85, 297.05,
↪ 297.45, 298.15, 298.15, 298.15, 298.15, 298.15, 299.86, 300.75, 301.35, 303.15, 303.
↪15, 304.35, 304.85, 305.45, 306.25, 308.15, 308.15, 308.15, 308.22, 308.35, 308.45,␣
↪308.85, 309.05, 311.65, 311.85, 311.85, 311.95, 312.25, 314.68, 314.85, 317.75, 317.85,
↪ 318.05, 318.15, 318.66, 320.35, 320.35, 320.45, 320.65, 322.55, 322.65, 322.85, 322.
↪95, 322.95, 323.35, 323.55, 324.65, 324.75, 324.85, 324.85, 325.15, 327.05, 327.15,␣
↪327.2, 327.25, 327.35, 328.22, 328.75, 328.85, 333.73, 338.95]
```

(continues on next page)

(continued from previous page)

```
>>> Psats = [58.93, 94.4, 118.52, 797.1, 996.5, 1581.2, 2365, 3480, 3893, 5182, 6041,␣
↪6853, 7442, 7935, 9290, 9639, 10983, 11283, 13014, 14775, 15559, 20364, 22883, 24478,␣
↪24598, 25131, 25665, 25931, 25998, 26079, 26264, 29064, 29598, 30397, 30544, 30611,␣
↪30784, 30851, 32636, 33931, 34864, 37637, 37824, 39330, 40130, 41063, 42396, 45996,␣
↪46090, 46356, 45462, 46263, 46396, 47129, 47396, 52996, 52929, 53262, 53062, 53796,␣
↪58169, 59328, 66395, 66461, 67461, 67661, 67424, 72927, 73127, 73061, 73927, 79127,␣
↪79527, 80393, 79927, 80127, 81993, 80175, 85393, 85660, 85993, 86260, 86660, 92726,␣
↪92992, 92992, 93126, 93326, 94366, 98325, 98592, 113737, 136626]
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='Antoine',
↪ do_statistics=True, multiple_tries=True, model_kwargs={'base': 10.0})
>>> res, stats['MAE']
({'A': 9.2515513342, 'B': 1230.099383065, 'C': -40.08076540233, 'base': 10.0}, 0.
↪01059288655304)
```

The fitting function returns the regressed coefficients, and optionally some statistics. The mean absolute relative error or "MAE" is often a good parameter for determining the goodness of fit; Antoine yielded an error of about 1%.

There are lots of methods available; Antoine was just used (the returned coefficients are in units of K and Pa with a base of 10), but for comparison several more are as well. Note that some require the critical temperature and/or pressure.

```
>>> Tc, Pc = 508.1, 4700000.0
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='Yaws_Psat
↪', do_statistics=True, multiple_tries=True)
>>> res, stats['MAE']
({'A': 1650.7, 'B': -32673., 'C': -728.7, 'D': 1.1, 'E': -0.000609}, 0.0178)
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='DIPPR101
↪', do_statistics=True, multiple_tries=3)
>>> stats['MAE']
0.0106
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='Wagner',␣
↪do_statistics=True, multiple_tries=True, model_kwargs={'Tc': Tc, 'Pc': Pc})
>>> res, stats['MAE']
({'Tc': 508.1, 'Pc': 4700000.0, 'a': -15.7110, 'b': 23.63, 'c': -27.74, 'd': 25.152}, 0.
↪0485)
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='TRC_
↪Antoine_extended', do_statistics=True, multiple_tries=True, model_kwargs={'Tc': Tc})
>>> res, stats['MAE']
({'Tc': 508.1, 'to': 67.0, 'A': 9.2515481, 'B': 1230.0976, 'C': -40.080954, 'n': 2.5, 'E
↪': 333.0, 'F': -24950.0}, 0.01059)
```

A very common scenario is that some coefficients are desired to be fixed in the regression. This is supported with the *model_kwargs* attribute. For example, in the above DIPPR101 case we can fix the *E* coefficient to 1 as follows:

```
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='DIPPR101
↪', do_statistics=True, multiple_tries=3, model_kwargs={'E': -1})
>>> res['E'], stats['MAE']
(-1, 0.01310)
```

Similarly, the feature is often used to set unneeded coefficients to zero In this case the TDE_PVExpansion function has up to 8 parameters but only three are justified.

```
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='TDE_
↪PVExpansion', do_statistics=True, multiple_tries=True, model_kwargs={'a4': 0.0, 'a5':␣
↪0.0, 'a6': 0.0, 'a7': 0.0, 'a8': 0})
```

(continues on next page)

(continued from previous page)

```
>>> res, stats['MAE']
({'a4': 0.0, 'a5': 0.0, 'a6': 0.0, 'a7': 0.0, 'a8': 0, 'a1': 48.396547, 'a2': -4914.1260,
↪ 'a3': -3.78894783}, 0.0131003)
```

Fitting coefficients is a complicated numerical problem. MINPACK's lmfit implements Levenberg-Marquardt with a number of tricks, and is used through SciPy in the fitting by default. Other minimization algorithms are supported, but generally don't do nearly as well. All minimization algorithms can only converge to a minima near points that they evaluate, and the choice of initial guesses is quite important. For many methods, there are several hardcoded guesses. By default, each of those guesses are evaluated and the minimization is initialized with the best guess. However, for maximum accuracy, *multiple_tries* should be set to True, and *all* initial guesses are converged, and the best fit is returned.

Initial guesses for parameters can also be provided. In the below example, the initial parameters from http://ddbonline.ddbst.com/AntoineCalculation/AntoineCalculationCGI.exe for acetone are provided as initial guesses (converting them to a Pa and K basis, from mmHg and deg C).

```
>>> from math import log10
>>> res, stats = TDependentProperty.fit_data_to_model(Ts=Ts, data=Psats, model='Antoine',
↪ do_statistics=True, multiple_tries=True, guesses={'A':  7.6313 +log10(101325/760), 'B
↪':  1566.69 , 'C': 273.419 -273.15}, model_kwargs={'base': 10.0})
```

In this case the initial guesses are good, but different parameters are still obtained by the fitting algorithm.

To speed up these calculations, an interface to numba is available. Simply set *use_numba* to True. Note that the first regression per session may be slower as it has to compile the function.

3.1.13 Adding New Correlation Coefficient Methods From Data

In the following example, data for the molar volume of three phases of liquid oxygen are added, from Roder, H. M. "The Molar Volume (Density) of Solid Oxygen in Equilibrium with Vapor." Journal of Physical and Chemical Reference Data 7, no. 3 (1978): 949–58.

Each of the phases is treated as a different method. After fitting the data to linear and quadratic fits, the results are plotted.

```
>>> Ts_alpha = [4.2, 10.0, 18.5, 20, 21, 22, 23.880]
>>> Vms_alpha = [20.75e-6, 20.75e-6, 20.75e-6, 20.75e-6, 20.75e-6, 20.78e-6, 20.82e-6]
>>> Ts_beta = [23.880, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 43.801]
>>> Vms_beta = [20.95e-6, 20.95e-6, 21.02e-6, 21.08e-6, 21.16e-6, 21.24e-6, 21.33e-6, 21.
↪42e-6, 21.52e-6, 21.63e-6, 21.75e-6, 21.87e-6]
>>> Ts_gamma = [42.801, 44.0, 46.0, 48.0, 50.0, 52.0, 54.0, 54.361]
>>> Vms_gamma = [23.05e-6, 23.06e-6, 23.18e-6, 23.30e-6, 23.43e-6, 23.55e-6, 23.67e-6,␣
↪23.69e-6]
```

```
>>> obj = VolumeSolid(CASRN='7782-44-7')
>>> obj.fit_add_model(Ts=Ts_alpha, data=Vms_alpha, model='linear', name='alpha')
>>> obj.fit_add_model(Ts=Ts_beta, data=Vms_beta, model='quadratic', name='beta')
>>> obj.fit_add_model(Ts=Ts_gamma, data=Vms_gamma, model='quadratic', name='gamma')
>>> obj.plot_T_dependent_property(Tmin=4.2, Tmax=50)
```

3.2 Temperature and Pressure Dependent Properties

The pressure dependent objects work much like the temperature dependent ones; in fact, they subclass *TDependentProperty*. They have many new methods that require pressure as an input however. They work in two parts: a low-pressure correlation component, and a high-pressure correlation component. The high-pressure component usually but not always requires a low-pressure calculation to be performed first as its input.

3.2.1 Creating Objects

All arguments and information the property object requires must be provided in the constructor of the object. If a piece of information is not provided, whichever methods require it will not be available for that object. Many pressure-dependent property correlations are actually dependent on other properties being calculated first. A mapping of those dependencies is as follows:

- Liquid molar volume: Depends on *VaporPressure*
- Gas viscosity: Depends on *VolumeGas*
- Liquid viscosity: Depends on *VaporPressure*
- Gas thermal conductivity: Depends on *VolumeGas*, *HeatCapacityGas*, *ViscosityGas*

The required input objects should be created first, and provided as an input to the dependent object:

```
>>> water_psat = VaporPressure(Tb=373.124, Tc=647.14, Pc=22048320.0, omega=0.344, CASRN=
↪'7732-18-5')
>>> water_mu = ViscosityLiquid(CASRN="7732-18-5", MW=18.01528, Tm=273.15, Tc=647.14,␣
↪Pc=22048320.0, Vc=5.6e-05, omega=0.344, method="DIPPR_PERRY_8E", Psat=water_psat,␣
↪method_P="LUCAS")
```

Various data files will be searched to see if information such as DIPPR expression coefficients are available for the compound during the initialization. This behavior can be avoided by setting the optional *load_data* argument to False.

3.2.2 Pressure-dependent Methods

The pressure and temperature dependent object selects a low-pressure and a high-pressure method automatically during initialization. These method can be inspected with the *method* and *method_P* properties. If no low-pressure methods are available, *method* will be None. If no high-pressure methods are available, *method_P* will be None. *method* and *method_P* are also valid parameters when constructing the object, but if either of the methods specified is not available an exception will be raised.

```
>>> water_mu.method, water_mu.method_P
('DIPPR_PERRY_8E', 'LUCAS')
```

All available low-pressure methods can be found by inspecting the `all_methods` attribute:

```
>>> water_mu.all_methods
{'COOLPROP', 'DIPPR_PERRY_8E', 'VISWANATH_NATARAJAN_3', 'VDI_PPDS', 'LETSOU_STIEL'}
```

All available high-pressure methods can be found by inspecting the `all_methods_P` attribute:

```
>>> water_mu.all_methods_P
{'COOLPROP', 'LUCAS'}
```

Changing the low-pressure method or the high-pressure method is as easy as setting a new value to the attribute:

```
>>> water_mu.method = 'VDI_PPDS'
>>> water_mu.method
'VDI_PPDS'
>>> water_mu.method_P = 'COOLPROP'
>>> water_mu.method_P
'COOLPROP'
```

3.2.3 Calculating Properties

Calculation of the property at a specific temperature and pressure is as easy as calling the object which triggers the *__call__* method:

```
>>> water_mu.method = 'VDI_PPDS'
>>> water_mu.method_P = 'COOLPROP'
>>> water_mu(T=300.0, P=1e5)
0.000853742
```

This is actually a cached wrapper around the specific call, *TP_dependent_property*:

```
>>> water_mu.TP_dependent_property(300.0, P=1e5)
0.000853742
```

The caching of *__call__* is quite basic - the previously specified temperature and pressure are stored, and if the new T and P are the same as the previous T and P the previously calculated result is returned.

There is a lower-level interface for calculating properties with a specified method by name, `calculate_P`. *TP_dependent_property* is a wrapper around `calculate_P` that includes validation of the result.

```
>>> water_mu.calculate_P(T=300.0, P=1e5, method='COOLPROP')
0.000853742
>>> water_mu.calculate_P(T=300.0, P=1e5, method='LUCAS')
0.000865292
```

The above examples all show using calculating the property with a pressure specified. The same *TDependentProperty* methods are available too, so all the low-pressure calculation calls are also available.

```
>>> water_mu.calculate(T=300.0, method='VISWANATH_NATARAJAN_3')
0.000856467
>>> water_mu.T_dependent_property(T=400.0)
0.000217346
```

3.2.4 Limits and Extrapolation

The same temperature limits and low-pressure extrapolation methods are available as for *TDependentProperty*.

```
>>> water_mu.valid_methods(T=480)
['DIPPR_PERRY_8E', 'COOLPROP', 'VDI_PPDS', 'LETSOU_STIEL']
>>> water_mu.extrapolation
'linear'
```

To better understand what methods are available, the `valid_methods_P` method checks all available high-pressure correlations against their temperature and pressure limits.

```
>>> water_mu.valid_methods_P(T=300, P=1e9)
['LUCAS', 'COOLPROP']
>>> water_mu.valid_methods_P(T=300, P=1e10)
['LUCAS']
>>> water_mu.valid_methods_P(T=900, P=1e6)
['LUCAS']
```

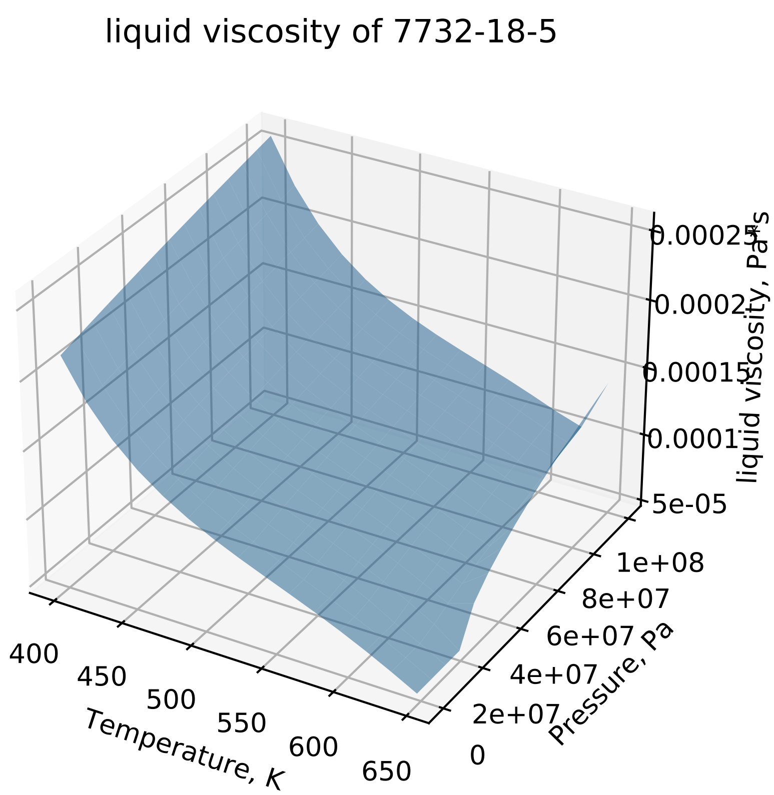
If the temperature and pressure are not provided, all available methods are returned; the returned value favors the methods by the ranking defined in thermo, with the currently selected method as the first item.

```
>>> water_mu.valid_methods_P()
['LUCAS', 'COOLPROP']
```

3.2.5 Plotting

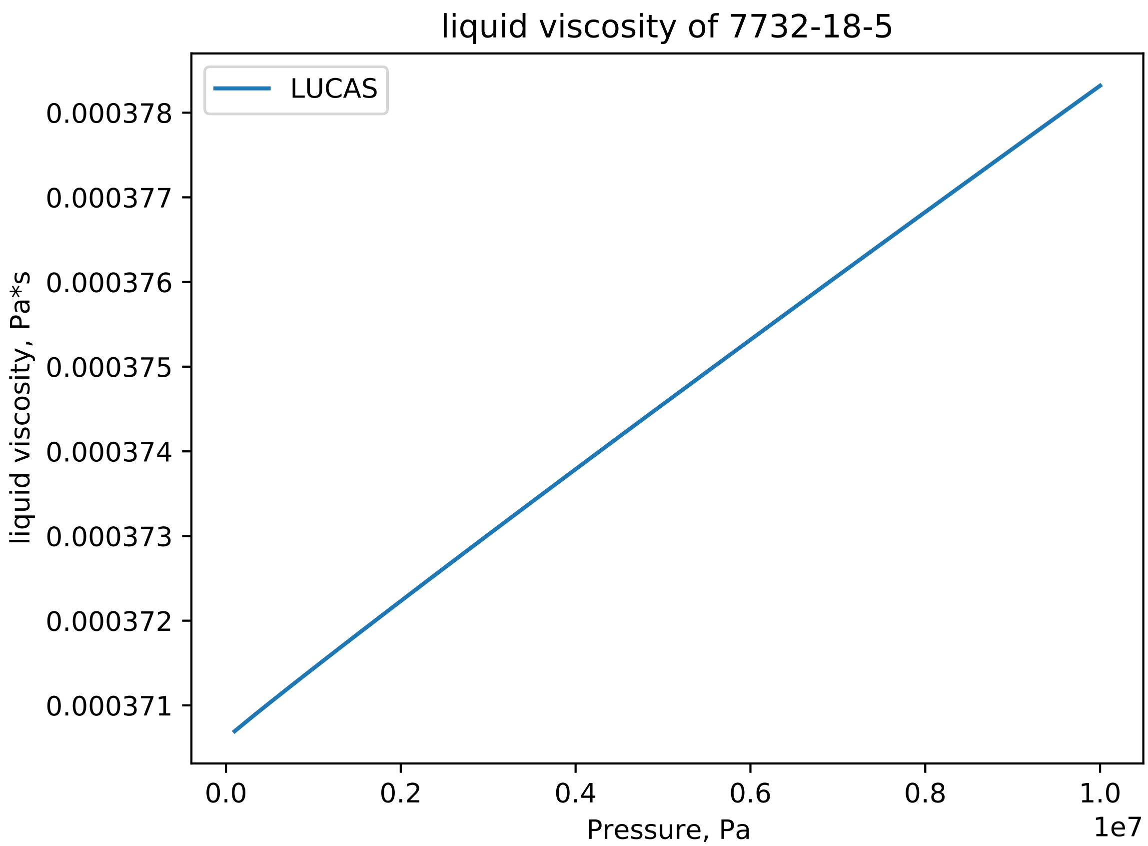
It is possible to compare the correlations graphically with the method *plot_TP_dependent_property*.

```
>>> water_mu.plot_TP_dependent_property(Tmin=400, Pmin=1e5, Pmax=1e8, methods_P=[
↪'COOLPROP','LUCAS'], pts=15, only_valid=False)
```

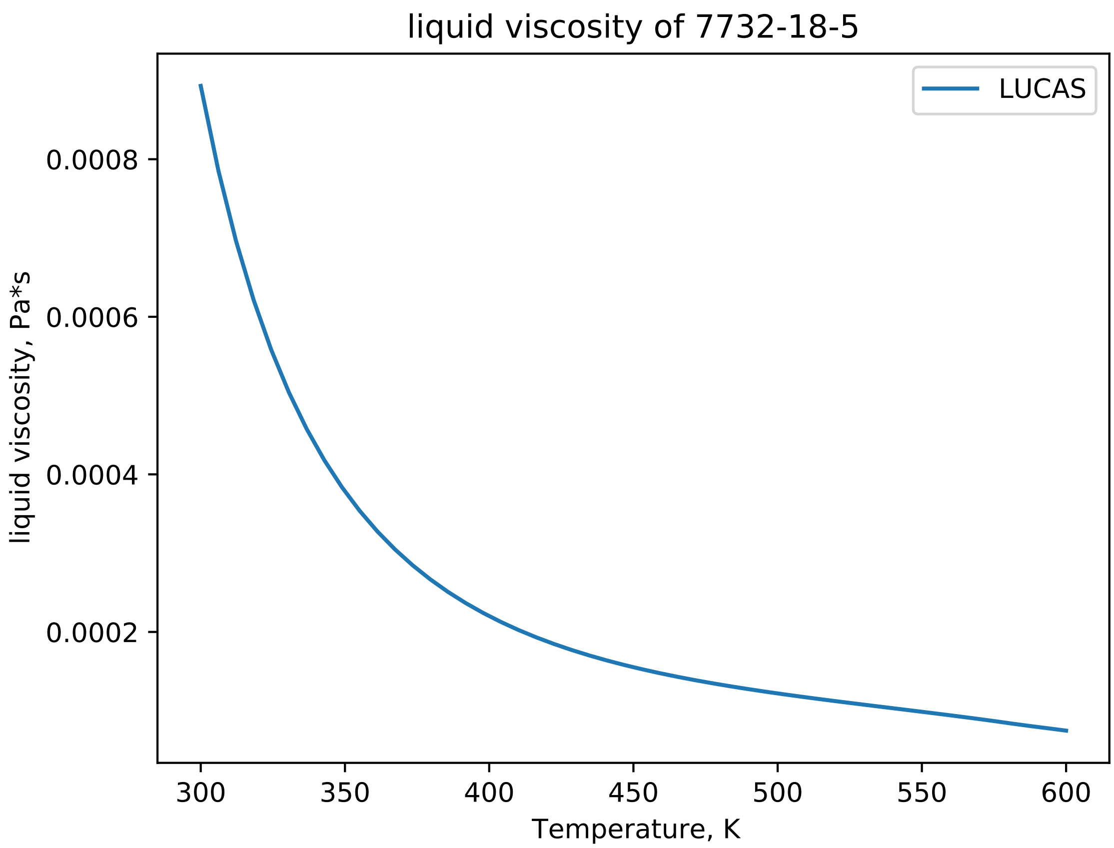


This can be a little confusing; but isotherms and isobars can be plotted as well, which are more straight forward. The respective methods are *plot_isotherm* and *plot_isobar*:

```
>>> water_mu.plot_isotherm(T=350, Pmin=1e5, Pmax=1e7, pts=50)
```


liquid viscosity of 7732-18-5
LUCAS
0.000378
0.000377
0.000376
0.000375
0.000374
0.000373
0.000372
0.000371
liquid viscosity, Pa*s
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Pressure, Pa

```
>>> water_mu.plot_isobar(P=1e7, Tmin=300, Tmax=600, pts=50)
```



3.2.6 Calculating Conditions From Properties

The method is *solve_property* works only on the low-pressure correlations.

```
>>> water_mu.solve_property(1e-3)
294.0711641
```

3.2.7 Property Derivatives

Functionality for calculating the temperature derivative of the property is implemented twice; as *T_dependent_property_derivative* using the low-pressure correlations, and as *TP_dependent_property_derivative_T* using the high-pressure correlations that require pressure as an input.

```
>>> water_mu.T_dependent_property_derivative(300)
-1.893961e-05
>>> water_mu.TP_dependent_property_derivative_T(300, P=1e7)
-1.927268e-05
```

The derivatives are numerical unless a special implementation has been added to the property's *calculate_derivative_T* and/or *calculate_derivative* method.

Higher order derivatives are available as well with the *order* argument.

```
>>> water_mu.T_dependent_property_derivative(300.0, order=2)
5.923372e-07
>>> water_mu.TP_dependent_property_derivative_T(300.0, P=1e6, order=2)
-1.40946e-06
```

Functionality for calculating the pressure derivative of the property is also implemented as `TP_dependent_property_derivative_P`:

```
>>> water_mu.TP_dependent_property_derivative_P(P=5e7, T=400)
4.27782809e-13
```

The derivatives are numerical unless a special implementation has been added to the property's `calculate_derivative_P` method.

Higher order derivatives are available as well with the *order* argument.

```
>>> water_mu.TP_dependent_property_derivative_P(P=5e7, T=400, order=2)
-1.1858461e-15
```

3.2.8 Property Integrals

The same functionality for integrating over a property as in temperature-dependent objects is available, but only for integrating over temperature using low pressure correlations. No other use cases have been identified requiring integration over high-pressure conditions, or integration over the pressure domain.

```
>>> water_mu.T_dependent_property_integral(300, 400) # Integrating over viscosity has no␣
↪physical meaning
0.04243
```

3.2.9 Using Tabular Data

If there are experimentally available data for a property at high and low pressure, an interpolation table can be created and used as follows. The CoolProp method is used to generate a small table, and is then added as a new method in the example below.

```
>>> from thermo import *
>>> import numpy as np
>>> Ts = [300, 400, 500]
>>> Ps = [1e5, 1e6, 1e7]
>>> table = [[water_mu.calculate_P(T, P, "COOLPROP") for T in Ts] for P in Ps]
>>> water_mu.method_P
'LUCAS'
>>> water_mu.add_tabular_data_P(Ts, Ps, table)
>>> water_mu.method_P
'Tabular data series #0'
>>> water_mu(400, 1e7), water_mu.calculate_P(400, 1e7, "COOLPROP")
(0.000221166933349, 0.000221166933349)
>>> water_mu(450, 5e6), water_mu.calculate_P(450, 5e6, "COOLPROP")
(0.00011340, 0.00015423)
```

The more data points used, the closer a property will match.

3.3 Mixture Properties

3.4 Notes

There is also the challenge that there is no clear criteria for distinguishing liquids from gases in supercritical mixtures. If the same method is not used for liquids and gases, there will be a sudden discontinuity which can cause numerical issues in modeling.

CHAPTER

FOUR

INTRODUCTION TO CHEMICALCONSTANTSPACKAGE AND PROPERTYCORRELATIONSPACKAGE

These two objects are designed to contain information needed by flash algorithms. In the first iteration of thermo, data was automatically looked up in databases and there was no way to replace that data. Thermo now keeps data and algorithms completely separate. This has also been very helpful to make unit tests that do not change their results.

There are five places to configure the flash and phase infrastructure:

- Constant data about chemicals, like melting point or boiling point or UNIFAC groups. This information needs to be put into an immutable `ChemicalConstantsPackage` object.
- Temperature-dependent data, like Antoine coefficients, Tait pressure-dependent volume parameters, or Laliberte electrolyte viscosity interaction parameters. These are stored in `TDependentProperty`, `TPDependentProperty`, and `MixtureProperty` objects. More information about configuring those to provide the desired properties can be found in property objects tutorial; this tutorial assumes you have already configured them as desired. These many objects are added to an `PropertyCorrelationsPackage` object before being provided to the flash algorithms.
- Phase-specific parameters that are not general and depend on a specific phase configuration for meaning; such as a volume translation coefficient or a binary interaction parameter. This information is provided when configuring each `Phase`.
- Information about bulk mixing rules or bulk property calculation methods; these don't have true thermodynamic definitions, and are configurable in the `BulkSettings` object.
- Settings of the `Flash` object; ideally no configuration would be required there. In some cases it might be useful to lower the tolerances or change an algorithm.

This tutorial covers the first two places, `ChemicalConstantsPackage` and `PropertyCorrelationsPackage`.

4.1 ChemicalConstantsPackage Object

4.1.1 Creating ChemicalConstantsPackage Objects

A *ChemicalConstantsPackage* can be created by specifying the known constant values of each chemical. All values are technically optional; the requirements of each *Flash* algorithm are different, but a minimum suggested amount is *names*, *CASs*, *MWs*, *Tcs*, *Pcs*, *omegas*, *Tbs*, and *atomss*. The list of all accepted properties can be found *here*.

```
>>> from thermo import ChemicalConstantsPackage, PropertyCorrelationsPackage
>>> constants = ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165],␣
↪names=['water', 'o-xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.324, 0.
↪331], Pcs=[22048320.0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14, 630.3, 616.2,␣
↪617.0])
```

4.1.2 Using ChemicalConstantsPackage Objects

Once created, all properties, even missing ones, can be accessed as attributes using the same names as required by the constructor:

```
>>> constants.MWs
[18.01528, 106.165, 106.165, 106.165]
>>> constants.Vml_STPs
[None, None, None, None]
```

It is the intention for these *ChemicalConstantsPackage* to be immutable. Python doesn't easily allow this to be enforced, but unexpected behavior will probably result if they are edited. If different properties are desired; create new *ChemicalConstantsPackage* objects.

The `__repr__` of the *ChemicalConstantsPackage* object returns a representation of the object that can be used to reconstruct it:

```
>>> constants
ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165], names=['water', 'o-
↪xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.324, 0.331], Pcs=[22048320.
↪0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14, 630.3, 616.2, 617.0])
>>> hash(eval(constants.__repr__())) == hash(constants)
True
```

4.1.3 Creating Smaller ChemicalConstantsPackage Objects

It is possible to create a new, smaller *ChemicalConstantsPackage* with fewer components by using the *subset* method, which accepts either indexes or slices and returns a new object:

```
>>> constants.subset([0, 1])
ChemicalConstantsPackage(MWs=[18.01528, 106.165], names=['water', 'o-xylene'], omegas=[0.
↪344, 0.3118], Pcs=[22048320.0, 3732000.0], Tcs=[647.14, 630.3])
>>> constants.subset(slice(1,3))
ChemicalConstantsPackage(MWs=[106.165, 106.165], names=['o-xylene', 'p-xylene'],␣
↪omegas=[0.3118, 0.324], Pcs=[3732000.0, 3511000.0], Tcs=[630.3, 616.2])
>>> constants.subset([0])
ChemicalConstantsPackage(MWs=[18.01528], names=['water'], omegas=[0.344], Pcs=[22048320.
↪0], Tcs=[647.14])
```

(continues on next page)

(continued from previous page)

It is also possible to reduce the number of properties set with the *subset* methods:

```
>>> constants.subset([1, 3], properties=('names', 'MWs'))
ChemicalConstantsPackage(MWs=[106.165, 106.165], names=['o-xylene', 'm-xylene'])
```

4.1.4 Adding or Replacing Constants

It is possible to create a new `ChemicalConstantsPackage` with added properties and/or replacing the old properties, from an existing object. This is helpful if better values for select properties are known. The `with_new_constants` method does this.

```
>>> constants.with_new_constants(Tcs=[650.0, 630.0, 620.0, 620.0], Tms=[20.0, 100.0, 50.
↪0, 12.3])
ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165], names=['water', 'o-
↪xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.324, 0.331], Pcs=[22048320.
↪0, 3732000.0, 3511000.0, 3541000.0], Tcs=[650.0, 630.0, 620.0, 620.0], Tms=[20.0, 100.
↪0, 50.0, 12.3])
```

4.1.5 Creating ChemicalConstantsPackage Objects from chemicals

A convenience method exists to load these constants from a different data files exists. Some values for all properties are available; not all compounds have all properties.

```
>>> obj = ChemicalConstantsPackage.constants_from_IDs(['methanol', 'ethanol',
↪'isopropanol'])
>>> obj.Tbs
[337.65, 351.39, 355.36]
```

When working with a fixed set of components, it may be a good idea to take this generated package, select only those properties being used, convert it to a string, and then embed that new object in a program. This will remove the need to load various data files, and if *chemicals* updates data files, different results won't be obtained from your constants package.

```
>>> small_obj = obj.subset(properties=('names', 'CASs', 'MWs', 'Tcs', 'Pcs', 'omegas',
↪'Tbs', 'Tms', 'atomss'))
>>> small_obj
ChemicalConstantsPackage(atomss=[{'C': 1, 'H': 4, 'O': 1}, {'C': 2, 'H': 6, 'O': 1}, {'C
↪': 3, 'H': 8, 'O': 1}], CASs=['67-56-1', '64-17-5', '67-63-0'], MWs=[32.04186, 46.
↪06844, 60.09502], names=['methanol', 'ethanol', 'isopropanol'], omegas=[0.559, 0.635,␣
↪0.665], Pcs=[8084000.0, 6137000.0, 4764000.0], Tbs=[337.65, 351.39, 355.36], Tcs=[512.
↪5, 514.0, 508.3], Tms=[175.15, 159.05, 183.65])
```

Once the object is printed, the generated text can be copy/pasted as valid Python into a program:

```
>>> obj = ChemicalConstantsPackage(atomss=[{'C': 1, 'H': 4, 'O': 1}, {'C': 2, 'H': 6, 'O
↪': 1}, {'C': 3, 'H': 8, 'O': 1}], CASs=['67-56-1', '64-17-5', '67-63-0'], MWs=[32.
↪04186, 46.06844, 60.09502], names=['methanol', 'ethanol', 'isopropanol'], omegas=[0.
↪5589999999999999, 0.635, 0.665], Pcs=[8084000.0, 6137000.0, 4764000.0], Tbs=[337.65,␣
↪351.39, 355.36], Tcs=[512.5, 514.0, 508.3], Tms=[175.15, 159.05, 183.65])
```

Warning: chemicals is a project with a focus on collecting data and correlations from various sources. In no way is it a project to critically evaluate these and provide recommendations. You are strongly encouraged to check values from it and modify them if you want different values. If you believe there is a value which has a typographical error please report it to the chemicals project. If data is missing or not as accuracte as you would like, and you know of a better method or source, new methods and sources can be added to chemicals fairly easily once the data entry is complete. It is not feasible to add individual components, so please submit a complete table of data from the source.

4.1.6 Storing and Loading ChemicalConstantsPackage Objects

For larger applications with many components, it is not as feasible to convert the *ChemicalConstantsPackage* to a string and embed it in a program. For that application, the object can be converted back and forth from JSON:

```
>>> obj = ChemicalConstantsPackage(MWs=[106.165, 106.165], names=['o-xylene', 'm-xylene
↪'])
>>> constants = ChemicalConstantsPackage(MWs=[18.01528, 106.165], names=['water', 'm-
↪xylene'])
>>> string = constants.as_json()
>>> new_constants = ChemicalConstantsPackage.from_json(string)
>>> hash(new_constants) == hash(constants)
True
```

4.2 PropertyCorrelationsPackage

CHAPTER

FIVE

INTRODUCTION TO PHASE AND FLASH CALCULATIONS

The framework for performing phase and flash calculations is designed around the following principles:

- Immutability
- Calculations are completely independent from any databases or lookups - every input must be provided as input
- Default to general-purpose algorithms that make no assumptions about specific systems
- Inclusion of separate flashes algorithms wherever faster algorithms can be used for specific cases
- Allow options to restart a flash from a nearby previously calculated result, with options to skip checking the result for stability
- Use very tight tolerances on all calculations
- Expose all constants used by algorithms

5.1 Phase Objects

A phase is designed to have a single state at any time, and contain all the information needed to compute phase-specific properties. Phases should always be initialized at a specific molar composition zs, T and P; and new phase objects at different conditions should be created from the existing ones with the `Phase.to` method (a little faster than creating them from scratch). That method also allows the new state to be set from any two of T, P, or V. When working in the T and P domain only, the `Phase.to_TP_zs` method is a little faster.

Phases are designed to be able to calculate every thermodynamic property. T and P are always attributes of the phase, but all other properties are functions that need to be called. Some examples of these properties are `V`, `H`, `S`, `Cp`, `dP_dT`, `d2P_dV2`, `fugacities`, `lnphis`, `dlnphis_dT`, and `dlnphis_dP`.

If a system is already known to be single-phase, the phase framework can be used directly without performing flash calculations. This may offer a speed boost in some applications.

5.1.1 Available Phases

Although the underlying equations of state often don't distinguish between liquid or vapor phase, it was convenient to create separate phase objects designed to hold gas, liquid, and solid phases separately.

The following phases can represent both a liquid and a vapor state. Their class is not a true indication that their properties are liquid or gas.

- Cubic equations of state - *CEOSLiquid* and *CEOSGas*
- IAPWS-95 Water and Steam - *IAPWS95Liquid* and *IAPWS95Gas*
- Wrapper objects for CoolProp's Helmholtz EOSs - *CoolPropLiquid* and *CoolPropGas*

The following phase objects can only represent a gas phase:

- Ideal-gas law - *IdealGas*
- High-accuracy properties of dry air - *DryAirLemmon*

The following phase objects can only represent a liquid phase:

- Ideal-liquid and/or activity coefficient models - *GibbsExcessLiquid*

5.1.2 Serialization

All phase models offer a *as_json* method and a *from_json* to serialize the object state for transport over a network, storing to disk, and passing data between processes.

```
>>> import json
>>> from scipy.constants import R
>>> from thermo import HeatCapacityGas, IdealGas, Phase
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13, R*1.57e-09,
↪ R*7e-08, R*-0.000261, R*3.539])), HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12,
↪ R*-6e-09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21], HeatCapacityGases=HeatCapacityGases)
>>> json_stuff = json.dumps(phase.as_json())
>>> new_phase = Phase.from_json(json.loads(json_stuff))
>>> assert new_phase == phase
```

Other json libraries can be used besides the standard json library by design.

Storing and recreating objects with Python's `pickle.dumps` library is also tested; this can be faster than using JSON at the cost of being binary data.

5.1.3 Hashing

All models have a *__hash__* method that can be used to compare different phases to see if they are absolutely identical (including which values have been calculated already).

They also have a *model_hash* method that can be used to compare different phases to see if they have identical model parameters.

They also have a *state_hash* method that can be used to compare different phases to see if they have identical temperature, composition, and model parameters.

5.2 Flashes with Pure Compounds

Pure components are really nice to work with because they have nice boundaries between each state, and the mole fraction is always 1; there is no composition dependence. There is a separate flash interfaces for pure components. These flashes are very mature and should be quite reliable.

5.2.1 Vapor-Liquid Cubic Equation Of State Example

The following example illustrates some of the types of flashes supported using the component methanol, the stated critical properties, a heat capacity correlation from Poling et. al., and the Peng-Robinson equation of state.

Obtain a heat capacity object, and select a source:

```
>>> from thermo.heat_capacity import POLING_POLY
>>> CpObj = HeatCapacityGas(CASRN='67-56-1')
>>> CpObj.method = POLING_POLY
>>> CpObj.POLING_coefs # Show the coefficients
[4.714, -0.006986, 4.211e-05, -4.443e-08, 1.535e-11]
>>> HeatCapacityGases = [CpObj]
```

Create a *ChemicalConstantsPackage* object which holds constant properties of the object, using a minimum of values:

```
>>> from thermo import ChemicalConstantsPackage, PropertyCorrelationsPackage, PRMIX,␣
↪SRKMIX, CEOSLiquid, CEOSGas, FlashPureVLS
>>> constants = ChemicalConstantsPackage(Tcs=[512.5], Pcs=[8084000.0], omegas=[0.559],␣
↪MWs=[32.04186], CASs=['67-56-1'])
```

Create a *PropertyCorrelationsPackage* object which holds temperature-dependent property objects, also setting *skip_missing* to True so no database lookups are performed:

```
>>> correlations = PropertyCorrelationsPackage(constants,␣
↪HeatCapacityGases=HeatCapacityGases, skip_missing=True)
```

Create liquid and gas cubic phase objects using the *Peng-Robinson equation of state*:

```
>>> eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
>>> liquid = CEOSLiquid(PRMIX, HeatCapacityGases=HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
>>> gas = CEOSGas(PRMIX, HeatCapacityGases=HeatCapacityGases, eos_kwargs=eos_kwargs)
```

Create the Flash object *FlashPureVLS* for pure components:

```
>>> flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])
```

Do a T-P flash:

```
>>> res = flasher.flash(T=300, P=1e5)
>>> res.phase, res.liquid0
('L', CEOSLiquid(eos_class=PRMIX, eos_kwargs={"Tcs": [512.5], "Pcs": [8084000.0], "omegas
↪": [0.559]}, HeatCapacityGases=[HeatCapacityGas(CASRN="67-56-1", extrapolation="linear
↪", method="POLING_POLY")], T=300.0, P=100000.0, zs=[1.0]))
```

Do a temperature and vapor-fraction flash:

```
>>> res = flasher.flash(T=300, VF=.3)
```

Do a pressure and vapor-fraction flash:

```
>>> res = flasher.flash(P=1e5, VF=.5)
```

Do a pressure and enthalpy flash:

```
>>> res = flasher.flash(P=1e5, H=100)
```

Do a pressure and entropy flash:

```
>>> res = flasher.flash(P=1e5, S=30)
```

Do a temperature and entropy flash:

```
>>> res = flasher.flash(T=400.0, S=30)
```

Do a temperature and enthalpy flash:

```
>>> res = flasher.flash(T=400.0, H=1000)
```

Do a volume and internal energy flash:

```
>>> res = flasher.flash(V=1e-4, U=1000)
```

As you can see, the interface is convenient and supports most types of flashes. In fact, the algorithms are generic; any of H, S, U, and can be combined with any combination of T, P, and V. Although most of the flashes shown above except TS and TH are usually well behaved, depending on the EOS combination there may be multiple solutions. No real guarantees can be made about which solution will be returned in those cases.

Flashes with two of H, S, and U are not implemented at present.

It is not necessary to use the same phase model for liquid and gas phases; the below example shows a flash switching the gas phase model to SRK.

```
>>> SRK_gas = CEOSGas(SRKMIX, HeatCapacityGases=HeatCapacityGases, eos_kwargs=eos_kwargs)
>>> flasher_inconsistent = FlashPureVLS(constants, correlations, gas=SRK_gas,␣
↪liquids=[liquid], solids=[])
>>> res = flasher_inconsistent.flash(T=400.0, VF=1)
```

Choosing to use an inconsistent model will slow down many calculations as more checks are required; and some flashes may have issues with discontinuities in some conditions, and simply a lack of solution in other conditions.

5.2.2 Vapor-Liquid Steam Example

The IAPWS-95 standard is implemented and available for easy use:

```
>>> from thermo import FlashPureVLS, IAPWS95Liquid, IAPWS95Gas, iapws_constants, iapws_
↪correlations
>>> liquid = IAPWS95Liquid(T=300, P=1e5, zs=[1])
>>> gas = IAPWS95Gas(T=300, P=1e5, zs=[1])
>>> flasher = FlashPureVLS(iapws_constants, iapws_correlations, gas, [liquid], [])
>>> PT = flasher.flash(T=800.0, P=1e7)
```

(continues on next page)

(continued from previous page)

```
>>> PT.rho_mass()
29.1071839176
>>> print(flasher.flash(T=600, VF=.5))
<EquilibriumState, T=600.0000, P=12344824.3572, zs=[1.0], betas=[0.5, 0.5], phases=[
↪<IAPWS95Gas, T=600 K, P=1.23448e+07 Pa>, <IAPWS95Liquid, T=600 K, P=1.23448e+07 Pa>]>
>>> print(flasher.flash(T=600.0, H=50802))
<EquilibriumState, T=600.0000, P=10000469.1288, zs=[1.0], betas=[1.0], phases=[
↪<IAPWS95Gas, T=600 K, P=1.00005e+07 Pa>]>
>>> print(flasher.flash(P=1e7, S=104.))
<EquilibriumState, T=599.6790, P=10000000.0000, zs=[1.0], betas=[1.0], phases=[
↪<IAPWS95Gas, T=599.679 K, P=1e+07 Pa>]>
>>> print(flasher.flash(V=.00061, U=55850))
<EquilibriumState, T=800.5922, P=10144789.0899, zs=[1.0], betas=[1.0], phases=[
↪<IAPWS95Gas, T=800.592 K, P=1.01448e+07 Pa>]>
```

Not all flash calculations have been fully optimized, but the basic flashes are quite fast.

CHAPTER

SIX

DETAILS OF GIBBSEXCESSLIQUID PHASE MODEL

There are lots of options that get called "ideal". The GibbsExcessLiquid object implements many of them, which means the configuration is complicated and the defaults may not act as expected.

CHAPTER

SEVEN

API REFERENCE

7.1 Activity Coefficients (thermo.activity)

This module contains a base class *GibbsExcess* for handling activity coefficient based models. The design is for a sub-class to provide the minimum possible number of derivatives of Gibbs energy, and for this base class to provide the rest of the methods. An ideal-liquid class with no excess Gibbs energy *IdealSolution* is also available.

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

7.1.1 Base Class

class thermo.activity.**GibbsExcess**

Bases: object

Class for representing an activity coefficient model. While these are typically presented as tools to compute activity coefficients, in truth they are excess Gibbs energy models and activity coefficients are just one derived aspect of them.

This class does not implement any activity coefficient models itself; it must be subclassed by another model. All properties are derived with the CAS SymPy, not relying on any derivations previously published, and checked numerically for consistency.

Different subclasses have different parameter requirements for initialization; *IdealSolution* is available as a simplest model with activity coefficients of 1 to show what needs to be implemented in subclasses. It is also intended subclasses implement the method *to_T_xs*, which creates a new object at the specified temperature and composition but with the same parameters.

These objects are intended to lazy-calculate properties as much as possible, and for the temperature and composition of an object to be immutable.

Methods

CpE()	Calculate and return the first temperature derivative of excess enthalpy of a liquid phase using an activity coefficient model.
HE()	Calculate and return the excess entropy of a liquid phase using an activity coefficient model.
SE()	Calculates the excess entropy of a liquid phase using an activity coefficient model.
as_json()	Method to create a JSON-friendly representation of the Gibbs Excess model which can be stored, and reloaded later.
d2GE_dTdns()	Calculate and return the mole number derivative of the first temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d2nGE_dTdns()	Calculate and return the partial mole number derivative of the first temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d2nGE_dninjs()	Calculate and return the second partial mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
dGE_dns()	Calculate and return the mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
dHE_dT()	Calculate and return the first temperature derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dHE_dns()	Calculate and return the mole number derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dHE_dxs()	Calculate and return the mole fraction derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dSE_dT()	Calculate and return the first temperature derivative of excess entropy of a liquid phase using an activity coefficient model.
dSE_dns()	Calculate and return the mole number derivative of excess entropy of a liquid phase using an activity coefficient model.
dSE_dxs()	Calculate and return the mole fraction derivative of excess entropy of a liquid phase using an activity coefficient model.
dgammas_dT()	Calculate and return the temperature derivatives of activity coefficients of a liquid phase using an activity coefficient model.
dgammas_dns()	Calculate and return the mole number derivative of activity coefficients of a liquid phase using an activity coefficient model.

continues on next page

Table 1 – continued from previous page

dnGE_dns()	Calculate and return the partial mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
dnHE_dns()	Calculate and return the partial mole number derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dnSE_dns()	Calculate and return the partial mole number derivative of excess entropy of a liquid phase using an activity coefficient model.
from_json(json_repr)	Method to create a Gibbs Excess model from a JSON-friendly serialization of another Gibbs Excess model.
gammas()	Calculate and return the activity coefficients of a liquid phase using an activity coefficient model.
gammas_infinite_dilution()	Calculate and return the infinite dilution activity coefficients of each component.
model_hash()	Basic method to calculate a hash of the non-state parts of the model This is useful for comparing to models to determine if they are the same, i.e. in a VLL flash it is important to know if both liquids have the same model.
state_hash()	Basic method to calculate a hash of the state of the model and its model parameters.

CpE()

Calculate and return the first temperature derivative of excess enthalpy of a liquid phase using an activity coefficient model.

$$\frac{\partial h^E}{\partial T} = -T\frac{\partial^2 g^E}{\partial T^2}$$

Returns

dHE_dT [float] First temperature derivative of excess enthalpy of the liquid phase, [J/mol/K]

HE()

Calculate and return the excess entropy of a liquid phase using an activity coefficient model.

$$h^E = -T\frac{\partial g^E}{\partial T} + g^E$$

Returns

HE [float] Excess enthalpy of the liquid phase, [J/mol]

SE()

Calculates the excess entropy of a liquid phase using an activity coefficient model.

$$s^E = \frac{h^E - g^E}{T}$$

Returns

SE [float] Excess entropy of the liquid phase, [J/mol/K]

Notes

Note also the relationship of the expressions for partial excess entropy:

$$S_i^E = -R\left(T\frac{\partial \ln \gamma_i}{\partial T} + \ln \gamma_i\right)$$

__eq__(*other*)
Return self==value.

__hash__()
Method to calculate and return a hash representing the exact state of the object. This includes *T*, *xs*, the model class, and which values have already been calculated.

Returns

hash [int] Hash of the object, [-]

__repr__()
Method to create a string representation of the state of the model. Included is *T*, *xs*, and all constants necessary to create the model. This can be passed into `exec` to re-create the model. Note that parsing strings like this can be slow.

Returns

repr [str] String representation of the object, [-]

Examples

```
>>> IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
```

as_json()
Method to create a JSON-friendly representation of the Gibbs Excess model which can be stored, and reloaded later.

Returns

json_repr [dict] JSON-friendly representation, [-]

Examples

```
>>> import json
>>> model = IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
>>> json_view = model.as_json()
>>> json_str = json.dumps(json_view)
>>> assert type(json_str) is str
>>> model_copy = IdealSolution.from_json(json.loads(json_str))
>>> assert model_copy == model
```

d2GE_dTdns()
Calculate and return the mole number derivative of the first temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial^2 G^E}{\partial n_i \partial T}$$

Returns

d2GE_dTdns [list[float]] First mole number derivative of the temperature derivative of excess Gibbs entropy of the liquid phase, [J/(mol^2*K)]

`d2nGE_dTdns()`

Calculate and return the partial mole number derivative of the first temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial^2 nG^E}{\partial n_i \partial T}$$

Returns

d2nGE_dTdns [list[float]] First partial mole number derivative of the temperature derivative of excess Gibbs entropy of the liquid phase, [J/(mol*K)]

`d2nGE_dninjs()`

Calculate and return the second partial mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial^2 nG^E}{\partial n_i \partial n_i}$$

Returns

d2nGE_dninjs [list[list[float]]] Second partial mole number derivative of excess Gibbs energy of a liquid phase, [J/(mol^2)]

`dGE_dns()`

Calculate and return the mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial G^E}{\partial n_i}$$

Returns

dGE_dns [list[float]] First mole number derivative of excess Gibbs entropy of the liquid phase, [J/(mol^2*K)]

`dHE_dT()`

Calculate and return the first temperature derivative of excess enthalpy of a liquid phase using an activity coefficient model.

$$\frac{\partial h^E}{\partial T} = -T\frac{\partial^2 g^E}{\partial T^2}$$

Returns

dHE_dT [float] First temperature derivative of excess enthalpy of the liquid phase, [J/mol/K]

`dHE_dns()`

Calculate and return the mole number derivative of excess enthalpy of a liquid phase using an activity coefficient model.

$$\frac{\partial h^E}{\partial n_i}$$

Returns

dHE_dns [list[float]] First mole number derivative of excess enthalpy of the liquid phase, [J/mol^2]

dHE_dxs()

Calculate and return the mole fraction derivative of excess enthalpy of a liquid phase using an activity coefficient model.

$$\frac{\partial h^E}{\partial x_i} = -T\frac{\partial^2 g^E}{\partial T \partial x_i} + \frac{\partial g^E}{\partial x_i}$$

Returns

dHE_dxs [list[float]] First mole fraction derivative of excess enthalpy of the liquid phase, [J/mol]

dSE_dT()

Calculate and return the first temperature derivative of excess entropy of a liquid phase using an activity coefficient model.

$$\frac{\partial s^E}{\partial T} = \frac{1}{T}\left(\frac{-\partial g^E}{\partial T} + \frac{\partial h^E}{\partial T} - \frac{(G+H)}{T}\right)$$

Returns

dSE_dT [float] First temperature derivative of excess entropy of the liquid phase, [J/mol/K]

dSE_dns()

Calculate and return the mole number derivative of excess entropy of a liquid phase using an activity coefficient model.

$$\frac{\partial S^E}{\partial n_i}$$

Returns

dSE_dns [list[float]] First mole number derivative of excess entropy of the liquid phase, [J/(mol^2*K)]

dSE_dxs()

Calculate and return the mole fraction derivative of excess entropy of a liquid phase using an activity coefficient model.

$$\frac{\partial S^E}{\partial x_i} = \frac{1}{T}\left(\frac{\partial h^E}{\partial x_i} - \frac{\partial g^E}{\partial x_i}\right) = -\frac{\partial^2 g^E}{\partial x_i \partial T}$$

Returns

dSE_dxs [list[float]] First mole fraction derivative of excess entropy of the liquid phase, [J/(mol*K)]

dgammas_dT()

Calculate and return the temperature derivatives of activity coefficients of a liquid phase using an activity coefficient model.

$$\frac{\partial \gamma_i}{\partial T} = \left(\frac{\frac{\partial^2 nG^E}{\partial T \partial n_i}}{RT} - \frac{\frac{\partial n_i G^E}{\partial n_i}}{RT^2}\right)\exp\left(\frac{\frac{\partial n_i G^E}{\partial n_i}}{RT}\right)$$

Returns

dgammas_dT [list[float]] Temperature derivatives of activity coefficients, [1/K]

dgammas_dns()

Calculate and return the mole number derivative of activity coefficients of a liquid phase using an activity coefficient model.

$$\frac{\partial \gamma_i}{\partial n_i} = \gamma_i\left(\frac{\frac{\partial^2 G^E}{\partial x_i \partial x_j}}{RT}\right)$$

Returns

dgammas_dns [list[list[float]]] Mole number derivatives of activity coefficients, [1/mol]

`dnGE_dns()`

Calculate and return the partial mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial nG^E}{\partial n_i}$$

Returns

dnGE_dns [list[float]] First partial mole number derivative of excess Gibbs entropy of the liquid phase, [J/(mol)]

`dnHE_dns()`

Calculate and return the partial mole number derivative of excess enthalpy of a liquid phase using an activity coefficient model.

$$\frac{\partial nh^E}{\partial n_i}$$

Returns

dnHE_dns [list[float]] First partial mole number derivative of excess enthalpy of the liquid phase, [J/mol]

`dnSE_dns()`

Calculate and return the partial mole number derivative of excess entropy of a liquid phase using an activity coefficient model.

$$\frac{\partial nS^E}{\partial n_i}$$

Returns

dnSE_dns [list[float]] First partial mole number derivative of excess entropy of the liquid phase, [J/(mol*K)]

`classmethod from_json(json_repr)`

Method to create a Gibbs Excess model from a JSON-friendly serialization of another Gibbs Excess model.

Parameters

json_repr [dict] JSON-friendly representation, [-]

Returns

model [`GibbsExcess`] Newly created object from the json serialization, [-]

Notes

It is important that the input string be in the same format as that created by `GibbsExcess.as_json`.

Examples

```
>>> model = IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
>>> json_view = model.as_json()
>>> new_model = IdealSolution.from_json(json_view)
>>> assert model == new_model
```

gammas()

Calculate and return the activity coefficients of a liquid phase using an activity coefficient model.

$$\gamma_i = \exp\left(\frac{\frac{\partial n_i G^E}{\partial n_i}}{RT}\right)$$

Returns

gammas [list[float]] Activity coefficients, [-]

gammas_infinite_dilution()

Calculate and return the infinite dilution activity coefficients of each component.

Returns

gammas_infinite [list[float]] Infinite dilution activity coefficients, [-]

Notes

The algorithm is as follows. For each component, set its composition to zero. Normalize the remaining compositions to 1. Create a new object with that composition, and calculate the activity coefficient of the component whose concentration was set to zero.

model_hash()

Basic method to calculate a hash of the non-state parts of the model This is useful for comparing to models to determine if they are the same, i.e. in a VLL flash it is important to know if both liquids have the same model.

Note that the hashes should only be compared on the same system running in the same process!

Returns

model_hash [int] Hash of the object's model parameters, [-]

state_hash()

Basic method to calculate a hash of the state of the model and its model parameters.

Note that the hashes should only be compared on the same system running in the same process!

Returns

state_hash [int] Hash of the object's model parameters and state, [-]

7.1.2 Ideal Liquid Class

class thermo.activity.**IdealSolution**(*T=None*, *xs=None*)

Bases: *thermo.activity.GibbsExcess*

Class for representing an ideal liquid, with no excess gibbs energy and thus activity coefficients of 1.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Examples

```
>>> model = IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
>>> model.GE()
0.0
>>> model.gammas()
[1.0, 1.0, 1.0, 1.0]
>>> model.dgammas_dT()
[0.0, 0.0, 0.0, 0.0]
```

Attributes

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Methods

GE()	Calculate and return the excess Gibbs energy of a liquid phase using an activity coefficient model.
d2GE_dT2()	Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d2GE_dTdxs()	Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy of an ideal liquid.
d2GE_dxixjs()	Calculate and return the second mole fraction derivatives of excess Gibbs energy of an ideal liquid.
d3GE_dT3()	Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d3GE_dxixjxks()	Calculate and return the third mole fraction derivatives of excess Gibbs energy of an ideal liquid.
dGE_dT()	Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
dGE_dxs()	Calculate and return the mole fraction derivatives of excess Gibbs energy of an ideal liquid.

continues on next page

Table 2 – continued from previous page

to_T_xs(T, xs)	Method to construct a new IdealSolution instance at temperature T, and mole fractions xs with the same parameters as the existing object.

GE()

Calculate and return the excess Gibbs energy of a liquid phase using an activity coefficient model.

$$g^E = 0$$

Returns

GE [float] Excess Gibbs energy of an ideal liquid, [J/mol]

d2GE_dT2()

Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial^2 g^E}{\partial T^2} = 0$$

Returns

d2GE_dT2 [float] Second temperature derivative of excess Gibbs energy of an ideal liquid, [J/(mol*K^2)]

d2GE_dTdxs()

Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy of an ideal liquid.

$$\frac{\partial^2 g^E}{\partial x_i \partial T} = 0$$

Returns

d2GE_dTdxs [list[float]] Temperature derivative of mole fraction derivatives of excess Gibbs energy of an ideal liquid, [J/(mol*K)]

d2GE_dxixjs()

Calculate and return the second mole fraction derivatives of excess Gibbs energy of an ideal liquid.

$$\frac{\partial^2 g^E}{\partial x_i \partial x_j} = 0$$

Returns

d2GE_dxixjs [list[list[float]]] Second mole fraction derivatives of excess Gibbs energy of an ideal liquid, [J/mol]

d3GE_dT3()

Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial^3 g^E}{\partial T^3} = 0$$

Returns

d3GE_dT3 [float] Third temperature derivative of excess Gibbs energy of an ideal liquid, [J/(mol*K^3)]

d3GE_dxixjxks()

Calculate and return the third mole fraction derivatives of excess Gibbs energy of an ideal liquid.

$$\frac{\partial^3 g^E}{\partial x_i \partial x_j \partial x_k} = 0$$

Returns

d3GE_dxixjxks [list[list[list[float]]]] Third mole fraction derivatives of excess Gibbs energy of an ideal liquid, [J/mol]

dGE_dT()

Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.

$$\frac{\partial g^E}{\partial T} = 0$$

Returns

dGE_dT [float] First temperature derivative of excess Gibbs energy of an ideal liquid, [J/(mol*K)]

dGE_dxs()

Calculate and return the mole fraction derivatives of excess Gibbs energy of an ideal liquid.

$$\frac{\partial g^E}{\partial x_i} = 0$$

Returns

dGE_dxs [list[float]] Mole fraction derivatives of excess Gibbs energy of an ideal liquid, [J/mol]

to_T_xs(*T*, *xs*)

Method to construct a new `IdealSolution` instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions of each component, [-]

Returns

obj [IdealSolution] New `IdealSolution` object at the specified conditions [-]

Examples

```
>>> p = IdealSolution(T=300.0, xs=[.1, .2, .3, .4])
>>> p.to_T_xs(T=500.0, xs=[.25, .25, .25, .25])
IdealSolution(T=500.0, xs=[0.25, 0.25, 0.25, 0.25])
```

7.1.3 Notes

Excellent references for working with activity coefficient models are [1] and [2].

References

7.2 Bulk Phases (thermo.bulk)

This module contains a phase wrapper for obtaining properties of a pseudo-phase made of multiple other phases. This is useful in the context of multiple liquid phases; or multiple solid phases; or looking at all the phases together.

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

7.2.1 Bulk Class

class thermo.bulk.**Bulk**(*T*, *P*, *zs*, *phases*, *phase_fractions*, *phase_bulk=None*)

Bases: `thermo.phases.phase.Phase`

Class to encapsulate multiple `Phase` objects and provide a unified interface for obtaining properties from a group of phases.

This class exists for three purposes:

- Providing a common interface for obtaining properties like *Cp* - whether there is one phase or 100, calling *Cp* on the bulk will retrieve that value.
- Retrieving "bulk" properties that do make sense to be calculated for a combination of phases together.
- Allowing configurable estimations of non-bulk properties like isothermal compressibility or speed of sound for the group of phases together.

Parameters

T [float] Temperature of the bulk, [K]

P [float] Pressure of the bulk, [Pa]

zs [list[float]] Mole fractions of the bulk, [-]

phases [list[`Phase`]] Phase objects, [-]

phase_fractions [list[float]] Molar fractions of each phase, [-]

phase_bulk [str, optional] None to represent a bulk of all present phases; 'l' to represent a bulk of only liquid phases; *s* to represent a bulk of only solid phases, [-]

Notes

Please think carefully when retrieving a property of the bulk. If there are two liquid phases in a bulk, and a single viscosity value is retrieved, can that be used directly for a single phase pressure drop calculation? Not with any theoretical consistency, that's for sure.

Attributes

beta Phase fraction of the bulk phase.

betas_mass Method to calculate and return the mass fraction of all of the phases in the bulk.

betas_volume Method to calculate and return the volume fraction of all of the phases in the bulk.

Methods

Cp()	Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase.
Cp_ideal_gas()	Method to calculate and return the ideal-gas heat capacity of the phase.
H()	Method to calculate and return the constant-temperature and constant phase-fraction enthalpy of the bulk phase.
H_ideal_gas()	Method to calculate and return the ideal-gas enthalpy of the phase.
H_reactive()	Method to calculate and return the constant-temperature and constant phase-fraction reactive enthalpy of the bulk phase.
Joule_Thomson()	Method to calculate and return the Joule-Thomson coefficient of the bulk according to the selected calculation methodology.
MW()	Method to calculate and return the molecular weight of the bulk phase.
Pmc()	Method to calculate and return the mechanical critical pressure of the phase.
S()	Method to calculate and return the constant-temperature and constant phase-fraction entropy of the bulk phase.
S_ideal_gas()	Method to calculate and return the ideal-gas entropy of the phase.
S_reactive()	Method to calculate and return the constant-temperature and constant phase-fraction reactive entropy of the bulk phase.
Tmc()	Method to calculate and return the mechanical critical temperature of the phase.
V()	Method to calculate and return the molar volume of the bulk phase.
V_iter([force])	Method to calculate and return the molar volume of the bulk phase, with precision suitable for a *TV* calculation to calculate a matching pressure.

continues on next page

Table 3 – continued from previous page

Vmc()	Method to calculate and return the mechanical critical volume of the phase.
Zmc()	Method to calculate and return the mechanical critical compressibility of the phase.
d2P_dT2()	Method to calculate and return the second temperature derivative of pressure of the bulk according to the selected calculation methodology.
d2P_dT2_frozen()	Method to calculate and return the second constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions.
d2P_dTdV()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the bulk according to the selected calculation methodology.
d2P_dTdV_frozen()	Method to calculate and return the second derivative of pressure with respect to volume and temperature of the bulk phase, at constant phase fractions and phase compositions.
d2P_dV2()	Method to calculate and return the second volume derivative of pressure of the bulk according to the selected calculation methodology.
d2P_dV2_frozen()	Method to calculate and return the constant-temperature second derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions.
dA_dP()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.
dA_dT()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dG_dP()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dT()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.
dP_dT()	Method to calculate and return the first temperature derivative of pressure of the bulk according to the selected calculation methodology.
dP_dT_frozen()	Method to calculate and return the constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions.
dP_dV()	Method to calculate and return the first volume derivative of pressure of the bulk according to the selected calculation methodology.
dP_dV_frozen()	Method to calculate and return the constant-temperature derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions.

continues on next page

Table 3 – continued from previous page

dU_dP()	Method to calculate and return the constant-temperature pressure derivative of internal energy.
dU_dT()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
isobaric_expansion()	Method to calculate and return the isobatic expansion coefficient of the bulk according to the selected calculation methodology.
k()	Calculate and return the thermal conductivity of the bulk according to the selected thermal conductivity settings in BulkSettings, the settings in ThermalConductivityGasMixture and ThermalConductivityLiquidMixture, and the configured pure-component settings in ThermalConductivityGas and ThermalConductivityLiquid.
kappa()	Method to calculate and return the isothermal compressibility of the bulk according to the selected calculation methodology.
mu()	Calculate and return the viscosity of the bulk according to the selected viscosity settings in BulkSettings, the settings in ViscosityGasMixture and ViscosityLiquidMixture, and the configured pure-component settings in ViscosityGas and ViscosityLiquid.
sigma()	Calculate and return the surface tension of the bulk according to the selected surface tension settings in BulkSettings, the settings in SurfaceTensionMixture and the configured pure-component settings in SurfaceTension.
speed_of_sound()	Method to calculate and return the molar speed of sound of the bulk according to the selected calculation methodology.

Cp()

Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase. This is a phase-fraction weighted calculation.

$$C_p = \sum_i^p C_{p,i}\beta_i$$

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

Cp_ideal_gas()

Method to calculate and return the ideal-gas heat capacity of the phase.

$$C_p^{ig} = \sum_i z_i C_{p,i}^{ig}$$

Returns

Cp [float] Ideal gas heat capacity, [J/(mol*K)]

H()

Method to calculate and return the constant-temperature and constant phase-fraction enthalpy of the bulk phase. This is a phase-fraction weighted calculation.

$$H = \sum_i^p H_i \beta_i$$

Returns

H [float] Molar enthalpy, [J/(mol)]

H_ideal_gas()

Method to calculate and return the ideal-gas enthalpy of the phase.

$$H^{ig} = \sum_i z_i H_i^{ig}$$

Returns

H [float] Ideal gas enthalpy, [J/(mol)]

H_reactive()

Method to calculate and return the constant-temperature and constant phase-fraction reactive enthalpy of the bulk phase. This is a phase-fraction weighted calculation.

$$H_{\text{reactive}} = \sum_i^p H_{\text{reactive},i} \beta_i$$

Returns

H_reactive [float] Reactive molar enthalpy, [J/(mol)]

Joule_Thomson()

Method to calculate and return the Joule-Thomson coefficient of the bulk according to the selected calculation methodology.

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H$$

Returns

mu_JT [float] Joule-Thomson coefficient [K/Pa]

MW()

Method to calculate and return the molecular weight of the bulk phase. This is a phase-fraction weighted calculation.

$$\text{MW} = \sum_i^p \text{MW}_i \beta_i$$

Returns

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

Pmc()

Method to calculate and return the mechanical critical pressure of the phase.

Returns

Pmc [float] Mechanical critical pressure, [Pa]

S()

Method to calculate and return the constant-temperature and constant phase-fraction entropy of the bulk phase. This is a phase-fraction weighted calculation.

$$S = \sum_i^p S_i \beta_i$$

Returns

S [float] Molar entropy, [J/(mol*K)]

S_ideal_gas()

Method to calculate and return the ideal-gas entropy of the phase.

$$S^{ig} = \sum_i z_i S_i^{ig} - R\ln\left(\frac{P}{P_{ref}}\right) - R\sum_i z_i \ln(z_i)$$

Returns

S [float] Ideal gas molar entropy, [J/(mol*K)]

S_reactive()

Method to calculate and return the constant-temperature and constant phase-fraction reactive entropy of the bulk phase. This is a phase-fraction weighted calculation.

$$S_{\text{reactive}} = \sum_i^p S_{\text{reactive},i} \beta_i$$

Returns

S_reactive [float] Reactive molar entropy, [J/(mol*K)]

Tmc()

Method to calculate and return the mechanical critical temperature of the phase.

Returns

Tmc [float] Mechanical critical temperature, [K]

V()

Method to calculate and return the molar volume of the bulk phase. This is a phase-fraction weighted calculation.

$$V = \sum_i^p V_i \beta_i$$

Returns

V [float] Molar volume, [m^3/mol]

V_iter(*force=False*)

Method to calculate and return the molar volume of the bulk phase, with precision suitable for a *TV* calculation to calculate a matching pressure. This is a phase-fraction weighted calculation.

$$V = \sum_i^p V_i \beta_i$$

Returns

V [float or mpf] Molar volume, [m^3/mol]

`Vmc()`
Method to calculate and return the mechanical critical volume of the phase.

Returns

Vmc [float] Mechanical critical volume, [m^3/mol]

`Zmc()`
Method to calculate and return the mechanical critical compressibility of the phase.

Returns

Zmc [float] Mechanical critical compressibility, [-]

`property beta`
Phase fraction of the bulk phase. Should always be 1 when representing all phases of a flash; but can be less than one if representing multiple solids or liquids as a single phase in a larger mixture.

Returns

beta [float] Phase fraction of bulk, [-]

`property betas_mass`
Method to calculate and return the mass fraction of all of the phases in the bulk.

Returns

betas_mass [list[float]] Mass phase fractions of all the phases in the bulk object, ordered vapor, liquid, then solid, [-]

`property betas_volume`
Method to calculate and return the volume fraction of all of the phases in the bulk.

Returns

betas_volume [list[float]] Volume phase fractions of all the phases in the bulk, ordered vapor, liquid, then solid , [-]

`d2P_dT2()`
Method to calculate and return the second temperature derivative of pressure of the bulk according to the selected calculation methodology.

Returns

d2P_dT2 [float] Second temperature derivative of pressure, [Pa/K^2]

`d2P_dT2_frozen()`
Method to calculate and return the second constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial^2 P}{\partial T^2}\right)_{V,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial^2 P}{\partial T^2}\right)_{i,V_i,\beta_i,zs_i}$$

Returns

d2P_dT2_frozen [float] Frozen constant-volume second derivative of pressure with respect to temperature of the bulk phase, [Pa/K^2]

`d2P_dTdV()`
Method to calculate and return the second derivative of pressure with respect to temperature and volume of the bulk according to the selected calculation methodology.

Returns

d2P_dTdV [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

d2P_dTdV_frozen()

Method to calculate and return the second derivative of pressure with respect to volume and temperature of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial^2 P}{\partial V \partial T}\right)_{\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial^2 P}{\partial V \partial T}\right)_{i,\beta_i,zs_i}$$

Returns

d2P_dTdV_frozen [float] Frozen second derivative of pressure with respect to volume and temperature of the bulk phase, [Pa*mol^2/m^6]

d2P_dV2()

Method to calculate and return the second volume derivative of pressure of the bulk according to the selected calculation methodology.

Returns

d2P_dV2 [float] Second volume derivative of pressure, [Pa*mol^2/m^6]

d2P_dV2_frozen()

Method to calculate and return the constant-temperature second derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial^2 P}{\partial V^2}\right)_{T,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial^2 P}{\partial V^2}\right)_{i,T,\beta_i,zs_i}$$

Returns

d2P_dV2_frozen [float] Frozen constant-temperature second derivative of pressure with respect to volume of the bulk phase, [Pa*mol^2/m^6]

dA_dP()

Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial U}{\partial P}\right)_T$$

Returns

dA_dP [float] Constant-temperature pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dT()

Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial U}{\partial T}\right)_P$$

Returns

dA_dT [float] Constant-pressure temperature derivative of Helmholtz energy, [J/(mol*K)]

dG_dP()

Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dG_dP [float] Constant-temperature pressure derivative of Gibbs free energy, [J/(mol*Pa)]

`dG_dT()`

Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dG_dT [float] Constant-pressure temperature derivative of Gibbs free energy, [J/(mol*K)]

`dP_dT()`

Method to calculate and return the first temperature derivative of pressure of the bulk according to the selected calculation methodology.

Returns

dP_dT [float] First temperature derivative of pressure, [Pa/K]

`dP_dT_frozen()`

Method to calculate and return the constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial P}{\partial T}\right)_{V,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial P}{\partial T}\right)_{i,V_i,\beta_i,zs_i}$$

Returns

dP_dT_frozen [float] Frozen constant-volume derivative of pressure with respect to temperature of the bulk phase, [Pa/K]

`dP_dV()`

Method to calculate and return the first volume derivative of pressure of the bulk according to the selected calculation methodology.

Returns

dP_dV [float] First volume derivative of pressure, [Pa*mol/m^3]

`dP_dV_frozen()`

Method to calculate and return the constant-temperature derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial P}{\partial V}\right)_{T,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial P}{\partial V}\right)_{i,T,\beta_i,zs_i}$$

Returns

dP_dV_frozen [float] Frozen constant-temperature derivative of pressure with respect to volume of the bulk phase, [Pa*mol/m^3]

`dU_dP()`

Method to calculate and return the constant-temperature pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_T = -P\left(\frac{\partial V}{\partial P}\right)_T - V + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dU_dP [float] Constant-temperature pressure derivative of internal energy, [J/(mol*Pa)]

dU_dT()

Method to calculate and return the constant-pressure temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_P = -P\left(\frac{\partial V}{\partial T}\right)_P + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dU_dT [float] Constant-pressure temperature derivative of internal energy, [J/(mol*K)]

isobaric_expansion()

Method to calculate and return the isobatic expansion coefficient of the bulk according to the selected calculation methodology.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Returns

beta [float] Isobaric coefficient of a thermal expansion, [1/K]

k()

Calculate and return the thermal conductivity of the bulk according to the selected thermal conductivity settings in *BulkSettings*, the settings in *ThermalConductivityGasMixture* and *ThermalConductivityLiquidMixture*, and the configured pure-component settings in *ThermalConductivityGas* and *ThermalConductivityLiquid*.

Returns

k [float] Thermal Conductivity of bulk phase calculated with mixing rules, [Pa*s]

kappa()

Method to calculate and return the isothermal compressibility of the bulk according to the selected calculation methodology.

$$\kappa = -\frac{1}{V}\left(\frac{\partial V}{\partial P}\right)_T$$

Returns

kappa [float] Isothermal coefficient of compressibility, [1/Pa]

mu()

Calculate and return the viscosity of the bulk according to the selected viscosity settings in *BulkSettings*, the settings in *ViscosityGasMixture* and *ViscosityLiquidMixture*, and the configured pure-component settings in *ViscosityGas* and *ViscosityLiquid*.

Returns

mu [float] Viscosity of bulk phase calculated with mixing rules, [Pa*s]

sigma()

Calculate and return the surface tension of the bulk according to the selected surface tension settings in *BulkSettings*, the settings in *SurfaceTensionMixture* and the configured pure-component settings in *SurfaceTension*.

Returns

sigma [float] Surface tension of bulk phase calculated with mixing rules, [N/m]

Notes

A value is only returned if all phases in the bulk are liquids; this property is for a liquid-ideal gas calculation, not the interfacial tension between two liquid phases.

speed_of_sound()

Method to calculate and return the molar speed of sound of the bulk according to the selected calculation methodology.

$$w = \left[-V^2 \left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

A similar expression based on molar density is:

$$w = \left[\left(\frac{\partial P}{\partial \rho}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

Returns

w [float] Speed of sound for a real gas, [m*kg^0.5/(s*mol^0.5)]

7.2.2 Bulk Settings Class

class thermo.bulk.**BulkSettings**(*dP_dT='MOLE_WEIGHTED', dP_dV='MOLE_WEIGHTED', d2P_dV2='MOLE_WEIGHTED', d2P_dT2='MOLE_WEIGHTED', d2P_dTdV='MOLE_WEIGHTED', mu_LL='LOG_PROP_MASS_WEIGHTED', mu_LL_power_exponent=0.4, mu_VL='McAdams', mu_VL_power_exponent=0.4, k_LL='MASS_WEIGHTED', k_LL_power_exponent=0.4, k_VL='MASS_WEIGHTED', k_VL_power_exponent=0.4, sigma_LL='MASS_WEIGHTED', sigma_LL_power_exponent=0.4, T_liquid_volume_ref=298.15, T_normal=273.15, P_normal=101325.0, T_standard=288.15, P_standard=101325.0, T_gas_ref=288.15, P_gas_ref=101325.0, speed_of_sound='MOLE_WEIGHTED', kappa='MOLE_WEIGHTED', isobaric_expansion='MOLE_WEIGHTED', Joule_Thomson='MOLE_WEIGHTED', VL_ID='PIP', VL_ID_settings=None, S_ID='d2P_dVdT', S_ID_settings=None, solid_sort_method='prop', liquid_sort_method='prop', liquid_sort_cmps=[], solid_sort_cmps=[], liquid_sort_cmps_neg=[], solid_sort_cmps_neg=[], liquid_sort_prop='DENSITY_MASS', solid_sort_prop='DENSITY_MASS', phase_sort_higher_first=True, water_sort='water not special', equilibrium_perturbation=1e-07*)

Bases: object

Class containing configuration methods for determining how properties of a *Bulk* phase made of different phases are handled. All parameters are also attributes.

Parameters

dP_dT [str, optional] The method used to calculate the constant-volume temperature derivative of pressure of the bulk. One of *DP_DT_METHODS*, [-]

dP_dV [str, optional] The method used to calculate the constant-temperature volume derivative of pressure of the bulk. One of *DP_DV_METHODS*, [-]

d2P_dV2 [str, optional] The method used to calculate the second constant-temperature volume derivative of pressure of the bulk. One of *D2P_DV2_METHODS*, [-]

d2P_dT2 [str, optional] The method used to calculate the second constant-volume temperature derivative of pressure of the bulk. One of *D2P_DT2_METHODS*, [-]

d2P_dTdV [str, optional] The method used to calculate the temperature and volume derivative of pressure of the bulk. One of *D2P_DTDV_METHODS*, [-]

T_liquid_volume_ref [float, optional] Liquid molar volume reference temperature; if this is 298.15 K exactly, the molar volumes in `Vml_STPs` will be used, and if it is 288.70555555555555 K exactly, `Vml_60Fs` will be used, and otherwise the molar liquid volumes will be obtained from the temperature-dependent correlations specified, [K]

T_gas_ref [float, optional] Reference temperature to use for the calculation of ideal-gas molar volume and flow rate, [K]

P_gas_ref [float, optional] Reference pressure to use for the calculation of ideal-gas molar volume and flow rate, [Pa]

T_normal [float, optional] "Normal" gas reference temperature for the calculation of ideal-gas molar volume in the "normal" reference state; default 273.15 K (0 C) according to [1], [K]

P_normal [float, optional] "Normal" gas reference pressure for the calculation of ideal-gas molar volume in the "normal" reference state; default 101325 Pa (1 atm) according to [1], [Pa]

T_standard [float, optional] "Standard" gas reference temperature for the calculation of ideal-gas molar volume in the "standard" reference state; default 288.15 K (15° C) according to [2]; 288.70555555555555 is also often used (60° F), [K]

P_standard [float, optional] "Standard" gas reference pressure for the calculation of ideal-gas molar volume in the "standard" reference state; default 101325 Pa (1 atm) according to [2], [Pa]

mu_LL [str, optional] Mixing rule for multiple liquid phase liquid viscosity calculations; see *MU_LL_METHODS* for available options, [-]

mu_LL_power_exponent [float, optional] Liquid-liquid viscosity power-law mixing parameter, used only when a power law mixing rule is selected, [-]

mu_VL [str, optional] Mixing rule for vapor-liquid viscosity calculations; see *MU_VL_METHODS* for available options, [-]

mu_VL_power_exponent [float, optional] Vapor-liquid viscosity power-law mixing parameter, used only when a power law mixing rule is selected, [-]

k_LL [str, optional] Mixing rule for multiple liquid phase liquid thermal conductivity calculations; see *K_LL_METHODS* for available options, [-]

k_LL_power_exponent [float, optional] Liquid-liquid thermal conductivity power-law mixing parameter, used only when a power law mixing rule is selected, [-]

k_VL [str, optional] Mixing rule for vapor-liquid thermal conductivity calculations; see *K_VL_METHODS* for available options, [-]

k_VL_power_exponent [float, optional] Vapor-liquid thermal conductivity power-law mixing parameter, used only when a power law mixing rule is selected, [-]

sigma_LL [str, optional] Mixing rule for multiple liquid phase, air-liquid surface tension calculations; see *SIGMA_LL_METHODS* for available options, [-]

sigma_LL_power_exponent [float, optional] Air-liquid Liquid-liquid surface tension power-law mixing parameter, used only when a power law mixing rule is selected, [-]

equilibrium_perturbation [float, optional] The relative perturbation to use when calculating equilibrium derivatives numerically; for example if this is 1e-3 and T is the perturbation variable and the statis is 500 K, the perturbation calculation temperature will be 500.5 K, [various]

isobaric_expansion [str, optional] Mixing rule for multiphase isobaric expansion calculations; see *BETA_METHODS* for available options, [-]

speed_of_sound [str, optional] Mixing rule for multiphase speed of sound calculations; see *SPEED_OF_SOUND_METHODS* for available options, [-]

kappa [str, optional] Mixing rule for multiphase *kappa* calculations; see *KAPPA_METHODS* for available options, [-]

Joule_Thomson [str, optional] Mixing rule for multiphase *Joule-Thomson* calculations; see *JT_METHODS* for available options, [-]

Notes

The linear mixing rules "MOLE_WEIGHTED", "MASS_WEIGHTED", and "VOLUME_WEIGHTED" have the following formula, with β representing molar, mass, or volume phase fraction:

$$\text{bulk property} = \left(\sum_i^{phases} \beta_i \text{property}\right)$$

The power mixing rules "POWER_PROP_MOLE_WEIGHTED", "POWER_PROP_MASS_WEIGHTED", and "POWER_PROP_VOLUME_WEIGHTED" have the following formula, with β representing molar, mass, or volume phase fraction:

$$\text{bulk property} = \left(\sum_i^{phases} \beta_i \text{property}^{\text{exponent}}\right)^{1/\text{exponent}}$$

The logarithmic mixing rules "LOG_PROP_MOLE_WEIGHTED", "LOG_PROP_MASS_WEIGHTED", and "LOG_PROP_VOLUME_WEIGHTED" have the following formula, with β representing molar, mass, or volume phase fraction:

$$\text{bulk property} = \exp\left(\sum_i^{phases} \beta_i \ln(\text{property})\right)$$

The mixing rule "MINIMUM_PHASE_PROP" selects the lowest phase value of the property, always. The mixing rule "MAXIMUM_PHASE_PROP" selects the highest phase value of the property, always.

The mixing rule "AS_ONE_LIQUID" calculates a property using the bulk composition but applied to the liquid model only. The mixing rule "AS_ONE_GAS" calculates a property using the bulk composition but applied to the gas model only.

The mixing rule "FROM_DERIVATIVE_SETTINGS" is used to indicate that the property depends on other configurable properties; and when this is the specified option, those configurations will be used in the calculation of this property.

The mixing rule "EQUILIBRIUM_DERIVATIVE" performs derivative calculations on flashes themselves. This is quite slow in comparison to other methods.

References

[1], [2]

Methods

as_json	

as_json()

thermo.bulk.**DP_DT_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'EQUILIBRIUM_DERIVATIVE', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented calculation methods for the *DP_DT* bulk setting

thermo.bulk.**DP_DV_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'EQUILIBRIUM_DERIVATIVE', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented calculation methods for the *DP_DV* bulk setting

thermo.bulk.**D2P_DV2_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented calculation methods for the *D2P_DV2* bulk setting

thermo.bulk.**D2P_DT2_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented calculation methods for the *D2P_DT2* bulk setting

thermo.bulk.**D2P_DTDV_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented calculation methods for the *D2P_DTDV* bulk setting

thermo.bulk.**MU_LL_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'AS_ONE_LIQUID', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'POWER_PROP_MOLE_WEIGHTED', 'POWER_PROP_MASS_WEIGHTED', 'POWER_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented mixing rules for the *MU_LL* setting

thermo.bulk.**MU_VL_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'AS_ONE_LIQUID', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'POWER_PROP_MOLE_WEIGHTED', 'POWER_PROP_MASS_WEIGHTED', 'POWER_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP', 'AS_ONE_GAS', 'Beattie Whalley', 'McAdams', 'Cicchitti', 'Lin Kwok', 'Fourar Bories', 'Duckler']**

List of all valid and implemented mixing rules for the *MU_VL* setting

thermo.bulk.**K_LL_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'AS_ONE_LIQUID', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'POWER_PROP_MOLE_WEIGHTED', 'POWER_PROP_MASS_WEIGHTED', 'POWER_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']**

List of all valid and implemented mixing rules for the *K_LL* setting thermo.bulk.K_VL_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'AS_ONE_LIQUID', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'POWER_PROP_MOLE_WEIGHTED', 'POWER_PROP_MASS_WEIGHTED', 'POWER_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP', 'AS_ONE_GAS']

List of all valid and implemented mixing rules for the *K_VL* setting

thermo.bulk.SIGMA_LL_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'AS_ONE_LIQUID', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'POWER_PROP_MOLE_WEIGHTED', 'POWER_PROP_MASS_WEIGHTED', 'POWER_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP']

List of all valid and implemented mixing rules for the *SIGMA_LL* setting

thermo.bulk.BETA_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP', 'EQUILIBRIUM_DERIVATIVE', 'FROM_DERIVATIVE_SETTINGS']

List of all valid and implemented calculation methods for the *isothermal_compressibility* bulk setting

thermo.bulk.SPEED_OF_SOUND_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP', 'FROM_DERIVATIVE_SETTINGS']

List of all valid and implemented calculation methods for the *speed_of_sound* bulk setting

thermo.bulk.KAPPA_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP', 'EQUILIBRIUM_DERIVATIVE', 'FROM_DERIVATIVE_SETTINGS']

List of all valid and implemented calculation methods for the *kappa* bulk setting

thermo.bulk.JT_METHODS = ['MOLE_WEIGHTED', 'MASS_WEIGHTED', 'VOLUME_WEIGHTED', 'LOG_PROP_MOLE_WEIGHTED', 'LOG_PROP_MASS_WEIGHTED', 'LOG_PROP_VOLUME_WEIGHTED', 'MINIMUM_PHASE_PROP', 'MAXIMUM_PHASE_PROP', 'EQUILIBRIUM_DERIVATIVE', 'FROM_DERIVATIVE_SETTINGS']

List of all valid and implemented calculation methods for the *JT* bulk setting

7.3 Legacy Chemicals (thermo.chemical)

class thermo.chemical.**Chemical**(*ID*, *T=298.15*, *P=101325*, *autocalc=True*)

Bases: object

Creates a Chemical object which contains basic information such as molecular weight and the structure of the species, as well as thermodynamic and transport properties as a function of temperature and pressure.

Parameters

ID [str]

One of the following [-]:

- 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)

- CAS number

T [float, optional] Temperature of the chemical (default 298.15 K), [K]

P [float, optional] Pressure of the chemical (default 101325 Pa) [Pa]

Notes

> **Warning:** The Chemical class is not designed for high-performance or the ability to use different thermodynamic models. It is especially limited in its multiphase support and the ability to solve with specifications other than temperature and pressure. It is impossible to change constant properties such as a compound's critical temperature in this interface.
>
> It is recommended to switch over to the *thermo.flash* interface which solves those problems and is better positioned to grow. That interface also requires users to be responsible for their chemical constants and pure component correlations; while default values can easily be loaded for most compounds, the user is ultimately responsible for them.

Examples

Creating chemical objects:

```
>>> Chemical('hexane')
<Chemical [hexane], T=298.15 K, P=101325 Pa>
```

```
>>> Chemical('CCCCCCCC', T=500, P=1E7)
<Chemical [octane], T=500.00 K, P=10000000 Pa>
```

```
>>> Chemical('7440-36-0', P=1000)
<Chemical [antimony], T=298.15 K, P=1000 Pa>
```

Getting basic properties:

```
>>> N2 = Chemical('Nitrogen')
>>> N2.Tm, N2.Tb, N2.Tc # melting, boiling, and critical points [K]
(63.15, 77.355, 126.2)
>>> N2.Pt, N2.Pc # sublimation and critical pressure [Pa]
(12526.9697368421, 3394387.5)
>>> N2.CAS, N2.formula, N2.InChI, N2.smiles, N2.atoms # CAS number, formula, InChI␣
↪string, smiles string, dictionary of atomic elements and their count
('7727-37-9', 'N2', 'N2/c1-2', 'N#N', {'N': 2})
```

Changing the T/P of the chemical, and gettign temperature-dependent properties:

```
>>> N2.Cp, N2.rho, N2.mu # Heat capacity [J/kg/K], density [kg/m^3], viscosity␣
↪[Pa*s]
(1039.4978324480921, 1.1452416223829405, 1.7804740647270688e-05)
>>> N2.calculate(T=65, P=1E6) # set it to a liquid at 65 K and 1 MPa
>>> N2.phase
'l'
>>> N2.Cp, N2.rho, N2.mu # properties are now of the liquid phase
(2002.8819854804037, 861.3539919443364, 0.0002857739143670701)
```

Molar units are also available for properties:

```
>>> N2.Cpm, N2.Vm, N2.Hvapm # heat capacity [J/mol/K], molar volume [m^3/mol],␣
↪enthalpy of vaporization [J/mol]
(56.10753421205674, 3.252251717875631e-05, 5982.710998291719)
```

A great deal of properties are available; for a complete list look at the attributes list.

```
>>> N2.alpha, N2.JT # thermal diffusivity [m^2/s], Joule-Thompson coefficient [K/Pa]
(9.874883993253272e-08, -4.0009932695519242e-07)
```

```
>>> N2.isentropic_exponent, N2.isobaric_expansion
(1.4000000000000001, 0.0047654228408661571)
```

For pure species, the phase is easily identified, allowing for properties to be obtained without needing to specify the phase. However, the properties are also available in the hypothetical gas phase (when under the boiling point) and in the hypothetical liquid phase (when above the boiling point) as these properties are needed to evaluate mixture properties. Specify the phase of a property to be retrieved by appending 'l' or 'g' or 's' to the property.

```
>>> tol = Chemical('toluene')
```

```
>>> tol.rhog, tol.Cpg, tol.kg, tol.mug
(4.241646701894199, 1126.5533755283168, 0.00941385692301755, 6.973325939594919e-06)
```

Temperature dependent properties are calculated by objects which provide many useful features related to the properties. To determine the temperature at which nitrogen has a saturation pressure of 1 MPa:

```
>>> N2.VaporPressure.solve_property(1E6)
103.73528598652341
```

To compute an integral of the ideal-gas heat capacity of nitrogen to determine the enthalpy required for a given change in temperature. Note the thermodynamic objects calculate values in molar units always.

```
>>> N2.HeatCapacityGas.T_dependent_property_integral(100, 120) # J/mol/K
582.0121860897898
```

Derivatives of properties can be calculated as well, as may be needed by for example heat transfer calculations:

```
>>> N2.SurfaceTension.T_dependent_property_derivative(77)
-0.00022695346296730534
```

If a property is needed at multiple temperatures or pressures, it is faster to use the object directly to perform the calculation rather than setting the conditions for the chemical.

```
>>> [N2.VaporPressure(T) for T in range(80, 120, 10)]
[136979.4840843189, 360712.5746603142, 778846.276691705, 1466996.7208525643]
```

These objects are also how the methods by which the properties are calculated can be changed. To see the available methods for a property:

```
>>> N2.VaporPressure.all_methods
set(['VDI_PPDS', 'BOILING_CRITICAL', 'WAGNER_MCGARRY', 'AMBROSE_WALTON', 'COOLPROP',
↪ 'LEE_KESLER_PSAT', 'EOS', 'ANTOINE_POLING', 'SANJARI', 'DIPPR_PERRY_8E', 'Edalat
↪', 'WAGNER_POLING'])
```

To specify the method which should be used for calculations of a property. In the example below, the Lee-kesler correlation for vapor pressure is specified.

```
>>> N2.calculate(80)
>>> N2.Psat
136979.4840843189
>>> N2.VaporPressure.method = 'LEE_KESLER_PSAT'
>>> N2.Psat
134987.76815364443
```

For memory reduction, these objects are shared by all chemicals which are the same; new instances will use the same specified methods.

```
>>> N2_2 = Chemical('nitrogen')
>>> N2_2.VaporPressure.user_methods
['LEE_KESLER_PSAT']
```

To disable this behavior, set thermo.chemical.caching to False.

```
>>> import thermo
>>> thermo.chemical.caching = False
>>> N2_3 = Chemical('nitrogen')
>>> N2_3.VaporPressure.user_methods
[]
```

Properties may also be plotted via these objects:

```
>>> N2.VaporPressure.plot_T_dependent_property()
>>> N2.VolumeLiquid.plot_isotherm(T=77, Pmin=1E5, Pmax=1E7)
>>> N2.VolumeLiquid.plot_isobar(P=1E6,  Tmin=66, Tmax=120)
>>> N2.VolumeLiquid.plot_TP_dependent_property(Tmin=60, Tmax=100,  Pmin=1E5,␣
↪Pmax=1E7)
```

Attributes

T [float] Temperature of the chemical, [K]

P [float] Pressure of the chemical, [Pa]

phase [str] Phase of the chemical; one of 's', 'l', 'g', or 'l/g'.

ID [str] User specified string by which the chemical's CAS was looked up.

CAS [str] The CAS number of the chemical.

PubChem [int] PubChem Compound identifier (CID) of the chemical; all chemicals are sourced from their database. Chemicals can be looked at online at https://pubchem.ncbi.nlm.nih.gov.

MW [float] Molecular weight of the compound, [g/mol]

formula [str] Molecular formula of the compound.

atoms [dict] dictionary of counts of individual atoms, indexed by symbol with proper capitalization, [-]

similarity_variable [float] Similarity variable, see `chemicals.elements.similarity_variable` for the definition, [mol/g]

smiles [str] Simplified molecular-input line-entry system representation of the compound.

InChI [str] IUPAC International Chemical Identifier of the compound.

InChI_Key [str] 25-character hash of the compound's InChI.

IUPAC_name [str] Preferred IUPAC name for a compound.

synonyms [list of strings] All synonyms for the compound found in PubChem, sorted by popularity.

Tm [float] Melting temperature [K]

Tb [float] Boiling temperature [K]

Tc [float] Critical temperature [K]

Pc [float] Critical pressure [Pa]

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

Zc [float] Critical compressibility [-]

rhoc [float] Critical density [kg/m^3]

rhocm [float] Critical molar density [mol/m^3]

omega [float] Acentric factor [-]

StielPolar [float] Stiel Polar factor, see `chemicals.acentric.Stiel_polar_factor` for the definition [-]

Tt [float] Triple temperature, [K]

Pt [float] Triple pressure, [Pa]

Hfus [float] Enthalpy of fusion [J/kg]

Hfusm [float] Molar enthalpy of fusion [J/mol]

Hsub [float] Enthalpy of sublimation [J/kg]

Hsubm [float] Molar enthalpy of sublimation [J/mol]

Hfm [float] Standard state molar enthalpy of formation, [J/mol]

Hf [float] Standard enthalpy of formation in a mass basis, [J/kg]

Hfgm [float] Ideal-gas molar enthalpy of formation, [J/mol]

Hfg [float] Ideal-gas enthalpy of formation in a mass basis, [J/kg]

Hcm [float] Molar higher heat of combustion [J/mol]

Hc [float] Higher Heat of combustion [J/kg]

Hcm_lower [float] Molar lower heat of combustion [J/mol]

Hc_lower [float] Lower Heat of combustion [J/kg]

S0m [float] Standard state absolute molar entropy of the chemical, [J/mol/K]

S0 [float] Standard state absolute entropy of the chemical, [J/kg/K]

S0gm [float] Absolute molar entropy in an ideal gas state of the chemical, [J/mol/K]

S0g [float] Absolute mass entropy in an ideal gas state of the chemical, [J/kg/K]

Gfm [float] Standard state molar change of Gibbs energy of formation [J/mol]

Gf [float] Standard state change of Gibbs energy of formation [J/kg]

Gfgm [float] Ideal-gas molar change of Gibbs energy of formation [J/mol]

Gfg [float] Ideal-gas change of Gibbs energy of formation [J/kg]

Sfm [float] Standard state molar change of entropy of formation, [J/mol/K]

Sf [float] Standard state change of entropy of formation, [J/kg/K]

Sfgm [float] Ideal-gas molar change of entropy of formation, [J/mol/K]

Sfg [float] Ideal-gas change of entropy of formation, [J/kg/K]

Hcgm [float] Higher molar heat of combustion of the chemical in the ideal gas state, [J/mol]

Hcg [float] Higher heat of combustion of the chemical in the ideal gas state, [J/kg]

Hcgm_lower [float] Lower molar heat of combustion of the chemical in the ideal gas state, [J/mol]

Hcg_lower [float] Lower heat of combustion of the chemical in the ideal gas state, [J/kg]

Tflash [float] Flash point of the chemical, [K]

Tautoignition [float] Autoignition point of the chemical, [K]

LFL [float] Lower flammability limit of the gas in an atmosphere at STP, mole fraction [-]

UFL [float] Upper flammability limit of the gas in an atmosphere at STP, mole fraction [-]

TWA [tuple[quantity, unit]] Time-Weighted Average limit on worker exposure to dangerous chemicals.

STEL [tuple[quantity, unit]] Short-term Exposure limit on worker exposure to dangerous chemicals.

Ceiling [tuple[quantity, unit]] Ceiling limits on worker exposure to dangerous chemicals.

Skin [bool] Whether or not a chemical can be absorbed through the skin.

Carcinogen [str or dict] Carcinogen status information.

dipole [float] Dipole moment in debye, [3.33564095198e-30 ampere*second^2]

Stockmayer [float] Lennard-Jones depth of potential-energy minimum over k, [K]

molecular_diameter [float] Lennard-Jones molecular diameter, [angstrom]

GWP [float] Global warming potential (default 100-year outlook) (impact/mass chemical)/(impact/mass CO2), [-]

ODP [float] Ozone Depletion potential (impact/mass chemical)/(impact/mass CFC-11), [-]

logP [float] Octanol-water partition coefficient, [-]

`legal_status` [str or dict] Dictionary of legal status indicators for the chemical.

`economic_status` [list] Dictionary of economic status indicators for the chemical.

RI [float] Refractive Index on the Na D line, [-]

RIT [float] Temperature at which refractive index reading was made

conductivity [float] Electrical conductivity of the fluid, [S/m]

conductivityT [float] Temperature at which conductivity measurement was made

VaporPressure [object] Instance of `thermo.vapor_pressure.VaporPressure`, with data and methods loaded for the chemical; performs the actual calculations of vapor pressure of the chemical.

EnthalpyVaporization [object] Instance of *thermo.phase_change.EnthalpyVaporization*, with data and methods loaded for the chemical; performs the actual calculations of molar enthalpy of vaporization of the chemical.

VolumeSolid [object] Instance of *thermo.volume.VolumeSolid*, with data and methods loaded for the chemical; performs the actual calculations of molar volume of the solid phase of the chemical.

VolumeLiquid [object] Instance of *thermo.volume.VolumeLiquid*, with data and methods loaded for the chemical; performs the actual calculations of molar volume of the liquid phase of the chemical.

VolumeGas [object] Instance of *thermo.volume.VolumeGas*, with data and methods loaded for the chemical; performs the actual calculations of molar volume of the gas phase of the chemical.

HeatCapacitySolid [object] Instance of *thermo.heat_capacity.HeatCapacitySolid*, with data and methods loaded for the chemical; performs the actual calculations of molar heat capacity of the solid phase of the chemical.

HeatCapacityLiquid [object] Instance of *thermo.heat_capacity.HeatCapacityLiquid*, with data and methods loaded for the chemical; performs the actual calculations of molar heat capacity of the liquid phase of the chemical.

HeatCapacityGas [object] Instance of *thermo.heat_capacity.HeatCapacityGas*, with data and methods loaded for the chemical; performs the actual calculations of molar heat capacity of the gas phase of the chemical.

ViscosityLiquid [object] Instance of *thermo.viscosity.ViscosityLiquid*, with data and methods loaded for the chemical; performs the actual calculations of viscosity of the liquid phase of the chemical.

ViscosityGas [object] Instance of *thermo.viscosity.ViscosityGas*, with data and methods loaded for the chemical; performs the actual calculations of viscosity of the gas phase of the chemical.

ThermalConductivityLiquid [object] Instance of *thermo.thermal_conductivity.ThermalConductivityLiquid*, with data and methods loaded for the chemical; performs the actual calculations of thermal conductivity of the liquid phase of the chemical.

ThermalConductivityGas [object] Instance of *thermo.thermal_conductivity.ThermalConductivityGas*, with data and methods loaded for the chemical; performs the actual calculations of thermal conductivity of the gas phase of the chemical.

SurfaceTension [object] Instance of *thermo.interface.SurfaceTension*, with data and methods loaded for the chemical; performs the actual calculations of surface tension of the chemical.

Permittivity [object] Instance of *thermo.permittivity.PermittivityLiquid*, with data and methods loaded for the chemical; performs the actual calculations of permittivity of the chemical.

Psat_298 [float] Vapor pressure of the chemical at 298.15 K, [Pa]

phase_STP [str] Phase of the chemical at 298.15 K and 101325 Pa; one of 's', 'l', 'g', or 'l/g'.

Vml_Tb [float] Molar volume of liquid phase at the normal boiling point [m^3/mol]

Vml_Tm [float] Molar volume of liquid phase at the melting point [m^3/mol]

Vml_STP [float] Molar volume of liquid phase at 298.15 K and 101325 Pa [m^3/mol]

rhoml_STP [float] Molar density of liquid phase at 298.15 K and 101325 Pa [mol/m^3]

Vmg_STP [float] Molar volume of gas phase at 298.15 K and 101325 Pa according to the ideal gas law, [m^3/mol]

Vms_Tm [float] Molar volume of solid phase at the melting point [m^3/mol]

rhos_Tm [float] Mass density of solid phase at the melting point [kg/m^3]

Hvap_Tbm [float] Molar enthalpy of vaporization at the normal boiling point [J/mol]

Hvap_Tb [float] Mass enthalpy of vaporization at the normal boiling point [J/kg]

Hvapm_298 [float] Molar enthalpy of vaporization at 298.15 K [J/mol]

Hvap_298 [float] Mass enthalpy of vaporization at 298.15 K [J/kg]

`alpha` Thermal diffusivity of the chemical at its current temperature, pressure, and phase in units of [m^2/s].

`alphag` Thermal diffusivity of the gas phase of the chemical at its current temperature and pressure, in units of [m^2/s].

`alphal` Thermal diffusivity of the liquid phase of the chemical at its current temperature and pressure, in units of [m^2/s].

`API` API gravity of the liquid phase of the chemical, [degrees].

`aromatic_rings` Number of aromatic rings in a chemical, computed with RDKit from a chemical's SMILES.

`atom_fractions` Dictionary of atom:fractional occurence of the elements in a chemical.

`Bvirial` Second virial coefficient of the gas phase of the chemical at its current temperature and pressure, in units of [mol/m^3].

`charge` Charge of a chemical, computed with RDKit from a chemical's SMILES.

`Cp` Mass heat capacity of the chemical at its current phase and temperature, in units of [J/kg/K].

`Cpg` Gas-phase heat capacity of the chemical at its current temperature, in units of [J/kg/K].

`Cpgm` Gas-phase ideal gas heat capacity of the chemical at its current temperature, in units of [J/mol/K].

`Cpl` Liquid-phase heat capacity of the chemical at its current temperature, in units of [J/kg/K].

`Cplm` Liquid-phase heat capacity of the chemical at its current temperature, in units of [J/mol/K].

`Cpm` Molar heat capacity of the chemical at its current phase and temperature, in units of [J/mol/K].

`Cps` Solid-phase heat capacity of the chemical at its current temperature, in units of [J/kg/K].

`Cpsm` Solid-phase heat capacity of the chemical at its current temperature, in units of [J/mol/K].

`Cvg` Gas-phase ideal-gas contant-volume heat capacity of the chemical at its current temperature, in units of [J/kg/K].

`Cvgm` Gas-phase ideal-gas contant-volume heat capacity of the chemical at its current temperature, in units of [J/mol/K].

`eos` Equation of state object held by the chemical; used to calculate excess thermodynamic quantities, and also provides a vapor pressure curve, enthalpy of vaporization curve, fugacity, thermodynamic partial derivatives, and more; see `thermo.eos` for a full listing.

`Hill` Hill formula of a compound.

`Hvap` Enthalpy of vaporization of the chemical at its current temperature, in units of [J/kg].

Hvapm Enthalpy of vaporization of the chemical at its current temperature, in units of [J/mol].

isentropic_exponent Gas-phase ideal-gas isentropic exponent of the chemical at its current temperature, [dimensionless].

isobaric_expansion Isobaric (constant-pressure) expansion of the chemical at its current phase and temperature, in units of [1/K].

isobaric_expansion_g Isobaric (constant-pressure) expansion of the gas phase of the chemical at its current temperature and pressure, in units of [1/K].

isobaric_expansion_l Isobaric (constant-pressure) expansion of the liquid phase of the chemical at its current temperature and pressure, in units of [1/K].

JT Joule Thomson coefficient of the chemical at its current phase and temperature, in units of [K/Pa].

JTg Joule Thomson coefficient of the chemical in the gas phase at its current temperature and pressure, in units of [K/Pa].

JTl Joule Thomson coefficient of the chemical in the liquid phase at its current temperature and pressure, in units of [K/Pa].

k Thermal conductivity of the chemical at its current phase, temperature, and pressure in units of [W/m/K].

kg Thermal conductivity of the chemical in the gas phase at its current temperature and pressure, in units of [W/m/K].

kl Thermal conductivity of the chemical in the liquid phase at its current temperature and pressure, in units of [W/m/K].

mass_fractions Dictionary of atom:mass-weighted fractional occurence of elements.

mu Viscosity of the chemical at its current phase, temperature, and pressure in units of [Pa*s].

mug Viscosity of the chemical in the gas phase at its current temperature and pressure, in units of [Pa*s].

mul Viscosity of the chemical in the liquid phase at its current temperature and pressure, in units of [Pa*s].

nu Kinematic viscosity of the the chemical at its current temperature, pressure, and phase in units of [m^2/s].

nug Kinematic viscosity of the gas phase of the chemical at its current temperature and pressure, in units of [m^2/s].

nul Kinematic viscosity of the liquid phase of the chemical at its current temperature and pressure, in units of [m^2/s].

Parachor Parachor of the chemical at its current temperature and pressure, in units of [N^0.25*m^2.75/mol].

permittivity Relative permittivity (dielectric constant) of the chemical at its current temperature, [dimensionless].

Poynting Poynting correction factor [dimensionless] for use in phase equilibria methods based on activity coefficients or other reference states.

Pr Prandtl number of the chemical at its current temperature, pressure, and phase; [dimensionless].

Prg Prandtl number of the gas phase of the chemical at its current temperature and pressure, [dimensionless].

Prl Prandtl number of the liquid phase of the chemical at its current temperature and pressure, [dimensionless].

Psat Vapor pressure of the chemical at its current temperature, in units of [Pa].

PSRK_groups Dictionary of PSRK subgroup: count groups for the PSRK subgroups, as determined by DDBST's online service.

rdkitmol RDKit object of the chemical, without hydrogen.

rdkitmol_Hs RDKit object of the chemical, with hydrogen.

rho Mass density of the chemical at its current phase and temperature and pressure, in units of [kg/m^3].

rhog Gas-phase mass density of the chemical at its current temperature and pressure, in units of [kg/m^3].

rhogm Molar density of the chemical in the gas phase at the current temperature and pressure, in units of [mol/m^3].

rhol Liquid-phase mass density of the chemical at its current temperature and pressure, in units of [kg/m^3].

rholm Molar density of the chemical in the liquid phase at the current temperature and pressure, in units of [mol/m^3].

rhom Molar density of the chemical at its current phase and temperature and pressure, in units of [mol/m^3].

rhos Solid-phase mass density of the chemical at its current temperature, in units of [kg/m^3].

rhosm Molar density of the chemical in the solid phase at the current temperature and pressure, in units of [mol/m^3].

rings Number of rings in a chemical, computed with RDKit from a chemical's SMILES.

SG Specific gravity of the chemical, [dimensionless].

SGg Specific gravity of the gas phase of the chemical, [dimensionless].

SGl Specific gravity of the liquid phase of the chemical at the specified temperature and pressure, [dimensionless].

SGs Specific gravity of the solid phase of the chemical at the specified temperature and pressure, [dimensionless].

sigma Surface tension of the chemical at its current temperature, in units of [N/m].

solubility_parameter Solubility parameter of the chemical at its current temperature and pressure, in units of [Pa^0.5].

UNIFAC_Dortmund_groups Dictionary of Dortmund UNIFAC subgroup: count groups for the Dortmund UNIFAC subgroups, as determined by DDBST's online service.

UNIFAC_groups Dictionary of UNIFAC subgroup: count groups for the original UNIFAC subgroups, as determined by DDBST's online service.

UNIFAC_R UNIFAC R (normalized Van der Waals volume), dimensionless.

UNIFAC_Q UNIFAC Q (normalized Van der Waals area), dimensionless.

Van_der_Waals_area Unnormalized Van der Waals area, in units of [m^2/mol].

Van_der_Waals_volume Unnormalized Van der Waals volume, in units of [m^3/mol].

Vm Molar volume of the chemical at its current phase and temperature and pressure, in units of [m^3/mol].

Vmg Gas-phase molar volume of the chemical at its current temperature and pressure, in units of [m^3/mol].

Vml Liquid-phase molar volume of the chemical at its current temperature and pressure, in units of [m^3/mol].

Vms Solid-phase molar volume of the chemical at its current temperature, in units of [m^3/mol].

Z Compressibility factor of the chemical at its current phase and temperature and pressure, [dimensionless].

Zg Compressibility factor of the chemical in the gas phase at the current temperature and pressure, [dimensionless].

Zl Compressibility factor of the chemical in the liquid phase at the current temperature and pressure, [dimensionless].

Zs Compressibility factor of the chemical in the solid phase at the current temperature and pressure, [dimensionless].

Methods

Reynolds([V, D])	
draw_2d([width, height, Hs])	Interface for drawing a 2D image of the molecule.
draw_3d([width, height, style, Hs, atom_labels])	Interface for drawing an interactive 3D view of the molecule.

Bond	
Capillary	
Grashof	
Jakob	
Peclet_heat	
Tsat	
Weber	
calc_H	
calc_H_excess	
calc_S	
calc_S_excess	
calculate	
calculate_PH	
calculate_PS	
calculate_TH	
calculate_TS	
set_TP_sources	
set_constant_sources	
set_constants	
set_eos	
set_ref	
set_thermo	

property A

Helmholtz energy of the chemical at its current temperature and pressure, in units of [J/kg].

This property requires that `thermo.chemical.set_thermo` ran successfully to be accurate. It also depends on the molar volume of the chemical at its current conditions.

property API

API gravity of the liquid phase of the chemical, [degrees]. The reference condition is water at 15.6 °C (60 °F) and 1 atm (rho=999.016 kg/m^3, standardized).

Examples

```
>>> Chemical('water').API
9.999752435378895
```

property Am

Helmholtz energy of the chemical at its current temperature and pressure, in units of [J/mol].

This property requires that `thermo.chemical.set_thermo` ran successfully to be accurate. It also depends on the molar volume of the chemical at its current conditions.

Bond(*L=None*)

property Bvirial

Second virial coefficient of the gas phase of the chemical at its current temperature and pressure, in units of [mol/m^3].

This property uses the object-oriented interface *thermo.volume.VolumeGas*, converting its result with `thermo.utils.B_from_Z`.

Examples

```
>>> Chemical('water').Bvirial
-0.0009596286322838357
```

Capillary(*V=None*)

property Cp

Mass heat capacity of the chemical at its current phase and temperature, in units of [J/kg/K].

Utilizes the object oriented interfaces *thermo.heat_capacity.HeatCapacitySolid*, *thermo.heat_capacity.HeatCapacityLiquid*, and *thermo.heat_capacity.HeatCapacityGas* to perform the actual calculation of each property. Note that those interfaces provide output in molar units (J/mol/K).

Examples

```
>>> w = Chemical('water')
>>> w.Cp, w.phase
(4180.597021827336, 'l')
>>> Chemical('palladium').Cp
234.26767209171211
```

property Cpg

Gas-phase heat capacity of the chemical at its current temperature, in units of [J/kg/K]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityGas*; each Chemical instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> w = Chemical('water', T=520)
>>> w.Cpg
1967.6698314620658
```

property Cpgm

Gas-phase ideal gas heat capacity of the chemical at its current temperature, in units of [J/mol/K]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityGas*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water').Cpgm
33.583577868850675
>>> Chemical('water').HeatCapacityGas.T_dependent_property(320)
33.67865044005934
>>> Chemical('water').HeatCapacityGas.T_dependent_property_integral(300, 320)
672.6480417835064
```

property Cpl

Liquid-phase heat capacity of the chemical at its current temperature, in units of [J/kg/K]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityLiquid*; each Chemical instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Chemical('water', T=320).Cpl
4177.518996988284
```

Ideal entropy change of water from 280 K to 340 K, output converted back to mass-based units of J/kg/K.

```
>>> dSm = Chemical('water').HeatCapacityLiquid.T_dependent_property_integral_
↪over_T(280, 340)
>>> property_molar_to_mass(dSm, Chemical('water').MW)
812.1024585274956
```

property Cplm

Liquid-phase heat capacity of the chemical at its current temperature, in units of [J/mol/K]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityLiquid*; each Chemical instance creates one to actually perform the calculations.

Notes

Some methods give heat capacity along the saturation line, some at 1 atm but only up to the normal boiling point, and some give heat capacity at 1 atm up to the normal boiling point and then along the saturation line. Real-liquid heat capacity is pressure dependent, but this interface is not.

Examples

```
>>> Chemical('water').Cplm
75.31462591538556
>>> Chemical('water').HeatCapacityLiquid.T_dependent_property(320)
75.2591744360631
>>> Chemical('water').HeatCapacityLiquid.T_dependent_property_integral(300, 320)
1505.0619005000553
```

property Cpm

Molar heat capacity of the chemical at its current phase and temperature, in units of [J/mol/K].

Utilizes the object oriented interfaces *thermo.heat_capacity.HeatCapacitySolid*, *thermo.heat_capacity.HeatCapacityLiquid*, and *thermo.heat_capacity.HeatCapacityGas* to perform the actual calculation of each property.

Examples

```
>>> Chemical('cubane').Cpm
137.05489206785944
>>> Chemical('ethylbenzene', T=550, P=3E6).Cpm
294.18449553310046
```

property Cps

Solid-phase heat capacity of the chemical at its current temperature, in units of [J/kg/K]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacitySolid*; each Chemical instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Chemical('palladium', T=400).Cps
241.63563239992484
>>> Pd = Chemical('palladium', T=400)
>>> Cpsms = [Pd.HeatCapacitySolid.T_dependent_property(T) for T in np.
↪linspace(300,500, 5)]
>>> [property_molar_to_mass(Cps, Pd.MW) for Cps in Cpsms]
[234.40150347679008, 238.01856793835751, 241.63563239992484, 245.25269686149224,
↪ 248.86976132305958]
```

property Cpsm

Solid-phase heat capacity of the chemical at its current temperature, in units of [J/mol/K]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacitySolid*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('palladium').Cpsm
24.930765664000003
>>> Chemical('palladium').HeatCapacitySolid.T_dependent_property(320)
25.098979200000002
>>> Chemical('palladium').HeatCapacitySolid.all_methods
set(["PERRY151", 'CRCSTD', 'LASTOVKA_S'])
```

property Cvg

Gas-phase ideal-gas contant-volume heat capacity of the chemical at its current temperature, in units of [J/kg/K]. Subtracts R from the ideal-gas heat capacity; does not include pressure-compensation from an equation of state.

Examples

```
>>> w = Chemical('water', T=520)
>>> w.Cvg
1506.1471795798861
```

property Cvgm

Gas-phase ideal-gas contant-volume heat capacity of the chemical at its current temperature, in units of [J/mol/K]. Subtracts R from the ideal-gas heat capacity; does not include pressure-compensation from an equation of state.

Examples

```
>>> w = Chemical('water', T=520)
>>> w.Cvgm
27.13366316134193
```

Grashof(*Tw=None*, *L=None*)

property Hill

Hill formula of a compound. For a description of the Hill system, see chemicals.elements.atoms_to_Hill.

Examples

```
>>> Chemical('furfuryl alcohol').Hill
'C5H6O2'
```

property Hvap

Enthalpy of vaporization of the chemical at its current temperature, in units of [J/kg].

This property uses the object-oriented interface *thermo.phase_change.EnthalpyVaporization*, but converts its results from molar to mass units.

Examples

```
>>> Chemical('water', T=320).Hvap
2389540.219347256
```

property Hvapm

Enthalpy of vaporization of the chemical at its current temperature, in units of [J/mol]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.phase_change.EnthalpyVaporization*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320).Hvapm
43048.23612280223
>>> Chemical('water').EnthalpyVaporization.T_dependent_property(320)
43048.23612280223
>>> Chemical('water').EnthalpyVaporization.all_methods
set(['VDI_PPDS', 'MORGAN_KOBAYASHI', 'VETERE', 'VELASCO', 'LIU', 'COOLPROP',
↪'CRC_HVAP_298', 'CLAPEYRON', 'SIVARAMAN_MAGEE_KOBAYASHI', 'ALIBAKHSHI',
↪'DIPPR_PERRY_8E', 'RIEDEL', 'CHEN', 'PITZER', 'CRC_HVAP_TB'])
```

property JT

Joule Thomson coefficient of the chemical at its current phase and temperature, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Examples

```
>>> Chemical('water').JT
-2.2150394958666407e-07
```

property JTg

Joule Thomson coefficient of the chemical in the gas phase at its current temperature and pressure, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Utilizes the temperature-derivative method of *thermo.volume.VolumeGas* and the temperature-dependent heat capacity method *thermo.heat_capacity.HeatCapacityGas* to obtain the properties required for the actual calculation.

Examples

```
>>> Chemical('dodecane', T=400, P=1000).JTg
5.4089897835384913e-05
```

property JTl

Joule Thomson coefficient of the chemical in the liquid phase at its current temperature and pressure, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Utilizes the temperature-derivative method of *thermo.volume.VolumeLiquid* and the temperature-dependent heat capacity method *thermo.heat_capacity.HeatCapacityLiquid* to obtain the properties required for the actual calculation.

Examples

```
>>> Chemical('dodecane', T=400).JTl
-3.0827160465192742e-07
```

Jakob(*Tw=None*)

property PSRK_groups

Dictionary of PSRK subgroup: count groups for the PSRK subgroups, as determined by DDBST's online service.

Examples

```
>>> Chemical('Cumene').PSRK_groups
{1: 2, 9: 5, 13: 1}
```

property Parachor

Parachor of the chemical at its current temperature and pressure, in units of [N^0.25*m^2.75/mol].

$$P = \frac{\sigma^{0.25} MW}{\rho_L - \rho_V}$$

Calculated based on surface tension, density of the liquid phase, and molecular weight. For uses of this property, see `thermo.utils.Parachor`.

The gas density is calculated using the ideal-gas law.

Examples

```
>>> Chemical('octane').Parachor
6.2e-05
```

Peclet_heat(*V=None*, *D=None*)

property Poynting

Poynting correction factor [dimensionless] for use in phase equilibria methods based on activity coefficients or other reference states. Performs the shortcut calculation assuming molar volume is independent of pressure.

$$\text{Poy} = \exp\left[\frac{V_l(P - P^{sat})}{RT}\right]$$

The full calculation normally returns values very close to the approximate ones. This property is defined in terms of pure components only.

Notes

The full equation shown below can be used as follows:

$$\text{Poy} = \exp\left[\frac{\int_{P_i^{sat}}^{P} V_i^l dP}{RT}\right]$$

```
>>> from scipy.integrate import quad
>>> c = Chemical('pentane', T=300, P=1E7)
>>> exp(quad(lambda P : c.VolumeLiquid(c.T, P), c.Psat, c.P)[0]/R/c.T)
1.5821826990975127
```

Examples

```
>>> Chemical('pentane', T=300, P=1E7).Poynting
1.5743051250679803
```

property Pr

Prandtl number of the chemical at its current temperature, pressure, and phase; [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Examples

```
>>> Chemical('acetone').Pr
4.183039103542709
```

property Prg

Prandtl number of the gas phase of the chemical at its current temperature and pressure, [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Utilizes the temperature and pressure dependent object oriented interfaces *thermo.viscosity.ViscosityGas*, *thermo.thermal_conductivity.ThermalConductivityGas*, and *thermo.heat_capacity.HeatCapacityGas* to calculate the actual properties.

Examples

```
>>> Chemical('NH3').Prg
0.847263731933008
```

property Prl

Prandtl number of the liquid phase of the chemical at its current temperature and pressure, [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Utilizes the temperature and pressure dependent object oriented interfaces *thermo.viscosity.ViscosityLiquid*, *thermo.thermal_conductivity.ThermalConductivityLiquid*, and *thermo.heat_capacity.HeatCapacityLiquid* to calculate the actual properties.

Examples

```
>>> Chemical('nitrogen', T=70).Prl
2.7828214501488886
```

property Psat

Vapor pressure of the chemical at its current temperature, in units of [Pa]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.vapor_pressure.VaporPressure*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320).Psat
10533.614271198725
>>> Chemical('water').VaporPressure.T_dependent_property(320)
10533.614271198725
>>> Chemical('water').VaporPressure.all_methods
set(['VDI_PPDS', 'BOILING_CRITICAL', 'WAGNER_MCGARRY', 'AMBROSE_WALTON',
↪'COOLPROP', 'LEE_KESLER_PSAT', 'EOS', 'ANTOINE_POLING', 'SANJARI', 'DIPPR_
↪PERRY_8E', 'Edalat'])
```

property R_specific

Specific gas constant, in units of [J/kg/K].

Examples

```
>>> Chemical('water').R_specific
461.52265188218
```

Reynolds(*V=None*, *D=None*)

property SG

Specific gravity of the chemical, [dimensionless].

For gas-phase conditions, this is calculated at 15.6 °C (60 °F) and 1 atm for the chemical and the reference fluid, air. For liquid and solid phase conditions, this is calculated based on a reference fluid of water at 4°C at 1 atm, but the with the liquid or solid chemical's density at the currently specified conditions.

Examples

```
>>> Chemical('MTBE').SG
0.7428160596603596
```

property SGg

Specific gravity of the gas phase of the chemical, [dimensionless]. The reference condition is air at 15.6 °C (60 °F) and 1 atm (rho=1.223 kg/m^3). The definition for gases uses the compressibility factor of the reference gas and the chemical both at the reference conditions, not the conditions of the chemical.

Examples

```
>>> Chemical('argon').SGg
1.3795835970877504
```

property SGl

Specific gravity of the liquid phase of the chemical at the specified temperature and pressure, [dimensionless]. The reference condition is water at 4 °C and 1 atm (rho=999.017 kg/m^3). For liquids, SG is defined that the reference chemical's T and P are fixed, but the chemical itself varies with the specified T and P.

Examples

```
>>> Chemical('water', T=365).SGl
0.9650065522428539
```

property SGs

Specific gravity of the solid phase of the chemical at the specified temperature and pressure, [dimensionless]. The reference condition is water at 4 °C and 1 atm (rho=999.017 kg/m^3). The SG varries with temperature and pressure but only very slightly.

Examples

```
>>> Chemical('iron').SGs
7.87774317235069
```

Tsat(*P*)

property U

Internal energy of the chemical at its current temperature and pressure, in units of [J/kg].

This property requires that `thermo.chemical.set_thermo` ran successfully to be accurate. It also depends on the molar volume of the chemical at its current conditions.

property UNIFAC_Dortmund_groups

Dictionary of Dortmund UNIFAC subgroup: count groups for the Dortmund UNIFAC subgroups, as determined by DDBST's online service.

Examples

```
>>> Chemical('Cumene').UNIFAC_Dortmund_groups
{1: 2, 9: 5, 13: 1}
```

property UNIFAC_Q

UNIFAC Q (normalized Van der Waals area), dimensionless. Used in the UNIFAC model.

Examples

```
>>> Chemical('decane').UNIFAC_Q
6.016
```

property UNIFAC_R

UNIFAC R (normalized Van der Waals volume), dimensionless. Used in the UNIFAC model.

Examples

```
>>> Chemical('benzene').UNIFAC_R
3.1878
```

property UNIFAC_groups

Dictionary of UNIFAC subgroup: count groups for the original UNIFAC subgroups, as determined by DDBST's online service.

Examples

```
>>> Chemical('Cumene').UNIFAC_groups
{1: 2, 9: 5, 13: 1}
```

property Um

Internal energy of the chemical at its current temperature and pressure, in units of [J/mol].

This property requires that `thermo.chemical.set_thermo` ran successfully to be accurate. It also depends on the molar volume of the chemical at its current conditions.

property Van_der_Waals_area

Unnormalized Van der Waals area, in units of [m^2/mol].

Examples

```
>>> Chemical('hexane').Van_der_Waals_area
964000.0
```

property Van_der_Waals_volume

Unnormalized Van der Waals volume, in units of [m^3/mol].

Examples

```
>>> Chemical('hexane').Van_der_Waals_volume
6.8261966e-05
```

property Vm

Molar volume of the chemical at its current phase and temperature and pressure, in units of [m^3/mol].

Utilizes the object oriented interfaces *thermo.volume.VolumeSolid*, *thermo.volume.VolumeLiquid*, and *thermo.volume.VolumeGas* to perform the actual calculation of each property.

Examples

```
>>> Chemical('ethylbenzene', T=550, P=3E6).Vm
0.00017758024401627633
```

property Vmg

Gas-phase molar volume of the chemical at its current temperature and pressure, in units of [m^3/mol]. For calculation of this property at other temperatures or pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeGas*; each Chemical instance creates one to actually perform the calculations.

Examples

Estimate the molar volume of the core of the sun, at 15 million K and 26.5 PetaPascals, assuming pure helium (actually 68% helium):

```
>>> Chemical('helium', T=15E6, P=26.5E15).Vmg
4.805464238181197e-07
```

property Vmg_ideal

Gas-phase molar volume of the chemical at its current temperature and pressure calculated with the ideal-gas law, in units of [m^3/mol].

Examples

```
>>> Chemical('helium', T=300.0, P=1e5).Vmg_ideal
0.0249433878544
```

property Vml

Liquid-phase molar volume of the chemical at its current temperature and pressure, in units of [m^3/mol]. For calculation of this property at other temperatures or pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeLiquid*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('cyclobutane', T=225).Vml
7.42395423425395e-05
```

property Vms

Solid-phase molar volume of the chemical at its current temperature, in units of [m^3/mol]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeSolid*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('iron').Vms
7.09593392630242e-06
```

Weber(*V=None*, *D=None*)

property Z

Compressibility factor of the chemical at its current phase and temperature and pressure, [dimensionless].

Examples

```
>>> Chemical('MTBE', T=900, P=1E-2).Z
0.9999999999079768
```

property Zg

Compressibility factor of the chemical in the gas phase at the current temperature and pressure, [dimensionless].

Utilizes the object oriented interface and *thermo.volume.VolumeGas* to perform the actual calculation of molar volume.

Examples

```
>>> Chemical('sulfur hexafluoride', T=700, P=1E9).Zg
11.140084184207813
```

property Zl

Compressibility factor of the chemical in the liquid phase at the current temperature and pressure, [dimensionless].

Utilizes the object oriented interface and *thermo.volume.VolumeLiquid* to perform the actual calculation of molar volume.

Examples

```
>>> Chemical('water').Zl
0.0007385375470263454
```

property Zs

Compressibility factor of the chemical in the solid phase at the current temperature and pressure, [dimensionless].

Utilizes the object oriented interface and *thermo.volume.VolumeSolid* to perform the actual calculation of molar volume.

Examples

```
>>> Chemical('palladium').Z
0.00036248477437931853
```

property absolute_permittivity

Absolute permittivity of the chemical at its current temperature, in units of [farad/meter]. Those units are equivalent to ampere^2*second^4/kg/m^3.

Examples

```
>>> Chemical('water', T=293.15).absolute_permittivity
7.096684821859018e-10
```

property alpha

Thermal diffusivity of the chemical at its current temperature, pressure, and phase in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Examples

```
>>> Chemical('furfural').alpha
8.696537158635412e-08
```

property alphag

Thermal diffusivity of the gas phase of the chemical at its current temperature and pressure, in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Utilizes the temperature and pressure dependent object oriented interfaces *thermo.volume.VolumeGas*, *thermo.thermal_conductivity.ThermalConductivityGas*, and *thermo.heat_capacity.HeatCapacityGas* to calculate the actual properties.

Examples

```
>>> Chemical('ammonia').alphag
1.6931865425158556e-05
```

property alphal

Thermal diffusivity of the liquid phase of the chemical at its current temperature and pressure, in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Utilizes the temperature and pressure dependent object oriented interfaces *thermo.volume.VolumeLiquid*, *thermo.thermal_conductivity.ThermalConductivityLiquid*, and *thermo.heat_capacity.HeatCapacityLiquid* to calculate the actual properties.

Examples

```
>>> Chemical('nitrogen', T=70).alphal
9.444949636299626e-08
```

property aromatic_rings

Number of aromatic rings in a chemical, computed with RDKit from a chemical's SMILES. If RDKit is not available, holds None.

Examples

```
>>> Chemical('Paclitaxel').aromatic_rings
3
```

property atom_fractions

Dictionary of atom:fractional occurence of the elements in a chemical. Useful when performing element balances. For mass-fraction occurences, see *mass_fractions*.

Examples

```
>>> Chemical('Ammonium aluminium sulfate').atom_fractions
{'H': 0.25, 'S': 0.125, 'Al': 0.0625, 'O': 0.5, 'N': 0.0625}
```

calc_H(*T*, *P*)

calc_H_excess(*T*, *P*)

calc_S(*T*, *P*)

calc_S_excess(*T*, *P*)

calculate(*T=None*, *P=None*)

calculate_PH(*P*, *H*)

calculate_PS(*P*, *S*)

calculate_TH(*T*, *H*)

calculate_TS(*T*, *S*)

property charge

Charge of a chemical, computed with RDKit from a chemical's SMILES. If RDKit is not available, holds None.

Examples

```
>>> Chemical('sodium ion').charge
1
```

draw_2d(*width=300*, *height=300*, *Hs=False*)

Interface for drawing a 2D image of the molecule. Requires an HTML5 browser, and the libraries RDKit and IPython. An exception is raised if either of these libraries is absent.

Parameters

width [int] Number of pixels wide for the view

height [int] Number of pixels tall for the view

Hs [bool] Whether or not to show hydrogen

Examples

```
>>> Chemical('decane').draw_2d()
<PIL.Image.Image image mode=RGBA size=300x300 at 0x...>
```

draw_3d(*width=300*, *height=500*, *style='stick'*, *Hs=True*, *atom_labels=True*)

Interface for drawing an interactive 3D view of the molecule. Requires an HTML5 browser, and the libraries RDKit, pymol3D, and IPython. An exception is raised if all three of these libraries are not installed.

Parameters

width [int] Number of pixels wide for the view, [pixels]

height [int] Number of pixels tall for the view, [pixels]

style [str] One of 'stick', 'line', 'cross', or 'sphere', [-]

Hs [bool] Whether or not to show hydrogen, [-]

atom_labels [bool] Whether or not to label the atoms, [-]

Examples

```
>>> Chemical('cubane').draw_3d()
<IPython.core.display.HTML object>
```

property economic_status

Dictionary of economic status indicators for the chemical.

Examples

```
>>> Chemical('benzene').economic_status
["US public: {'Manufactured': 6165232.1, 'Imported': 463146.474, 'Exported':␣
→271908.252}",
 u'1,000,000 - 10,000,000 tonnes per annum',
 u'Intermediate Use Only',
 'OECD HPV Chemicals']
```

property eos

Equation of state object held by the chemical; used to calculate excess thermodynamic quantities, and also provides a vapor pressure curve, enthalpy of vaporization curve, fugacity, thermodynamic partial derivatives, and more; see *thermo.eos* for a full listing.

Examples

```
>>> Chemical('methane').eos.V_g
0.024410195021818258
```

property isentropic_exponent

Gas-phase ideal-gas isentropic exponent of the chemical at its current temperature, [dimensionless]. Does not include pressure-compensation from an equation of state.

Examples

```
>>> Chemical('hydrogen').isentropic_exponent
1.405237786321222
```

property isobaric_expansion

Isobaric (constant-pressure) expansion of the chemical at its current phase and temperature, in units of [1/K].

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Examples

Radical change in value just above and below the critical temperature of water:

```
>>> Chemical('water', T=647.1, P=22048320.0).isobaric_expansion
0.34074205839222449
```

```
>>> Chemical('water', T=647.2, P=22048320.0).isobaric_expansion
0.18143324022215077
```

property isobaric_expansion_g

Isobaric (constant-pressure) expansion of the gas phase of the chemical at its current temperature and pressure, in units of [1/K].

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Utilizes the temperature-derivative method of `thermo.VolumeGas` to perform the actual calculation. The derivatives are all numerical.

Examples

```
>>> Chemical('Hexachlorobenzene', T=900).isobaric_expansion_g
0.001151869741981048
```

property isobaric_expansion_l

Isobaric (constant-pressure) expansion of the liquid phase of the chemical at its current temperature and pressure, in units of [1/K].

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Utilizes the temperature-derivative method of *thermo.volume.VolumeLiquid* to perform the actual calculation. The derivatives are all numerical.

Examples

```
>>> Chemical('dodecane', T=400).isobaric_expansion_l
0.0011617555762469477
```

property k

Thermal conductivity of the chemical at its current phase, temperature, and pressure in units of [W/m/K].

Utilizes the object oriented interfaces *thermo.thermal_conductivity.ThermalConductivityLiquid* and *thermo.thermal_conductivity.ThermalConductivityGas* to perform the actual calculation of each property.

Examples

```
>>> Chemical('ethanol', T=300).kl
0.16313594741877802
>>> Chemical('ethanol', T=400).kg
0.026019924109310026
```

property kg

Thermal conductivity of the chemical in the gas phase at its current temperature and pressure, in units of [W/m/K].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.thermal_conductivity.ThermalConductivityGas*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320).kg
0.021273128263091207
```

property kl

Thermal conductivity of the chemical in the liquid phase at its current temperature and pressure, in units of [W/m/K].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.thermal_conductivity.ThermalConductivityLiquid*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320).kl
0.6369957248212118
```

property legal_status

Dictionary of legal status indicators for the chemical.

Examples

```
>>> Chemical('benzene').legal_status
{'DSL': 'LISTED', 'EINECS': 'LISTED', 'NLP': 'UNLISTED', 'SPIN': 'LISTED', 'TSCA
↪': 'LISTED'}
```

property mass_fractions

Dictionary of atom:mass-weighted fractional occurence of elements. Useful when performing mass balances. For atom-fraction occurences, see *atom_fractions*.

Examples

```
>>> Chemical('water').mass_fractions
{'H': 0.11189834407236524, 'O': 0.8881016559276347}
```

property mu

Viscosity of the chemical at its current phase, temperature, and pressure in units of [Pa*s].

Utilizes the object oriented interfaces *thermo.viscosity.ViscosityLiquid* and *thermo.viscosity.ViscosityGas* to perform the actual calculation of each property.

Examples

```
>>> Chemical('ethanol', T=300).mu
0.001044526538460911
>>> Chemical('ethanol', T=400).mu
1.1853097849748217e-05
```

property mug

Viscosity of the chemical in the gas phase at its current temperature and pressure, in units of [Pa*s].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.viscosity.ViscosityGas*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320, P=100).mug
1.0431450856297212e-05
```

property mul

Viscosity of the chemical in the liquid phase at its current temperature and pressure, in units of [Pa*s].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.viscosity.ViscosityLiquid*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320).mul
0.0005767262693751547
```

property nu

Kinematic viscosity of the the chemical at its current temperature, pressure, and phase in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> Chemical('argon').nu
1.3846930410865003e-05
```

property nug

Kinematic viscosity of the gas phase of the chemical at its current temperature and pressure, in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Utilizes the temperature and pressure dependent object oriented interfaces *thermo.volume.VolumeGas*, *thermo.viscosity.ViscosityGas* to calculate the actual properties.

Examples

```
>>> Chemical('methane', T=115).nug
2.5056924327995865e-06
```

property nul

Kinematic viscosity of the liquid phase of the chemical at its current temperature and pressure, in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Utilizes the temperature and pressure dependent object oriented interfaces *thermo.volume.VolumeLiquid*, *thermo.viscosity.ViscosityLiquid* to calculate the actual properties.

Examples

```
>>> Chemical('methane', T=110).nul
2.858088468937331e-07
```

property permittivity

Relative permittivity (dielectric constant) of the chemical at its current temperature, [dimensionless].

For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.permittivity.PermittivityLiquid*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('toluene', T=250).permittivity
2.49775625
```

property rdkitmol

RDKit object of the chemical, without hydrogen. If RDKit is not available, holds None.

For examples of what can be done with RDKit, see their website.

property rdkitmol_Hs

RDKit object of the chemical, with hydrogen. If RDKit is not available, holds None.

For examples of what can be done with RDKit, see their website.

property rho

Mass density of the chemical at its current phase and temperature and pressure, in units of [kg/m^3].

Utilizes the object oriented interfaces *thermo.volume.VolumeSolid*, *thermo.volume.VolumeLiquid*, and *thermo.volume.VolumeGas* to perform the actual calculation of each property. Note that those interfaces provide output in units of m^3/mol.

Examples

```
>>> Chemical('decane', T=550, P=2E6).rho
498.67008448640604
```

property rhog

Gas-phase mass density of the chemical at its current temperature and pressure, in units of [kg/m^3]. For calculation of this property at other temperatures or pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeGas*; each Chemical instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

Estimate the density of the core of the sun, at 15 million K and 26.5 PetaPascals, assuming pure helium (actually 68% helium):

```
>>> Chemical('helium', T=15E6, P=26.5E15).rhog
8329.27226509739
```

Compared to a result on Wikipedia of 150000 kg/m^3, the fundamental equation of state performs poorly.

```
>>> He = Chemical('helium', T=15E6, P=26.5E15)
>>> He.VolumeGas.method_P = 'IDEAL'
>>> He.rhog
850477.8
```

The ideal-gas law performs somewhat better, but vastly overshoots the density prediction.

property rhogm

Molar density of the chemical in the gas phase at the current temperature and pressure, in units of [mol/m^3].

Utilizes the object oriented interface and *thermo.volume.VolumeGas* to perform the actual calculation of molar volume.

Examples

```
>>> Chemical('tungsten hexafluoride').rhogm
42.01349946063116
```

property rhol

Liquid-phase mass density of the chemical at its current temperature and pressure, in units of [kg/m^3]. For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeLiquid*; each Chemical instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Chemical('o-xylene', T=297).rhol
876.9946785618097
```

property rholm

Molar density of the chemical in the liquid phase at the current temperature and pressure, in units of [mol/m^3].

Utilizes the object oriented interface and *thermo.volume.VolumeLiquid* to perform the actual calculation of molar volume.

Examples

```
>>> Chemical('nitrogen', T=70).rholm
29937.20179186975
```

property rhom

Molar density of the chemical at its current phase and temperature and pressure, in units of [mol/m^3].

Utilizes the object oriented interfaces *thermo.volume.VolumeSolid*, *thermo.volume.VolumeLiquid*, and *thermo.volume.VolumeGas* to perform the actual calculation of each property. Note that those interfaces provide output in units of m^3/mol.

Examples

```
>>> Chemical('1-hexanol').rhom
7983.414573003429
```

property rhos

Solid-phase mass density of the chemical at its current temperature, in units of [kg/m^3]. For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeSolid*; each Chemical instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Chemical('iron').rhos
7869.999999999994
```

property rhosm

Molar density of the chemical in the solid phase at the current temperature and pressure, in units of [mol/m^3].

Utilizes the object oriented interface and *thermo.volume.VolumeSolid* to perform the actual calculation of molar volume.

Examples

```
>>> Chemical('palladium').rhosm
112760.75925577903
```

property rings

Number of rings in a chemical, computed with RDKit from a chemical's SMILES. If RDKit is not available, holds None.

Examples

```
>>> Chemical('Paclitaxel').rings
7
```

set_TP_sources()

set_constant_sources()

set_constants()

set_eos(*T*, *P*, *eos=<class 'thermo.eos.PR'>*)

set_ref(*T_ref=298.15*, *P_ref=101325*, *phase_ref='calc'*, *H_ref=0*, *S_ref=0*)

set_thermo()

property sigma

Surface tension of the chemical at its current temperature, in units of [N/m].

For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.interface.SurfaceTension*; each Chemical instance creates one to actually perform the calculations.

Examples

```
>>> Chemical('water', T=320).sigma
0.06855002575793023
>>> Chemical('water', T=320).SurfaceTension.solve_property(0.05)
416.8307110842183
```

property solubility_parameter

Solubility parameter of the chemical at its current temperature and pressure, in units of [Pa^0.5].

$$\delta = \sqrt{\frac{\Delta H_{vap} - RT}{V_m}}$$

Calculated based on enthalpy of vaporization and molar volume. Normally calculated at STP. For uses of this property, see `thermo.solubility.solubility_parameter`.

Examples

```
>>> Chemical('NH3').solubility_parameter
24766.329043856073
```

7.4 Chemical Constants and Correlations (thermo.chemical_package)

This module contains classes for storing data and objects which are necessary for doing thermodynamic calculations. The intention for these classes is to serve as an in-memory storage layer between the disk and methods which do full thermodynamic calculations.

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

7.4.1 Chemical Constants Class

class `thermo.chemical_package.ChemicalConstantsPackage`(*CASs=None, names=None, MWs=None, Tms=None, Tbs=None, Tcs=None, Pcs=None, Vcs=None, omegas=None, Zcs=None, rhocs=None, rhocs_mass=None, Hfus_Tms=None, Hfus_Tms_mass=None, Hvap_Tbs=None, Hvap_Tbs_mass=None, Vml_STPs=None, rhol_STPs=None, rhol_STPs_mass=None, Vml_60Fs=None, rhol_60Fs=None, rhol_60Fs_mass=None, Vmg_STPs=None, rhog_STPs=None, rhog_STPs_mass=None, Hfgs=None, Hfgs_mass=None, Gfgs=None, Gfgs_mass=None, Sfgs=None, Sfgs_mass=None, S0gs=None, S0gs_mass=None, Hf_STPs=None, Hf_STPs_mass=None, Tts=None, Pts=None, Hsub_Tts=None, Hsub_Tts_mass=None, Hcs=None, Hcs_mass=None, Hcs_lower=None, Hcs_lower_mass=None, Tflashs=None, Tautoignitions=None, LFLs=None, UFLs=None, TWAs=None, STELs=None, Ceilings=None, Skins=None, Carcinogens=None, legal_statuses=None, economic_statuses=None, GWPs=None, ODPs=None, logPs=None, Psat_298s=None, Hvap_298s=None, Hvap_298s_mass=None, Vml_Tms=None, rhos_Tms=None, Vms_Tms=None, rhos_Tms_mass=None, sigma_STPs=None, sigma_Tbs=None, sigma_Tms=None, RIs=None, RI_Ts=None, conductivities=None, conductivity_Ts=None, charges=None, dipoles=None, Stockmayers=None, molecular_diameters=None, Van_der_Waals_volumes=None, Van_der_Waals_areas=None, Parachors=None, StielPolars=None, atomss=None, atom_fractions=None, similarity_variables=None, phase_STPs=None, solubility_parameters=None, PubChems=None, formulas=None, smiless=None, InChIs=None, InChI_Keys=None, UNIFAC_groups=None, UNIFAC_Dortmund_groups=None, PSRK_groups=None, UNIFAC_Rs=None, UNIFAC_Qs=None*)

Class for storing efficiently chemical constants for a group of components. This is intended as a base object from which a set of thermodynamic methods can access miscellaneous for purposes such as phase identification or initialization.

Parameters

N [int] Number of components in the package, [-].

cmps [range] Iterator over all components, [-].

rhol_60Fs [list[float]] Liquid molar densities for each component at 60 °F, [mol/m^3].

atom_fractions [list[dict]] Breakdown of each component into its elemental fractions, as a dict, [-].

atomss [list[dict]] Breakdown of each component into its elements and their counts, as a dict, [-].

Carcinogens [list[dict]] Status of each component in cancer causing registries, [-].

CASs [list[str]] CAS registration numbers for each component, [-].

Ceilings [list[tuple[(float, str)]]] Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

charges [list[float]] Charge number (valence) for each component, [-].

conductivities [list[float]] Electrical conductivities for each component, [S/m].

conductivity_Ts [list[float]] Temperatures at which the electrical conductivities for each component were measured, [K].

dipoles [list[float]] Dipole moments for each component, [debye].

economic_statuses [list[dict]] Status of each component in in relation to import and export from various regions, [-].

formulas [list[str]] Formulas of each component, [-].

Gfgs [list[float]] Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

Gfgs_mass [list[float]] Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

GWPs [list[float]] Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

Hcs [list[float]] Higher standard molar heats of combustion for each component, [J/mol].

Hcs_mass [list[float]] Higher standard heats of combustion for each component, [J/kg].

Hcs_lower [list[float]] Lower standard molar heats of combustion for each component, [J/mol].

Hcs_lower_mass [list[float]] Lower standard heats of combustion for each component, [J/kg].

Hfgs [list[float]] Ideal gas standard molar enthalpies of formation for each component, [J/mol].

Hfgs_mass [list[float]] Ideal gas standard enthalpies of formation for each component, [J/kg].

Hfus_Tms [list[float]] Molar heats of fusion for each component at their respective melting points, [J/mol].

Hfus_Tms_mass [list[float]] Heats of fusion for each component at their respective melting points, [J/kg].

Hsub_Tts [list[float]] Heats of sublimation for each component at their respective triple points, [J/mol].

Hsub_Tts_mass [list[float]] Heats of sublimation for each component at their respective triple points, [J/kg].

Hvap_298s [list[float]] Molar heats of vaporization for each component at 298.15 K, [J/mol].

Hvap_298s_mass [list[float]] Heats of vaporization for each component at 298.15 K, [J/kg].

Hvap_Tbs [list[float]] Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

Hvap_Tbs_mass [list[float]] Heats of vaporization for each component at their respective normal boiling points, [J/kg].

InChI_Keys [list[str]] InChI Keys for each component, [-].

InChIs [list[str]] InChI strings for each component, [-].

legal_statuses [list[dict]] Status of each component in in relation to import and export rules from various regions, [-].

LFLs [list[float]] Lower flammability limits for each component, [-].

logPs [list[float]] Octanol-water partition coefficients for each component, [-].

molecular_diameters [list[float]] Lennard-Jones molecular diameters for each component, [angstrom].

MWs [list[float]] Similatiry variables for each component, [g/mol].

names [list[str]] Names for each component, [-].

ODPs [list[float]] Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

omegas [list[float]] Acentric factors for each component, [-].

Parachors [list[float]] Parachors for each component, [N^0.25*m^2.75/mol].

Pcs [list[float]] Critical pressures for each component, [Pa].

phase_STPs [list[str]] Standard states ('g', 'l', or 's') for each component, [-].

Psat_298s [list[float]] Vapor pressures for each component at 298.15 K, [Pa].

PSRK_groups [list[dict]] PSRK subgroup: count groups for each component, [-].

Pts [list[float]] Triple point pressures for each component, [Pa].

PubChems [list[int]] Pubchem IDs for each component, [-].

rhocs [list[float]] Molar densities at the critical point for each component, [mol/m^3].

rhocs_mass [list[float]] Densities at the critical point for each component, [kg/m^3].

rhol_STPs [list[float]] Molar liquid densities at STP for each component, [mol/m^3].

rhol_STPs_mass [list[float]] Liquid densities at STP for each component, [kg/m^3].

RIs [list[float]] Refractive indexes for each component, [-].

RI_Ts [list[float]] Temperatures at which the refractive indexes were reported for each component, [K].

S0gs [list[float]] Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

S0gs_mass [list[float]] Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

Sfgs [list[float]] Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

Sfgs_mass [list[float]] Ideal gas standard entropies of formation for each component, [J/(kg*K)].

solubility_parameters [list[float]] Solubility parameters for each component at 298.15 K, [Pa^0.5].

similarity_variables [list[float]] Similarity variables for each component, [mol/g].

Skins [list[bool]] Whether each compound can be absorbed through the skin or not, [-].

smiless [list[str]] SMILES identifiers for each component, [-].

STELs [list[tuple[(float, str)]]] Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

StielPolars [list[float]] Stiel polar factors for each component, [-].

Stockmayers [list[float]] Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

Tautoignitions [list[float]] Autoignition temperatures for each component, [K].

Tbs [list[float]] Boiling temperatures for each component, [K].

Tcs [list[float]] Critical temperatures for each component, [K].

Tms [list[float]] Melting temperatures for each component, [K].

Tflashs [list[float]] Flash point temperatures for each component, [K].

Tts [list[float]] Triple point temperatures for each component, [K].

TWAs [list[tuple[(float, str)]]] Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

UFLs [list[float]] Upper flammability limits for each component, [-].

UNIFAC_Dortmund_groups [list[dict]] UNIFAC_Dortmund_group: count groups for each component, [-].

UNIFAC_groups [list[dict]] UNIFAC_group: count groups for each component, [-].

UNIFAC_Rs [list[float]] UNIFAC R parameters for each component, [-].

UNIFAC_Qs [list[float]] UNIFAC Q parameters for each component, [-].

Van_der_Waals_areas [list[float]] Unnormalized Van der Waals areas for each component, [m^2/mol].

Van_der_Waals_volumes [list[float]] Unnormalized Van der Waals volumes for each component, [m^3/mol].

Vcs [list[float]] Critical molar volumes for each component, [m^3/mol].

Vml_STPs [list[float]] Liquid molar volumes for each component at STP, [m^3/mol].

Vml_Tms [list[float]] Liquid molar volumes for each component at their respective melting points, [m^3/mol].

Vms_Tms [list[float]] Solid molar volumes for each component at their respective melting points, [m^3/mol].

Vml_60Fs [list[float]] Liquid molar volumes for each component at 60 °F, [m^3/mol].

rhos_Tms [list[float]] Solid molar densities for each component at their respective melting points, [mol/m^3].

rhol_60Fs_mass [list[float]] Liquid mass densities for each component at 60 °F, [kg/m^3].

rhos_Tms_mass [list[float]] Solid mass densities for each component at their melting point, [kg/m^3].

Zcs [list[float]] Critical compressibilities for each component, [-].

n_atoms [int] Number of total atoms in a collection of 1 molecule of each species, [-].

water_index [int] Index of water in the package, [-].

Vmg_STPs [list[float]] Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

rhog_STPs [list[float]] Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

rhog_STPs_mass [list[float]] Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

sigma_STPs [list[float]] Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

sigma_Tms [list[float]] Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

sigma_Tbs [list[float]] Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

Hf_STPs [list[float]] Standard state molar enthalpies of formation for each component, [J/mol].

Hf_STPs_mass [list[float]] Standard state mass enthalpies of formation for each component, [J/kg].

Notes

All parameters are also attributes.

Examples

Create a package with water and the xylenes, suitable for use with equations of state:

```
>>> ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165], names=[
↪'water', 'o-xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.324, 0.
↪331], Pcs=[22048320.0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14, 630.3, 616.
↪2, 617.0])
ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165], names=['water',
↪'o-xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.324, 0.331],␣
↪Pcs=[22048320.0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14, 630.3, 616.2, 617.
↪0])
```

Methods

as_json()	Method to create a JSON friendly serialization of the chemical constants package which can be stored, and reloaded later.
constants_from_IDs(IDs)	Method to construct a new *ChemicalConstantsPackage* with loaded parameters from the chemicals library, using whatever default methods and values happen to be in that library.
correlations_from_IDs(IDs)	Method to construct a new *PropertyCorrelationsPackage* with loaded parameters from the chemicals library, using whatever default methods and values happen to be in that library.
from_IDs(IDs)	Method to construct a new *ChemicalConstantsPackage* and *PropertyCorrelationsPackage* with loaded parameters from the chemicals library, using whatever default methods and values happen to be in that library.
from_json(json_repr)	Method to create a ChemicalConstantsPackage from a JSON serialization of another ChemicalConstantsPackage.
subset([idxs, properties])	Method to construct a new ChemicalConstantsPackage that removes all components not specified in the *idxs* argument.
with_new_constants(**kwargs)	Method to construct a new ChemicalConstantsPackage that replaces or adds one or more properties for all components.

__add__(*b*)

Method to create a new *ChemicalConstantsPackage* object from two other *ChemicalConstantsPackage* objects.

Returns

new [*ChemicalConstantsPackage*] New object, [-]

Examples

```
>>> a = ChemicalConstantsPackage.constants_from_IDs(IDs=['water', 'hexane'])
>>> b = ChemicalConstantsPackage.constants_from_IDs(IDs=['toluene'])
>>> c = a + b
```

as_json()

Method to create a JSON friendly serialization of the chemical constants package which can be stored, and reloaded later.

Returns

json_repr [dict] Json friendly representation, [-]

Examples

```
>>> import json
>>> constants = ChemicalConstantsPackage(MWs=[18.01528, 106.165], names=['water
↪', 'm-xylene'])
>>> string = json.dumps(constants.as_json())
```

static constants_from_IDs(*IDs*)

Method to construct a new *ChemicalConstantsPackage* with loaded parameters from the chemicals library, using whatever default methods and values happen to be in that library. Expect values to change over time.

Parameters

IDs [list[str]] Identifying strings for each compound; most identifiers are accepted and all inputs are documented in `chemicals.identifiers.search_chemical`, [-]

Returns

constants [ChemicalConstantsPackage] New *ChemicalConstantsPackage* with loaded values, [-]

Notes

Warning: chemicals is a project with a focus on collecting data and correlations from various sources. In no way is it a project to critically evaluate these and provide recommendations. You are strongly encouraged to check values from it and modify them if you want different values. If you believe there is a value which has a typographical error please report it to the chemicals project. If data is missing or not as accuracte as you would like, and you know of a better method or source, new methods and sources can be added to chemicals fairly easily once the data entry is complete. It is not feasible to add individual components, so please submit a complete table of data from the source.

Examples

```
>>> constants = ChemicalConstantsPackage.constants_from_IDs(IDs=['water',
↪'hexane'])
```

static correlations_from_IDs(*IDs*)

Method to construct a new *PropertyCorrelationsPackage* with loaded parameters from the chemicals library, using whatever default methods and values happen to be in that library. Expect values to change over time.

Parameters

IDs [list[str]] Identifying strings for each compound; most identifiers are accepted and all inputs are documented in `chemicals.identifiers.search_chemical`, [-]

Returns

correlations [PropertyCorrelationsPackage] New *PropertyCorrelationsPackage* with loaded values, [-]

Notes

Warning: chemicals is a project with a focus on collecting data and correlations from various sources. In no way is it a project to critically evaluate these and provide recommendations. You are strongly encouraged to check values from it and modify them if you want different values. If you believe there is a value which has a typographical error please report it to the chemicals project. If data is missing or not as accuracte as you would like, and you know of a better method or source, new methods and sources can be added to chemicals fairly easily once the data entry is complete. It is not feasible to add individual components, so please submit a complete table of data from the source.

Examples

```
>>> correlations = ChemicalConstantsPackage.constants_from_IDs(IDs=['ethanol',
→'methanol'])
```

static from_IDs(*IDs*)

Method to construct a new *ChemicalConstantsPackage* and *PropertyCorrelationsPackage* with loaded parameters from the chemicals library, using whatever default methods and values happen to be in that library. Expect values to change over time.

Parameters

IDs [list[str]] Identifying strings for each compound; most identifiers are accepted and all inputs are documented in `chemicals.identifiers.search_chemical`, [-]

Returns

constants [PropertyCorrelationsPackage] New *PropertyCorrelationsPackage* with loaded values, [-]

correlations [PropertyCorrelationsPackage] New *PropertyCorrelationsPackage* with loaded values, [-]

Notes

Warning: chemicals is a project with a focus on collecting data and correlations from various sources. In no way is it a project to critically evaluate these and provide recommendations. You are strongly encouraged to check values from it and modify them if you want different values. If you believe there is a value which has a typographical error please report it to the chemicals project. If data is missing or not as accuracte as you would like, and you know of a better method or source, new methods and sources can be added to chemicals fairly easily once the data entry is complete. It is not feasible to add individual components, so please submit a complete table of data from the source.

Examples

```
>>> constants, correlations = ChemicalConstantsPackage.from_IDs(IDs=['water',
→'decane'])
```

classmethod from_json(*json_repr*)

Method to create a ChemicalConstantsPackage from a JSON serialization of another ChemicalConstantsPackage.

Parameters

json_repr [dict] Json representation, [-]

Returns

constants [ChemicalConstantsPackage] Newly created object from the json serialization, [-]

Notes

It is important that the input be in the same format as that created by *ChemicalConstantsPackage.as_json*.

Examples

```
>>> import json
>>> constants = ChemicalConstantsPackage(MWs=[18.01528, 106.165], names=['water
↪', 'm-xylene'])
>>> string = json.dumps(constants.as_json())
>>> new_constants  = ChemicalConstantsPackage.from_json(json.loads(string))
>>> assert hash(new_constants) == hash(constants)
```

properties = ('atom_fractions', 'atomss', 'Carcinogens', 'CASs', 'Ceilings', 'charges', 'conductivities', 'dipoles', 'economic_statuses', 'formulas', 'Gfgs', 'Gfgs_mass', 'GWPs', 'Hcs', 'Hcs_lower', 'Hcs_lower_mass', 'Hcs_mass', 'Hfgs', 'Hfgs_mass', 'Hfus_Tms', 'Hfus_Tms_mass', 'Hsub_Tts', 'Hsub_Tts_mass', 'Hvap_298s', 'Hvap_298s_mass', 'Hvap_Tbs', 'Hvap_Tbs_mass', 'InChI_Keys', 'InChIs', 'legal_statuses', 'LFLs', 'logPs', 'molecular_diameters', 'MWs', 'names', 'ODPs', 'omegas', 'Parachors', 'Pcs', 'phase_STPs', 'Psat_298s', 'PSRK_groups', 'Pts', 'PubChems', 'rhocs', 'rhocs_mass', 'rhol_STPs', 'rhol_STPs_mass', 'RIs', 'S0gs', 'S0gs_mass', 'Sfgs', 'Sfgs_mass', 'similarity_variables', 'Skins', 'smiless', 'STELs', 'StielPolars', 'Stockmayers', 'Tautoignitions', 'Tbs', 'Tcs', 'Tflashs', 'Tms', 'Tts', 'TWAs', 'UFLs', 'UNIFAC_Dortmund_groups', 'UNIFAC_groups', 'Van_der_Waals_areas', 'Van_der_Waals_volumes', 'Vcs', 'Vml_STPs', 'Vml_Tms', 'Zcs', 'UNIFAC_Rs', 'UNIFAC_Qs', 'rhos_Tms', 'Vms_Tms', 'rhos_Tms_mass', 'solubility_parameters', 'Vml_60Fs', 'rhol_60Fs', 'rhol_60Fs_mass', 'conductivity_Ts', 'RI_Ts', 'Vmg_STPs', 'rhog_STPs', 'rhog_STPs_mass', 'sigma_STPs', 'sigma_Tms', 'sigma_Tbs', 'Hf_STPs', 'Hf_STPs_mass')

Tuple of all properties that can be held by this object.

subset(*idxs=None*, *properties=None*)

Method to construct a new ChemicalConstantsPackage that removes all components not specified in the *idxs* argument. Although this class has a great many attributes, it is often sufficient to work with a subset of those properties; and if a list of properties is provided, only those properties will be added to the new object as well.

Parameters

idxs [list[int] or Slice or None] Indexes of components that should be included; if None, all components will be included , [-]

properties [tuple[str] or None] List of properties to be included; all properties will be included if this is not specified

Returns

subset_consts [ChemicalConstantsPackage] Object with reduced properties and or components, [-]

Notes

It is not intended for properties to be edited in this object! One optimization is that all entirely empty properties use the same list-of-Nones.

All properties should have been specified before constructing the first ChemicalConstantsPackage.

Examples

```
>>> base = ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165],
↪names=['water', 'o-xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.
↪324, 0.331], Pcs=[22048320.0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14,
↪630.3, 616.2, 617.0])
>>> base.subset([0])
ChemicalConstantsPackage(MWs=[18.01528], names=['water'], omegas=[0.344],
↪Pcs=[22048320.0], Tcs=[647.14])
>>> base.subset(slice(1,4))
ChemicalConstantsPackage(MWs=[106.165, 106.165, 106.165], names=['o-xylene', 'p-
↪xylene', 'm-xylene'], omegas=[0.3118, 0.324, 0.331], Pcs=[3732000.0, 3511000.
↪0, 3541000.0], Tcs=[630.3, 616.2, 617.0])
>>> base.subset(idxs=[0, 3], properties=('names', 'MWs'))
ChemicalConstantsPackage(MWs=[18.01528, 106.165], names=['water', 'm-xylene'])
```

with_new_constants(***kwargs*)

Method to construct a new ChemicalConstantsPackage that replaces or adds one or more properties for all components.

Parameters

kwargs [dict[str: list[float]]] Properties specified by name [various]

Returns

new_constants [ChemicalConstantsPackage] Object with new and/or replaced properties, [-]

Examples

```
>>> base = ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165],
↪names=['water', 'o-xylene', 'p-xylene', 'm-xylene'], omegas=[0.344, 0.3118, 0.
↪324, 0.331], Pcs=[22048320.0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14,
↪630.3, 616.2, 617.0])
>>> base.with_new_constants(Tms=[40.0, 20.0, 10.0, 30.0], omegas=[0.0, 0.1, 0.2,
↪ 0.3])
ChemicalConstantsPackage(MWs=[18.01528, 106.165, 106.165, 106.165], names=[
↪'water', 'o-xylene', 'p-xylene', 'm-xylene'], omegas=[0.0, 0.1, 0.2, 0.3],
↪Pcs=[22048320.0, 3732000.0, 3511000.0, 3541000.0], Tcs=[647.14, 630.3, 616.2,
↪617.0], Tms=[40.0, 20.0, 10.0, 30.0])
```

7.4.2 Chemical Correlations Class

class thermo.chemical_package.**PropertyCorrelationsPackage**(*constants*, *VaporPressures=None*, *SublimationPressures=None*, *VolumeGases=None*, *VolumeLiquids=None*, *VolumeSolids=None*, *HeatCapacityGases=None*, *HeatCapacityLiquids=None*, *HeatCapacitySolids=None*, *ViscosityGases=None*, *ViscosityLiquids=None*, *ThermalConductivityGases=None*, *ThermalConductivityLiquids=None*, *EnthalpyVaporizations=None*, *EnthalpySublimations=None*, *SurfaceTensions=None*, *PermittivityLiquids=None*, *VolumeGasMixtureObj=None*, *VolumeLiquidMixtureObj=None*, *VolumeSolidMixtureObj=None*, *HeatCapacityGasMixtureObj=None*, *HeatCapacityLiquidMixtureObj=None*, *HeatCapacitySolidMixtureObj=None*, *ViscosityGasMixtureObj=None*, *ViscosityLiquidMixtureObj=None*, *ThermalConductivityGasMixtureObj=None*, *ThermalConductivityLiquidMixtureObj=None*, *SurfaceTensionMixtureObj=None*, *skip_missing=False*)

Class for creating and storing T and P and zs dependent chemical property objects. All parameters are also attributes.

This object can be used either to hold already-created property objects; or to create new ones and hold them.

Parameters

constants [ChemicalConstantsPackage] Object holding all constant properties, [-]

VaporPressures [list[thermo.vapor_pressure.VaporPressure], optional] Objects holding vapor pressure data and methods, [-]

SublimationPressures [list[thermo.vapor_pressure.SublimationPressure], optional] Objects holding sublimation pressure data and methods, [-]

VolumeGases [list[thermo.volume.VolumeGas], optional] Objects holding gas volume data and methods, [-]

VolumeLiquids [list[thermo.volume.VolumeLiquid], optional] Objects holding liquid volume data and methods, [-]

VolumeSolids [list[thermo.volume.VolumeSolid], optional] Objects holding solid volume data and methods, [-]

HeatCapacityGases [list[thermo.heat_capacity.HeatCapacityGas], optional] Objects holding gas heat capacity data and methods, [-]

HeatCapacityLiquids [list[*thermo.heat_capacity.HeatCapacityLiquid*], optional] Objects holding liquid heat capacity data and methods, [-]

HeatCapacitySolids [list[*thermo.heat_capacity.HeatCapacitySolid*], optional] Objects holding solid heat capacity data and methods, [-]

ViscosityGases [list[*thermo.viscosity.ViscosityGas*], optional] Objects holding gas viscosity data and methods, [-]

ViscosityLiquids [list[*thermo.viscosity.ViscosityLiquid*], optional] Objects holding liquid viscosity data and methods, [-]

ThermalConductivityGases [list[*thermo.thermal_conductivity.ThermalConductivityGas*], optional] Objects holding gas thermal conductivity data and methods, [-]

ThermalConductivityLiquids [list[*thermo.thermal_conductivity.ThermalConductivityLiquid*], optional] Objects holding liquid thermal conductivity data and methods, [-]

EnthalpyVaporizations [list[*thermo.phase_change.EnthalpyVaporization*], optional] Objects holding enthalpy of vaporization data and methods, [-]

EnthalpySublimations [list[*thermo.phase_change.EnthalpySublimation*], optional] Objects holding enthalpy of sublimation data and methods, [-]

SurfaceTensions [list[*thermo.interface.SurfaceTension*], optional] Objects holding surface tension data and methods, [-]

PermittivityLiquids [list[*thermo.permittivity.PermittivityLiquid*], optional] Objects holding permittivity data and methods, [-]

skip_missing [bool, optional] If False, any properties not provided will have objects created; if True, no extra objects will be created.

VolumeSolidMixture [*thermo.volume.VolumeSolidMixture*, optional] Predictor object for the volume of solid mixtures, [-]

VolumeLiquidMixture [*thermo.volume.VolumeLiquidMixture*, optional] Predictor object for the volume of liquid mixtures, [-]

VolumeGasMixture [*thermo.volume.VolumeGasMixture*, optional] Predictor object for the volume of gas mixtures, [-]

HeatCapacityLiquidMixture [*thermo.heat_capacity.HeatCapacityLiquidMixture*, optional] Predictor object for the heat capacity of liquid mixtures, [-]

HeatCapacityGasMixture [*thermo.heat_capacity.HeatCapacityGasMixture*, optional] Predictor object for the heat capacity of gas mixtures, [-]

HeatCapacitySolidMixture [*thermo.heat_capacity.HeatCapacitySolidMixture*, optional] Predictor object for the heat capacity of solid mixtures, [-]

ViscosityLiquidMixture [*thermo.viscosity.ViscosityLiquidMixture*, optional] Predictor object for the viscosity of liquid mixtures, [-]

ViscosityGasMixture [*thermo.viscosity.ViscosityGasMixture*, optional] Predictor object for the viscosity of gas mixtures, [-]

ThermalConductivityLiquidMixture [*thermo.thermal_conductivity.ThermalConductivityLiquidMixture*, optional] Predictor object for the thermal conductivity of liquid mixtures, [-]

ThermalConductivityGasMixture [*thermo.thermal_conductivity.ThermalConductivityGasMixture*, optional] Predictor object for the thermal conductivity of gas mixtures, [-]

SurfaceTensionMixture [*thermo.interface.SurfaceTensionMixture*, optional] Predictor object for the surface tension of liquid mixtures, [-]

Examples

Create a package from CO2 and n-hexane, with ideal-gas heat capacities provided while excluding all other properties:

```
>>> constants = ChemicalConstantsPackage(CASs=['124-38-9', '110-54-3'], MWs=[44.
↪0095, 86.17536], names=['carbon dioxide', 'hexane'], omegas=[0.2252, 0.2975],␣
↪Pcs=[7376460.0, 3025000.0], Tbs=[194.67, 341.87], Tcs=[304.2, 507.6], Tms=[216.65,
↪ 178.075])
>>> correlations = PropertyCorrelationsPackage(constants=constants, skip_
↪missing=True, HeatCapacityGases=[HeatCapacityGas(poly_fit=(50.0, 1000.0, [-3.
↪1115474168865828e-21, 1.39156078498805e-17, -2.5430881416264243e-14, 2.
↪4175307893014295e-11, -1.2437314771044867e-08, 3.1251954264658904e-06, -0.
↪00021220221928610925, 0.000884685506352987, 29.266811602924644])),␣
↪HeatCapacityGas(poly_fit=(200.0, 1000.0, [1.3740654453881647e-21, -8.
↪344496203280677e-18, 2.2354782954548568e-14, -3.4659555330048226e-11, 3.
↪410703030634579e-08, -2.1693611029230923e-05, 0.008373280796376588, -1.
↪356180511425385, 175.67091124888998]))])
```

Create a package from various data files, creating all property objects:

```
>>> correlations = PropertyCorrelationsPackage(constants=constants, skip_
↪missing=False)
```

Attributes

pure_correlations [tuple(str)] List of all pure component property objects, [-]

Methods

subset(idxs)	Method to construct a new PropertyCorrelationsPackage that removes all components not specified in the *idxs* argument.

__add__(*b*)
Method to create a new *PropertyCorrelationsPackage* object from two other *PropertyCorrelationsPackage* objects.

Returns

new [*PropertyCorrelationsPackage*] New object, [-]

Examples

```
>>> a = ChemicalConstantsPackage.correlations_from_IDs(IDs=['water', 'hexane'])
>>> b = ChemicalConstantsPackage.correlations_from_IDs(IDs=['toluene'])
>>> c = a + b
```

subset(*idxs*)
Method to construct a new PropertyCorrelationsPackage that removes all components not specified in the *idxs* argument.

Parameters

idxs [list[int] or Slice or None] Indexes of components that should be included; if None, all components will be included , [-]

Returns

subset_correlations [PropertyCorrelationsPackage] Object with components, [-]

7.4.3 Sample Constants and Correlations

thermo.chemical_package.**iapws_constants = ChemicalConstantsPackage(CASs=['7732-18-5'], MWs=[18.015268], omegas=[0.344], Pcs=[22064000.0], Tcs=[647.096])**
ChemicalConstantsPackage : Object intended to hold the IAPWS-95 water constants for use with the *thermo.phases.IAPWS95* phase object.

thermo.chemical_package.**iapws_correlations = <thermo.chemical_package.PropertyCorrelationsPackage object>**
PropertyCorrelationsPackage: IAPWS correlations and properties, [-]

thermo.chemical_package.**lemmon2000_constants = ChemicalConstantsPackage(CASs=['132259-10-0'], MWs=[28.9586], omegas=[0.0335], Pcs=[3785020.0], Tcs=[132.6312])**
ChemicalConstantsPackage : Object intended to hold the Lemmon (2000) air constants for use with the *thermo.phases.DryAirLemmon* phase object.

thermo.chemical_package.**lemmon2000_correlations = <thermo.chemical_package.PropertyCorrelationsPackage object>**
PropertyCorrelationsPackage: Lemmon (2000) air correlations and properties, [-]

7.5 Creating Property Datasheets (thermo.datasheet)

thermo.datasheet.**tabulate_constants**(*chemical*, *full=False*, *vertical=False*)

thermo.datasheet.**tabulate_gas**(*chemical*, *Tmin=None*, *Tmax=None*, *pts=10*)

thermo.datasheet.**tabulate_liq**(*chemical*, *Tmin=None*, *Tmax=None*, *pts=10*)

thermo.datasheet.**tabulate_solid**(*chemical*, *Tmin=None*, *Tmax=None*, *pts=10*)

thermo.datasheet.**tabulate_streams**(*names=None*, **args*, ***kwargs*)

7.6 Electrochemistry (thermo.electrochem)

This module contains models for:

- Pure substance electrical conductivity lookups
- Correlations for aqueous electrolyte heat capacity, density, and viscosity
- Aqueous electrolyte conductivity
- Water equilibrium constants
- Balancing experimental ion analysis results so as to meet the electroneutrality condition

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

7.6.1 Aqueous Electrolyte Density

`thermo.electrochem.Laliberte_density`(*T*, *ws*, *CASRNs*)

Calculate the density of an aqueous electrolyte mixture using the form proposed by [1]. Parameters are loaded by the function as needed. Units are Kelvin and Pa*s.

$$\rho_m = \left(\frac{w_w}{\rho_w} + \sum_i \frac{w_i}{\rho_{app_i}}\right)^{-1}$$

Parameters

T [float] Temperature of fluid [K]

ws [array] Weight fractions of fluid components other than water

CASRNs [array] CAS numbers of the fluid components other than water

Returns

rho [float] Solution density, [kg/m^3]

Notes

Temperature range check is not used here.

References

[1]

Examples

```
>>> Laliberte_density(273.15, [0.0037838838], ['7647-14-5'])
1002.62501201
```

thermo.electrochem.**Laliberte_density_mix**(*T*, *ws*, *c0s*, *c1s*, *c2s*, *c3s*, *c4s*)

Calculate the density of an aqueous electrolyte mixture using the form proposed by [1]. All parameters must be provided to the function. Units are Kelvin and Pa*s.

$$\rho_m = \left(\frac{w_w}{\rho_w} + \sum_i \frac{w_i}{\rho_{app_i}}\right)^{-1}$$

Parameters

T [float] Temperature of fluid [K]

ws [array] Weight fractions of fluid components other than water

c0s [list[float]] Fit coefficient, [-]

c1s [list[float]] Fit coefficient, [-]

c2s [list[float]] Fit coefficient, [-]

c3s [list[float]] Fit coefficient, [1/degC]

c4s [list[float]] Fit coefficient, [degC]

Returns

rho [float] Solution density, [kg/m^3]

References

[1]

Examples

```
>>> Laliberte_density_mix(T=278.15, ws=[0.00581, 0.002], c0s=[-0.00324112223655149, 0.967814929691928], c1s=[0.0636354335906616, 5.540434135986], c2s=[1.01371399467365, 1.10374669742622], c3s=[0.0145951015210159, 0.0123340782160061], c4s=[3317.34854426537, 2589.61875022366])
1005.6947727219
```

thermo.electrochem.**Laliberte_density_i**(*T*, *w_w*, *c0*, *c1*, *c2*, *c3*, *c4*)

Calculate the density of a solute using the form proposed by Laliberte [1]. Parameters are needed, and a temperature, and water fraction. Units are Kelvin and Pa*s.

$$\rho_{app,i} = \frac{(c_0[1-w_w]+c_1)\exp(10^{-6}[t+c_4]^2)}{(1-w_w) + c_2 + c_3 t}$$

Parameters

T [float] Temperature of fluid [K]

w_w [float] Weight fraction of water in the solution, [-]

c0 [float] Fit coefficient, [-]

c1 [float] Fit coefficient, [-]

c2 [float] Fit coefficient, [-]

c3 [float] Fit coefficient, [1/degC]

c4 [float] Fit coefficient, [degC]

Returns

rho_i [float] Solute partial density, [kg/m^3]

Notes

Temperature range check is not used here.

References

[1]

Examples

```
>>> params = [-0.00324112223655149, 0.0636354335906616, 1.01371399467365, 0.
↪0145951015210159, 3317.34854426537]
>>> Laliberte_density_i(273.15+0, 1-0.0037838838, *params)
3761.8917585
```

thermo.electrochem.**Laliberte_density_w**(*T*)

Calculate the density of water using the form proposed by [1]. No parameters are needed, just a temperature. Units are Kelvin and kg/m^3.

$$\rho_w = \frac{\left\{\left([(-2.8054253\times 10^{-10}\cdot t + 1.0556302\times 10^{-7})t - 4.6170461\times 10^{-5}]t - 0.0079870401\right)t + 16.945176\right\}t + 999.83}{1 + 0.01687985\cdot t}$$

Parameters

T [float] Temperature of fluid [K]

Returns

rho_w [float] Water density, [kg/m^3]

Notes

Original source not cited No temperature range is used.

References

[1]

Examples

```
>>> Laliberte_density_w(298.15)
997.0448954179155
>>> Laliberte_density_w(273.15 + 50)
988.0362916114763
```

7.6.2 Aqueous Electrolyte Heat Capacity

thermo.electrochem.**Laliberte_heat_capacity**(*T*, *ws*, *CASRNs*)

Calculate the heat capacity of an aqueous electrolyte mixture using the form proposed by [1]. Parameters are loaded by the function as needed.

$$Cp_m = w_w Cp_w + \sum w_i Cp_i$$

Parameters

T [float] Temperature of fluid [K]

ws [array] Weight fractions of fluid components other than water

CASRNs [array] CAS numbers of the fluid components other than water

Returns

Cp [float] Solution heat capacity, [J/kg/K]

Notes

A temperature range check is not included in this function. Units are Kelvin and J/kg/K.

References

[1]

Examples

```
>>> Laliberte_heat_capacity(273.15+1.5, [0.00398447], ['7647-14-5'])
4186.575407596064
```

thermo.electrochem.**Laliberte_heat_capacity_mix**(*T*, *ws*, *a1s*, *a2s*, *a3s*, *a4s*, *a5s*, *a6s*)

Calculate the heat capacity of an aqueous electrolyte mixture using the form proposed by [1]. All parameters must be provided to this function.

$$Cp_m = w_w Cp_w + \sum w_i Cp_i$$

Parameters

T [float] Temperature of fluid [K]

ws [array] Weight fractions of fluid components other than water

CASRNs [array] CAS numbers of the fluid components other than water

Returns

Cp [float] Solution heat capacity, [J/kg/K]

Notes

A temperature range check is not included in this function. Units are Kelvin and J/kg/K.

References

[1]

Examples

```
>>> Laliberte_heat_capacity_mix(T=278.15, ws=[0.00581, 0.002], a1s=[-0.
↪0693559668993322, -0.103713247177424], a2s=[-0.0782134167486952, -0.
↪0647453826944371], a3s=[3.84798479408635, 2.92191453087969], a4s=[-11.
↪2762109247072, -5.48799065938436], a5s=[8.73187698542672, 2.41768600041476],␣
↪a6s=[1.81245930472755, 1.32062411084408])
4154.788562680796
```

thermo.electrochem.**Laliberte_heat_capacity_i**(*T*, *w_w*, *a1*, *a2*, *a3*, *a4*, *a5*, *a6*)

Calculate the heat capacity of a solute using the form proposed by [1] Parameters are needed, and a temperature, and water fraction.

$$Cp_i = a_1 e^\alpha + a_5(1-w_w)^{a_6}$$

$$\alpha = a_2 t + a_3 \exp(0.01t) + a_4(1-w_w)$$

Parameters

T [float] Temperature of fluid [K]

w_w [float] Weight fraction of water in the solution

a1-a6 [floats] Function fit parameters

Returns

Cp_i [float] Solute partial heat capacity, [J/kg/K]

Notes

Units are Kelvin and J/kg/K. Temperature range check is not used here.

References

[1]

Examples

```
>>> params = [-0.0693559668993322, -0.0782134167486952, 3.84798479408635, -11.
↪2762109247072, 8.73187698542672, 1.81245930472755]
>>> Laliberte_heat_capacity_i(1.5+273.15, 1-0.00398447, *params)
-2930.73539458
```

thermo.electrochem.**Laliberte_heat_capacity_w**(T)

Calculate the heat capacity of pure water in a fast but similar way as in [1]. [1] suggested the following interpolatative scheme, using points calculated from IAPWS-97 at a pressure of 0.1 MPa up to 95 °C and then at saturation pressure. The maximum temperature of [1] is 140 °C.

$$Cp_w = Cp_1 + (Cp_2 - Cp_1)\left(\frac{t - t_1}{t_2 - t_1}\right) + \frac{(Cp_3 - 2Cp_2 + Cp_1)}{2}\left(\frac{t - t_1}{t_2 - t_1}\right)\left(\frac{t - t_1}{t_2 - t_1} - 1\right)$$

In this implementation, the heat capacity of water is calculated from a chebyshev approximation of the scheme of [1] up to ~92 °C and then the heat capacity comes directly from IAPWS-95 at higher temperatures, also at the saturation pressure. There is no discontinuity between the methods.

Parameters

T [float] Temperature of fluid [K]

Returns

Cp_w [float] Water heat capacity, [J/kg/K]

Notes

Units are Kelvin and J/kg/K.

References

[1]

Examples

```
>>> Laliberte_heat_capacity_w(273.15+3.56)
4208.878727051538
```

7.6.3 Aqueous Electrolyte Viscosity

`thermo.electrochem.`**`Laliberte_viscosity`**(*T*, *ws*, *CASRNs*)

Calculate the viscosity of an aqueous mixture using the form proposed by [1]. Parameters are loaded by the function as needed. Units are Kelvin and Pa*s.

$$\mu_m = \mu_w^{w_w} \Pi \mu_i^{w_i}$$

Parameters

T [float] Temperature of fluid, [K]

ws [array] Weight fractions of fluid components other than water, [-]

CASRNs [array] CAS numbers of the fluid components other than water, [-]

Returns

mu [float] Viscosity of aqueous mixture, [Pa*s]

Notes

Temperature range check is not used here. Check is performed using NaCl at 5 degC from the first value in [1]'s spreadsheet.

References

[1]

Examples

```
>>> Laliberte_viscosity(273.15+5, [0.005810], ['7647-14-5'])
0.0015285828581961414
```

`thermo.electrochem.`**`Laliberte_viscosity_mix`**(*T*, *ws*, *v1s*, *v2s*, *v3s*, *v4s*, *v5s*, *v6s*)

Calculate the viscosity of an aqueous mixture using the form proposed by [1]. All parameters must be provided in this implementation.

$$\mu_m = \mu_w^{w_w} \Pi \mu_i^{w_i}$$

Parameters

T [float] Temperature of fluid, [K]

ws [array] Weight fractions of fluid components other than water, [-]

v1s [list[float]] Fit parameter, [-]

v2s [list[float]] Fit parameter, [-]

v3s [list[float]] Fit parameter, [-]

v4s [list[float]] Fit parameter, [1/degC]

v5s [list[float]] Fit parameter, [-]

v6s [list[float]] Fit parameter, [-]

Returns

mu [float] Viscosity of aqueous mixture, [Pa*s]

References

[1]

Examples

```
>>> Laliberte_viscosity_mix(T=278.15, ws=[0.00581, 0.002], v1s=[16.221788633396, 69.
↪5769240055845], v2s=[1.32293086770011, 4.17047793905946], v3s=[1.48485985010431,␣
↪3.57817553622189], v4s=[0.00746912559657377, 0.0116677996754397], v5s=[30.
↪7802007540575, 13897.6652650556], v6s=[2.05826852322558, 20.8027689840251])
0.0015377348091189648
```

`thermo.electrochem.Laliberte_viscosity_i`(*T*, *w_w*, *v1*, *v2*, *v3*, *v4*, *v5*, *v6*)

Calculate the viscosity of a solute using the form proposed by [1] Parameters are needed, and a temperature. Units are Kelvin and Pa*s.

$$\mu_i = \frac{\exp\left(\frac{v_1(1-w_w)^{v_2}+v_3}{v_4 t +1}\right) } {v_5(1-w_w)^{v_6}+1}$$

Parameters

T [float] Temperature of fluid, [K]

w_w [float] Weight fraction of water in the solution, [-]

v1 [float] Fit parameter, [-]

v2 [float] Fit parameter, [-]

v3 [float] Fit parameter, [-]

v4 [float] Fit parameter, [1/degC]

v5 [float] Fit parameter, [-]

v6 [float] Fit parameter, [-]

Returns

mu_i [float] Solute partial viscosity, [Pa*s]

Notes

Temperature range check is outside of this function. Check is performed using NaCl at 5 degC from the first value in [1]'s spreadsheet.

References

[1]

Examples

```
>>> params = [16.221788633396, 1.32293086770011, 1.48485985010431, 0.
↪00746912559657377, 30.7802007540575, 2.05826852322558]
>>> Laliberte_viscosity_i(273.15+5, 1-0.005810, *params)
0.004254025533308794
```

thermo.electrochem.**Laliberte_viscosity_w**(*T*)

Calculate the viscosity of a water using the form proposed by [1]. No parameters are needed, just a temperature. Units are Kelvin and Pa*s. t is temperature in degrees Celcius.

$$\mu_w = \frac{t + 246}{(0.05594t+5.2842)t + 137.37}$$

Parameters

T [float] Temperature of fluid, [K]

Returns

mu_w [float] Water viscosity, [Pa*s]

Notes

Original source or pure water viscosity is not cited. No temperature range is given for this equation.

References

[1]

Examples

```
>>> Laliberte_viscosity_w(298)
0.0008932264487033279
```

7.6.4 Aqueous Electrolyte Thermal Conductivity

thermo.electrochem.**thermal_conductivity_Magomedov**(*T*, *P*, *ws*, *CASRNs*, *k_w*)

Calculate the thermal conductivity of an aqueous mixture of electrolytes using the form proposed by Magomedov [1]. Parameters are loaded by the function as needed. Function will fail if an electrolyte is not in the database.

$$\lambda = \lambda_w \left[1 - \sum_{i=1}^n A_i (w_i + 2\times 10^{-4} w_i^3)\right] - 2\times 10^{-8} PT\sum_{i=1}^n w_i$$

Parameters

T [float] Temperature of liquid [K]

P [float] Pressure of the liquid [Pa]

ws [array] Weight fractions of liquid components other than water

CASRNs [array] CAS numbers of the liquid components other than water

k_w [float] Liquid thermal condiuctivity or pure water at T and P, [W/m/K]

Returns

kl [float] Liquid thermal condiuctivity, [W/m/K]

Notes

Range from 273 K to 473 K, P from 0.1 MPa to 100 MPa. C from 0 to 25 mass%. Internal untis are MPa for pressure and weight percent.

An example is sought for this function. It is not possible to reproduce the author's values consistently.

References

[1]

Examples

```
>>> thermal_conductivity_Magomedov(293., 1E6, [.25], ['7758-94-3'], k_w=0.59827)
0.548654049375
```

thermo.electrochem.**Magomedov_mix**(*T*, *P*, *ws*, *Ais*, *k_w*)

Calculate the thermal conductivity of an aqueous mixture of electrolytes using the correlation proposed by Magomedov [1]. All coefficients and the thermal conductivity of pure water must be provided.

$$\lambda = \lambda_w \left[1 - \sum_{i=1}^n A_i (w_i + 2\times10^{-4} w_i^3) \right] - 2\times10^{-8} PT \sum_{i=1}^n w_i$$

Parameters

T [float] Temperature of liquid [K]

P [float] Pressure of the liquid [Pa]

ws [list[float]] Weight fractions of liquid components other than water, [-]

Ais [list[float]] *Ai* coefficients which were regressed, [-]

k_w [float] Liquid thermal condiuctivity or pure water at T and P, [W/m/K]

Returns

kl [float] Liquid thermal condiuctivity, [W/m/K]

Notes

Range from 273 K to 473 K, P from 0.1 MPa to 100 MPa. C from 0 to 25 mass%. Internal untis are MPa for pressure and weight percent.

References

[1]

Examples

```
>>> Magomedov_mix(293., 1E6, [.25], [0.00294], k_w=0.59827)
0.548654049375
```

7.6.5 Aqueous Electrolyte Electrical Conductivity

thermo.electrochem.**dilute_ionic_conductivity**(*ionic_conductivities*, *zs*, *rhom*)

This function handles the calculation of the electrical conductivity of a dilute electrolytic aqueous solution. Requires the mole fractions of each ion, the molar density of the whole mixture, and ionic conductivity coefficients for each ion.

$$\lambda = \sum_i \lambda_i^\circ z_i \rho_m$$

Parameters

ionic_conductivities [list[float]] Ionic conductivity coefficients of each ion in the mixture [m^2*S/mol]

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

rhom [float] Overall molar density of the solution, [mol/m^3]

Returns

kappa [float] Electrical conductivity of the fluid, [S/m]

Notes

The ionic conductivity coefficients should not be *equivalent* coefficients; for example, 0.0053 m^2*S/mol is the equivalent conductivity coefficient of Mg+2, but this method expects twice its value - 0.0106. Both are reported commonly in literature.

Water can be included in this caclulation by specifying a coefficient of 0. The conductivity of any electrolyte eclipses its own conductivity by many orders of magnitude. Any other solvents present will affect the conductivity extensively and there are few good methods to predict this effect.

References

[1]

Examples

Complex mixture of electrolytes ['Cl-', 'HCO3-', 'SO4-2', 'Na+', 'K+', 'Ca+2', 'Mg+2']:

```
>>> ionic_conductivities = [0.00764, 0.00445, 0.016, 0.00501, 0.00735, 0.0119, 0.
↪01061]
>>> zs = [0.03104, 0.00039, 0.00022, 0.02413, 0.0009, 0.0024, 0.00103]
>>> dilute_ionic_conductivity(ionic_conductivities=ionic_conductivities, zs=zs,␣
↪rhom=53865.9)
22.05246783663
```

thermo.electrochem.**conductivity_McCleskey**(*T*, *M*, *lambda_coeffs*, *A_coeffs*, *B*, *multiplier*, *rho=1000.0*)

This function handles the calculation of the electrical conductivity of an electrolytic aqueous solution with one electrolyte in solution. It handles temperature dependency and concentrated solutions. Requires the temperature of the solution; its molality, and four sets of coefficients *lambda_coeffs*, *A_coeffs*, *B*, and *multiplier*.

$$\Lambda = \frac{\kappa}{C}$$

$$\Lambda = \Lambda^0(t) - A(t)\frac{m^{1/2}}{1+Bm^{1/2}}$$

$$\Lambda^\circ(t) = c_1 t^2 + c_2 t + c_3$$

$$A(t) = d_1 t^2 + d_2 t + d_3$$

In the above equations, t is temperature in degrees Celcius; m is molality in mol/kg, and C is the concentration of the elctrolytes in mol/m^3, calculated as the product of density and molality.

Parameters

T [float] Temperature of the solution, [K]

M [float] Molality of the solution with respect to one electrolyte (mol solute / kg solvent), [mol/kg]

lambda_coeffs [list[float]] List of coefficients for the polynomial used to calculate *lambda*; length-3 coefficients provided in [1], [-]

A_coeffs [list[float]] List of coefficients for the polynomial used to calculate *A*; length-3 coefficients provided in [1], [-]

B [float] Empirical constant for an electrolyte, [-]

multiplier [float] The multiplier to obtain the absolute conductivity from the equivalent conductivity; ex 2 for CaCl2, [-]

rho [float, optional] The mass density of the aqueous mixture, [kg/m^3]

Returns

kappa [float] Electrical conductivity of the solution at the specified molality and temperature [S/m]

Notes

Coefficients provided in [1] result in conductivity being calculated in units of mS/cm; they are converted to S/m before returned.

References

[1]

Examples

A 0.5 wt% solution of CaCl2, conductivity calculated in mS/cm

```
>>> conductivity_McCleskey(T=293.15, M=0.045053, A_coeffs=[.03918, 3.905,
... 137.7], lambda_coeffs=[0.01124, 2.224, 72.36], B=3.8, multiplier=2)
0.8482584585108555
```

thermo.electrochem.**ionic_strength**(*mis*, *zis*)

Calculate the ionic strength of a solution in one of two ways, depending on the inputs only. For Pitzer and Bromley models, *mis* should be molalities of each component. For eNRTL models, *mis* should be mole fractions of each electrolyte in the solution. This will sum to be much less than 1.

$$I = \frac{1}{2} \sum M_i z_i^2$$
$$I = \frac{1}{2} \sum x_i z_i^2$$

Parameters

mis [list] Molalities of each ion, or mole fractions of each ion [mol/kg or -]

zis [list] Charges of each ion [-]

Returns

I [float] ionic strength, [?]

References

[1], [2]

Examples

```
>>> ionic_strength([0.1393, 0.1393], [1, -1])
0.1393
```

7.6.6 Pure Liquid Electrical Conductivity

`thermo.electrochem.conductivity(CASRN, method=None)`

This function handles the retrieval of a chemical's conductivity. 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 100 chemicals.

Parameters

CASRN [string] CASRN [-]

Returns

kappa [float] Electrical conductivity of the fluid, [S/m]

T [float or None] Temperature at which conductivity measurement was made or None if not available, [K]

Other Parameters

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

Notes

Only one source is available in this function. It is:

- 'LANGE_COND' which is from Lange's Handbook, Table 8.34 Electrical Conductivity of Various Pure Liquids', a compillation of data in [1]. The individual datapoints in this source are not cited at all.

References

[1]

Examples

```
>>> conductivity('7732-18-5')
(4e-06, 291.15)
```

`thermo.electrochem.conductivity_methods(CASRN)`

Return all methods available to obtain electrical conductivity for the specified chemical.

Parameters

CASRN [str] CASRN, [-]

Returns

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

See also:

conductivity

thermo.electrochem.**conductivity_all_methods = ['LANGE_COND']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

7.6.7 Water Dissociation Equilibrium

thermo.electrochem.**Kweq_Arcis_Tremaine_Bandura_Lvov**(*T*, *rho_w*)

Calculates equilibrium constant for OH- and H+ in water, according to [1].

$$Q = \rho\exp(\alpha_0 + \alpha_1 T^{-1} + \alpha_2 T^{-2} \rho^{2/3})$$

$$-\log_{10} K_w = -2n\left[\log_{10}(1+Q) - \frac{Q}{Q+1}\rho(\beta_0 + \beta_1 T^{-1} + \beta_2 \rho)\right] - \log_{10} K_w^G + 2\log_{10}\frac{18.015268}{1000}$$

Parameters

T [float] Temperature of water [K]

rho_w [float] Density of water at temperature and pressure [kg/m^3]

Returns

Kweq [float] Ionization constant of water, [-]

Notes

Formulation is in terms of density in g/cm^3; density is converted internally.

n = 6; alpha0 = -0.864671; alpha1 = 8659.19; alpha2 = -22786.2; beta0 = 0.642044; beta1 = -56.8534; beta2 = -0.375754

References

[1]

Examples

```
>>> -1*log10(Kweq_Arcis_Tremaine_Bandura_Lvov(600, 700))
11.138236348
```

thermo.electrochem.**Kweq_IAPWS**(*T*, *rho_w*)

Calculates equilibrium constant for OH- and H+ in water, according to [1]. This is the most recent formulation available.

$$Q = \rho\exp(\alpha_0 + \alpha_1 T^{-1} + \alpha_2 T^{-2} \rho^{2/3})$$

$$-\log_{10} K_w = -2n\left[\log_{10}(1+Q) - \frac{Q}{Q+1}\rho(\beta_0 + \beta_1 T^{-1} + \beta_2 \rho)\right] - \log_{10} K_w^G + 2\log_{10}\frac{18.015268}{1000}$$

Parameters

T [float] Temperature of water [K]

rho_w [float] Density of water at temperature and pressure [kg/m^3]

Returns

Kweq [float] Ionization constant of water, [-]

Notes

Formulation is in terms of density in g/cm^3; density is converted internally.

n = 6; alpha0 = -0.864671; alpha1 = 8659.19; alpha2 = -22786.2; beta0 = 0.642044; beta1 = -56.8534; beta2 = -0.375754

References

[1]

Examples

Example from IAPWS check:

```
>>> -1*log10(Kweq_IAPWS(600, 700))
11.203153057603775
```

thermo.electrochem.**Kweq_IAPWS_gas**(*T*)

Calculates equilibrium constant for OH- and H+ in water vapor, according to [1]. This is the most recent formulation available.

$$-log_{10}K_w^G = \gamma_0 + \gamma_1 T^{-1} + \gamma_2 T^{-2} + \gamma_3 T^{-3}$$

Parameters

T [float] Temperature of H2O [K]

Returns

K_w_G [float]

Notes

gamma0 = 6.141500E-1; gamma1 = 4.825133E4; gamma2 = -6.770793E4; gamma3 = 1.010210E7

References

[1]

Examples

```
>>> Kweq_IAPWS_gas(800)
1.4379721554798815e-61
```

thermo.electrochem.**Kweq_1981**(*T*, *rho_w*)

Calculates equilibrium constant for OH- and H+ in water, according to [1]. Second most recent formulation.

$$\log_{10} K_w = A + B/T + C/T^2 + D/T^3 + (E + F/T + G/T^2)\log_{10}\rho_w$$

Parameters

T [float] Temperature of fluid [K]

rho_w [float] Density of water, [kg/m^3]

Returns

Kweq [float] Ionization constant of water, [-]

Notes

Density is internally converted to units of g/cm^3.

A = -4.098; B = -3245.2; C = 2.2362E5; D = -3.984E7; E = 13.957; F = -1262.3; G = 8.5641E5

References

[1]

Examples

```
>>> -1*log10(Kweq_1981(600, 700))
11.274522047
```

7.6.8 Balancing Ions

thermo.electrochem.**balance_ions**(*anions*, *cations*, *anion_zs=None*, *cation_zs=None*, *anion_concs=None*, *cation_concs=None*, *rho_w=997.1*, *method='increase dominant'*, *selected_ion=None*)

Performs an ion balance to adjust measured experimental ion compositions to electroneutrality. Can accept either the actual mole fractions of the ions, or their concentrations in units of [mg/L] as well for convinience.

The default method will locate the most prevalent ion in the type of ion not in excess - and increase it until the two ion types balance.

Parameters

anions [list(ChemicalMetadata)] List of all negatively charged ions measured as being in the solution; ChemicalMetadata instances or simply objects with the attributes *MW* and *charge*, [-]

cations [list(ChemicalMetadata)] List of all positively charged ions measured as being in the solution; ChemicalMetadata instances or simply objects with the attributes *MW* and *charge*, [-]

anion_zs [list, optional] Mole fractions of each anion as measured in the aqueous solution, [-]

cation_zs [list, optional] Mole fractions of each cation as measured in the aqueous solution, [-]

anion_concs [list, optional] Concentrations of each anion in the aqueous solution in the units often reported (for convinience only) [mg/L]

cation_concs [list, optional] Concentrations of each cation in the aqueous solution in the units often reported (for convinience only) [mg/L]

rho_w [float, optional] Density of the aqueous solutionr at the temperature and pressure the anion and cation concentrations were measured (if specified), [kg/m^3]

method [str, optional] The method to use to balance the ionimbalance; one of 'dominant', 'decrease dominant', 'increase dominant', 'proportional insufficient ions increase', 'proportional excess ions decrease', 'proportional cation adjustment', 'proportional anion adjustment', 'Na or Cl increase', 'Na or Cl decrease', 'adjust', 'increase', 'decrease', 'makeup'].

selected_ion [ChemicalMetadata, optional] Some methods adjust only one user-specified ion; this is that input. For the case of the 'makeup' method, this is a tuple of (anion, cation) ChemicalMetadata instances and only the ion type not in excess will be used.

Returns

anions [list[ChemicalMetadata]] List of all negatively charged ions measured as being in the solution; ChemicalMetadata instances after potentially adding in an ion which was not present but specified by the user, [-]

cations [list[ChemicalMetadata]] List of all positively charged ions measured as being in the solution; ChemicalMetadata instances after potentially adding in an ion which was not present but specified by the user, [-]

anion_zs [list[float],] Mole fractions of each anion in the aqueous solution after the charge balance, [-]

cation_zs [list[float]] Mole fractions of each cation in the aqueous solution after the charge balance, [-]

z_water [float[float]] Mole fraction of the water in the solution, [-]

Notes

The methods perform the charge balance as follows:

- 'dominant' : The ion with the largest mole fraction in solution has its concentration adjusted up or down as necessary to balance the solution.
- 'decrease dominant' : The ion with the largest mole fraction in the type of ion with *excess* charge has its own mole fraction decreased to balance the solution.
- 'increase dominant' : The ion with the largest mole fraction in the type of ion with *insufficient* charge has its own mole fraction decreased to balance the solution.
- 'proportional insufficient ions increase' : The ion charge type which is present insufficiently has each of the ions mole fractions *increased* proportionally until the solution is balanced.
- 'proportional excess ions decrease' : The ion charge type which is present in excess has each of the ions mole fractions *decreased* proportionally until the solution is balanced.
- 'proportional cation adjustment' : All *cations* have their mole fractions increased or decreased proportionally as necessary to balance the solution.
- 'proportional anion adjustment' : All *anions* have their mole fractions increased or decreased proportionally as necessary to balance the solution.
- 'Na or Cl increase' : Either Na+ or Cl- is *added* to the solution until the solution is balanced; the species will be added if they were not present initially as well.
- 'Na or Cl decrease' : Either Na+ or Cl- is *removed* from the solution until the solution is balanced; the species will be added if they were not present initially as well.
- 'adjust' : An ion specified with the parameter *selected_ion* has its mole fraction *increased or decreased* as necessary to balance the solution. An exception is raised if the specified ion alone cannot balance the solution.

- 'increase' : An ion specified with the parameter *selected_ion* has its mole fraction *increased* as necessary to balance the solution. An exception is raised if the specified ion alone cannot balance the solution.
- 'decrease' : An ion specified with the parameter *selected_ion* has its mole fraction *decreased* as necessary to balance the solution. An exception is raised if the specified ion alone cannot balance the solution.
- 'makeup' : Two ions ase specified as a tuple with the parameter *selected_ion*. Whichever ion type is present in the solution insufficiently is added; i.e. if the ions were Mg+2 and Cl-, and there was too much negative charge in the solution, Mg+2 would be added until the solution was balanced.

Examples

```
>>> anions_n = ['Cl-', 'HCO3-', 'SO4-2']
>>> cations_n = ['Na+', 'K+', 'Ca+2', 'Mg+2']
>>> cations = [identifiers.pubchem_db.search_name(i) for i in cations_n]
>>> anions = [identifiers.pubchem_db.search_name(i) for i in anions_n]
>>> an_res, cat_res, an_zs, cat_zs, z_water = balance_ions(anions, cations,
... anion_zs=[0.02557, 0.00039, 0.00026], cation_zs=[0.0233, 0.00075,
... 0.00262, 0.00119], method='proportional excess ions decrease')
>>> an_zs
[0.02557, 0.00039, 0.00026]
>>> cat_zs
[0.01948165456267761, 0.00062709188506472299, 0.0021906409851594564, 0.
↪0009949857909693717]
>>> z_water
0.9504856267761288
```

7.6.9 Fit Coefficients and Data

All of these coefficients are lazy-loaded, so they must be accessed as an attribute of this module.

```
In [1]: from thermo.electrochem import Magomedovk_thermal_cond, cond_data_McCleskey, CRC_
↪aqueous_thermodynamics, electrolyte_dissociation_reactions, Laliberte_data

In [2]: Magomedovk_thermal_cond
Out[2]:
              Formula              Chemical       Ai
CASRN
497-19-8       Na2CO3      Sodium carbonate  -0.00050
584-08-7        K2CO3   Potassium carbonate   0.00160
7447-39-4       CuCl2      Cuprous chloride   0.00360
7488-54-2      Rb2SO4      Rubidium sulfate   0.00134
7601-89-0      NaClO4    Sodium perchlorate   0.00250
7646-79-9       CoCl2    Cobaltous chloride   0.00320
7664-93-9       H2SO4          Acid sulfate   0.00305
7699-45-8       ZnBr2          Zinc bromide   0.00410
7718-54-9       NiCl2    Nickelous chloride   0.00330
7758-94-3       FeCl2      Ferrous chloride   0.00294
7761-88-8       AgNO3        Silver nitrate   0.00190
7775-09-9      NaClO3       Sodium chlorate   0.00240
7778-50-9    K2Cr2O7  Potassium dichromate   0.00188
7786-81-4       NiSO4      Nickelous sulfate   0.00140
```

(continues on next page)

(continued from previous page)

```
7789-00-6      K2CrO4   Potassium chromate   0.00130
7789-23-3          KF   Potassium fluoride   0.00180
7789-38-0      NaBrO3       Sodium bromate   0.00170
7789-39-1        RbBr     Rubidium bromide   0.00305
7789-42-6       CdBr2      Cadmium bromide   0.00274
7789-46-0       FeBr2      Ferrous bromide   0.00375
7790-29-6         RbI      Rubidium iodide   0.00322
7790-80-9        CdI2       Cadmium iodide   0.00302
7791-11-9        RbCl     Rubidium chloride   0.00238
10042-76-9   Sr(NO3)2    Strontium nitrate   0.00153
10043-01-3  Al2(SO4)3     Aluminum sulfate   0.00335
10099-74-8   Pb(NO3)2         Lead nitrate   0.00138
10102-68-8        CaI2       Calcium iodide   0.00340
10139-47-6        ZnI2          Zinc iodide   0.00410
10325-94-7   Cd(NO3)2      Cadmium nitrate   0.00155
10377-51-2         LiI       Lithium iodide   0.00435
10377-58-9        MgI2     Magnesium iodide   0.00417
10476-81-0       SrBr2    Strontium bromide   0.00290
10476-85-4       SrCl2   Strontium chloride   0.00170
10476-86-5        SrI2     Strontium iodide   0.00311
12027-06-4        NH4I      Ammonium iodide   0.00480
13126-12-0       RbNO3     Rubidium nitrate   0.00214
13462-88-9       NiBr2    Nickelous bromide   0.00396
13462-90-3        NiI2     Nickelous iodide   0.00393
15238-00-3        CoI2     Cobaltous iodide   0.00384

In [3]: cond_data_McCleskey
Out[3]:
             formula        c1     c2  ...       d3      B  multiplier
CASRN                                  ...
7447-40-7        KCl  0.009385  2.533  ...    44.11   1.70           1
7647-14-5       NaCl  0.008967  2.196  ...    44.55   1.30           1
7647-01-0        HCl -0.006766  6.614  ...    48.53   0.01           1
7447-41-8       LiCl  0.008784  1.996  ...    42.79   1.00           1
7647-17-8       CsCl  0.010080  2.479  ...    41.29   1.40           1
12125-02-9     NH4Cl  0.006575  2.684  ...    30.00   0.70           1
10043-52-4     CaCl2  0.011240  2.224  ...   137.70   3.80           2
7786-30-3      MgCl2  0.009534  2.247  ...   129.80   3.10           2
10361-37-2     BaCl2  0.010380  2.346  ...   111.80   2.40           2
10476-85-4     SrCl2  0.009597  2.279  ...    60.18   0.80           2
7664-93-9      H2SO4 -0.019850  7.421  ...  1869.00  11.50           2
7757-82-6     Na2SO4  0.009501  2.317  ...   135.50   2.20           2
7778-80-5      K2SO4  0.008819  2.872  ...   247.10   5.30           2
10294-54-9    Cs2SO4  0.012730  2.457  ...   187.40   3.30           2
7778-18-9      CaSO4  0.011920  2.564  ...   644.40   9.60           2
7646-93-7      KHSO4 -0.003092  9.759  ...  1776.00   8.20           1
298-14-6       KHCO3  0.007807  2.040  ...    38.58   0.90           1
584-08-7       K2CO3  0.011450  2.726  ...    81.12   2.10           2
144-55-8      NaHCO3  0.012600  1.543  ...    52.94   1.10           1
497-19-8      Na2CO3  0.022960  5.211  ...   455.80   4.80           2
1310-73-2       NaOH  0.006936  3.872  ...    56.76   0.20           1
7681-49-4        NaF  0.007346  2.032  ...    69.99   2.30           1
```

(continues on next page)

(continued from previous page)

```
7789-23-3         KF  0.007451  2.294  ...    39.40   0.90           1
7758-02-3        KBr  0.007076  2.612  ...    29.49   0.60           1
7757-79-1       KNO3  0.009117  2.309  ...    49.12   0.70           1
7779-88-6   Zn(NO3)2  0.015260  4.519  ...   302.90   4.00           2

[26 rows x 9 columns]

In [4]: CRC_aqueous_thermodynamics
Out[4]:
                   Formula                            Name  ...  S(aq)  Cp(aq)
CAS                                                         ...
57-12-5                CN-                     Cyanide ion  ...   94.1     NaN
71-47-6              CHOO-                     Formate ion  ...   92.0   -87.9
71-50-1            CH3COO-                     Acetate ion  ...   86.6    -6.3
71-52-3              HCO3-                 Bicarbonate ion  ...   91.2     NaN
302-04-5              SCN-                 Thiocyanate ion  ...  144.3   -40.2
...                    ...                             ...  ...    ...     ...
117412-24-5         BeO2-2           Beryllium dioxide ion  ... -159.0     NaN
127622-32-6         Y(OH)+2           Yttrium hydroxide ion  ...    NaN     NaN
129466-35-9        Te(OH)3+  Tellurium(IV) trihydroxide ion  ...  111.7     NaN
186449-38-7         InOH+2             Indium hydroxide ion  ...  -88.0     NaN
2099995000-00-0  Y2(OH)2+4         Yttrium dihydroxide ion  ...    NaN     NaN

[173 rows x 7 columns]

In [5]: electrolyte_dissociation_reactions
Out[5]:
                  Electrolyte name Electrolyte CAS  ... Cation charge Cation count
0    Diammonium Hydrogen phosphate       7783-28-0  ...             1            2
1                 Ammonium Sulfate       7783-20-2  ...             1            2
2                 ammonium sulfite      10196-04-0  ...             1            2
3               Ammonium phosphate      10361-65-6  ...             1            3
4     Ammonium siliconhexafluoride      16919-19-0  ...             1            2
..                             ...             ...  ...           ...          ...
259                  Zinc selenite      13597-46-1  ...             2            1
260                  Zinc selenate      13597-54-1  ...             2            1
261                   Zinc Nitrate       7779-88-6  ...             2            1
262                  Zinc Chloride       7646-85-7  ...             2            1
263                   Zinc Sulfate       7733-02-0  ...             2            1

[264 rows x 11 columns]

In [6]: Laliberte_data
Out[6]:
                          Name     Formula  ...   Max w.2  No of points in corr.2
CASRN                                       ...
7783-20-2     Ammonium Sulfate   (NH4)2SO4  ...       NaN                     NaN
10043-01-3    Aluminum Sulfate   Al2(SO4)3  ...       NaN                     NaN
7446-70-0    Aluminum Chloride       AlCl3  ...       NaN                     NaN
10022-31-8      Barium Nitrate    Ba(NO3)2  ...  0.047274                    96.0
10361-37-2     Barium Chloride       BaCl2  ...  0.248237                    16.0
...                        ...         ...  ...       ...                     ...
```

(continues on next page)

(continued from previous page)

```
57-50-1              Sucrose    Sucrose  ...        NaN                     NaN
13825-74-6    Titanyl Sulfate     TiOSO4  ...        NaN                     NaN
7779-88-6        Zinc Nitrate   Zn(NO3)2  ...  0.077132                   144.0
7646-85-7       Zinc Chloride      ZnCl2  ...        NaN                     NaN
7733-02-0        Zinc Sulfate      ZnSO4  ...        NaN                     NaN

[109 rows x 32 columns]
```

7.7 Cubic Equations of State (thermo.eos)

This module contains implementations of most cubic equations of state for pure components. This includes Peng-Robinson, SRK, Van der Waals, PRSV, TWU and many other variants.

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

7.7.1 Base Class

class thermo.eos.**GCEOS**

Bases: `object`

Class for solving a generic Pressure-explicit three-parameter cubic equation of state. Does not implement any parameters itself; must be subclassed by an equation of state class which uses it. Works for mixtures or pure species for all properties except fugacity. All properties are derived with the CAS SymPy, not relying on any derivations previously published.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V^2 + \delta V + \epsilon}$$

The main methods (in order they are called) are `GCEOS.solve`, `GCEOS.set_from_PT`, `GCEOS.volume_solutions`, and `GCEOS.set_properties_from_solution`.

`GCEOS.solve` calls `GCEOS.check_sufficient_inputs`, which checks if two of T, P, and V were set. It then solves for the remaining variable. If T is missing, method `GCEOS.solve_T` is used; it is parameter specific, and so must be implemented in each specific EOS. If P is missing, it is directly calculated. If V is missing, it is calculated with the method `GCEOS.volume_solutions`. At this point, either three possible volumes or one user specified volume are known. The value of *a_alpha*, and its first and second temperature derivative are calculated with the EOS-specific method `GCEOS.a_alpha_and_derivatives`.

If V is not provided, `GCEOS.volume_solutions` calculates the three possible molar volumes which are solutions to the EOS; in the single-phase region, only one solution is real and correct. In the two-phase region, all volumes are real, but only the largest and smallest solution are physically meaningful, with the largest being that of the gas and the smallest that of the liquid.

`GCEOS.set_from_PT` is called to sort out the possible molar volumes. For the case of a user-specified V, the possibility of there existing another solution is ignored for speed. If there is only one real volume, the method `GCEOS.set_properties_from_solution` is called with it. If there are two real volumes, `GCEOS.set_properties_from_solution` is called once with each volume. The phase is returned by `GCEOS.set_properties_from_solution`, and the volumes is set to either `GCEOS.V_l` or `GCEOS.V_g` as appropriate.

`GCEOS.set_properties_from_solution` is a large function which calculates all relevant partial derivatives and properties of the EOS. 17 derivatives and excess enthalpy and entropy are calculated first. Finally, it sets all these properties as attibutes for either the liquid or gas phase with the convention of adding on *_l* or *_g* to the variable names, respectively.

Attributes

T [float] Temperature of cubic EOS state, [K]

P [float] Pressure of cubic EOS state, [Pa]

a [float] *a* parameter of cubic EOS; formulas vary with the EOS, [Pa*m^6/mol^2]

b [float] *b* parameter of cubic EOS; formulas vary with the EOS, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of $a\alpha$ calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of $a\alpha$ calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

Zc [float] Critical compressibility of cubic EOS state, [-]

phase [str] One of 'l', 'g', or 'l/g' to represent whether or not there is a liquid-like solution, vapor-like solution, or both available, [-]

raw_volumes [list[(float, complex), 3]] Calculated molar volumes from the volume solver; depending on the state and selected volume solver, imaginary volumes may be represented by 0 or -1j to save the time of actually calculating them, [m^3/mol]

V_l [float] Liquid phase molar volume, [m^3/mol]

V_g [float] Vapor phase molar volume, [m^3/mol]

V [float or None] Molar volume specified as input; otherwise None, [m^3/mol]

Z_l [float] Liquid phase compressibility, [-]

Z_g [float] Vapor phase compressibility, [-]

PIP_l [float] Liquid phase phase identification parameter, [-]

PIP_g [float] Vapor phase phase identification parameter, [-]

dP_dT_l [float] Liquid phase temperature derivative of pressure at constant volume, [Pa/K].

$$\left(\frac{\partial P}{\partial T}\right)_V = \frac{R}{V - b} - \frac{a \frac{d\alpha(T)}{dT}}{V^2 + V\delta + \epsilon}$$

dP_dT_g [float] Vapor phase temperature derivative of pressure at constant volume, [Pa/K].

$$\left(\frac{\partial P}{\partial T}\right)_V = \frac{R}{V - b} - \frac{a \frac{d\alpha(T)}{dT}}{V^2 + V\delta + \epsilon}$$

dP_dV_l [float] Liquid phase volume derivative of pressure at constant temperature, [Pa*mol/m^3].

$$\left(\frac{\partial P}{\partial V}\right)_T = -\frac{RT}{(V - b)^2} - \frac{a\left(-2V - \delta\right)\alpha(T)}{\left(V^2 + V\delta + \epsilon\right)^2}$$

dP_dV_g [float] Gas phase volume derivative of pressure at constant temperature, [Pa*mol/m^3].

$$\left(\frac{\partial P}{\partial V}\right)_T = -\frac{RT}{(V - b)^2} - \frac{a\left(-2V - \delta\right)\alpha(T)}{\left(V^2 + V\delta + \epsilon\right)^2}$$

dV_dT_l [float] Liquid phase temperature derivative of volume at constant pressure, [m^3/(mol*K)].

$$\left(\frac{\partial V}{\partial T}\right)_P = -\frac{\left(\frac{\partial P}{\partial T}\right)_V}{\left(\frac{\partial P}{\partial V}\right)_T}$$

dV_dT_g [float] Gas phase temperature derivative of volume at constant pressure, [m^3/(mol*K)].

$$\left(\frac{\partial V}{\partial T}\right)_P = -\frac{\left(\frac{\partial P}{\partial T}\right)_V}{\left(\frac{\partial P}{\partial V}\right)_T}$$

dV_dP_l [float] Liquid phase pressure derivative of volume at constant temperature, [m^3/(mol*Pa)].

$$\left(\frac{\partial V}{\partial P}\right)_T = -\frac{\left(\frac{\partial V}{\partial T}\right)_P}{\left(\frac{\partial P}{\partial T}\right)_V}$$

dV_dP_g [float] Gas phase pressure derivative of volume at constant temperature, [m^3/(mol*Pa)].

$$\left(\frac{\partial V}{\partial P}\right)_T = -\frac{\left(\frac{\partial V}{\partial T}\right)_P}{\left(\frac{\partial P}{\partial T}\right)_V}$$

dT_dV_l [float] Liquid phase volume derivative of temperature at constant pressure, [K*mol/m^3].

$$\left(\frac{\partial T}{\partial V}\right)_P = \frac{1}{\left(\frac{\partial V}{\partial T}\right)_P}$$

dT_dV_g [float] Gas phase volume derivative of temperature at constant pressure, [K*mol/m^3]. See `GCEOS.set_properties_from_solution` for the formula.

dT_dP_l [float] Liquid phase pressure derivative of temperature at constant volume, [K/Pa].

$$\left(\frac{\partial T}{\partial P}\right)_V = \frac{1}{\left(\frac{\partial P}{\partial T}\right)_V}$$

dT_dP_g [float] Gas phase pressure derivative of temperature at constant volume, [K/Pa].

$$\left(\frac{\partial T}{\partial P}\right)_V = \frac{1}{\left(\frac{\partial P}{\partial T}\right)_V}$$

d2P_dT2_l [float] Liquid phase second derivative of pressure with respect to temperature at constant volume, [Pa/K^2].

$$\left(\frac{\partial^2 P}{\partial T^2}\right)_V = -\frac{a\frac{d^2\alpha(T)}{dT^2}}{V^2 + V\delta + \epsilon}$$

d2P_dT2_g [float] Gas phase second derivative of pressure with respect to temperature at constant volume, [Pa/K^2].

$$\left(\frac{\partial^2 P}{\partial T^2}\right)_V = -\frac{a\frac{d^2\alpha(T)}{dT^2}}{V^2 + V\delta + \epsilon}$$

d2P_dV2_l [float] Liquid phase second derivative of pressure with respect to volume at constant temperature, [Pa*mol^2/m^6].

$$\left(\frac{\partial^2 P}{\partial V^2}\right)_T = 2\left(\frac{RT}{(V-b)^3} - \frac{a\left(2V+\delta\right)^2\alpha(T)}{\left(V^2+V\delta+\epsilon\right)^3} + \frac{a\alpha(T)}{\left(V^2+V\delta+\epsilon\right)^2}\right)$$

d2P_dTdV_l [float] Liquid phase second derivative of pressure with respect to volume and then temperature, [Pa*mol/(K*m^3)].

$$\left(\frac{\partial^2 P}{\partial T \partial V}\right) = -\frac{R}{(V-b)^2} + \frac{a(2V+\delta)\frac{d\alpha(T)}{dT}}{(V^2+V\delta+\epsilon)^2}$$

d2P_dTdV_g [float] Gas phase second derivative of pressure with respect to volume and then temperature, [Pa*mol/(K*m^3)].

$$\left(\frac{\partial^2 P}{\partial T \partial V}\right) = -\frac{R}{(V-b)^2} + \frac{a(2V+\delta)\frac{d\alpha(T)}{dT}}{(V^2+V\delta+\epsilon)^2}$$

H_dep_l [float] Liquid phase departure enthalpy, [J/mol]. See *GCEOS.set_properties_from_solution* for the formula.

H_dep_g [float] Gas phase departure enthalpy, [J/mol]. See *GCEOS.set_properties_from_solution* for the formula.

S_dep_l [float] Liquid phase departure entropy, [J/(mol*K)]. See *GCEOS.set_properties_from_solution* for the formula.

S_dep_g [float] Gas phase departure entropy, [J/(mol*K)]. See *GCEOS.set_properties_from_solution* for the formula.

G_dep_l [float] Liquid phase departure Gibbs energy, [J/mol].

$$G_{dep} = H_{dep} - TS_{dep}$$

G_dep_g [float] Gas phase departure Gibbs energy, [J/mol].

$$G_{dep} = H_{dep} - TS_{dep}$$

Cp_dep_l [float] Liquid phase departure heat capacity, [J/(mol*K)]

$$C_{p,dep} = (C_p - C_v)_{\text{from EOS}} + C_{v,dep} - R$$

Cp_dep_g [float] Gas phase departure heat capacity, [J/(mol*K)]

$$C_{p,dep} = (C_p - C_v)_{\text{from EOS}} + C_{v,dep} - R$$

Cv_dep_l [float] Liquid phase departure constant volume heat capacity, [J/(mol*K)]. See *GCEOS.set_properties_from_solution* for the formula.

Cv_dep_g [float] Gas phase departure constant volume heat capacity, [J/(mol*K)]. See *GCEOS.set_properties_from_solution* for the formula.

c1 [float] Full value of the constant in the *a* parameter, set in some EOSs, [-]

c2 [float] Full value of the constant in the *b* parameter, set in some EOSs, [-]

A_dep_g Departure molar Helmholtz energy from ideal gas behavior for the gas phase, [J/mol].

A_dep_l Departure molar Helmholtz energy from ideal gas behavior for the liquid phase, [J/mol].

beta_g Isobaric (constant-pressure) expansion coefficient for the gas phase, [1/K].

beta_l Isobaric (constant-pressure) expansion coefficient for the liquid phase, [1/K].

Cp_minus_Cv_g Cp - Cv for the gas phase, [J/mol/K].

Cp_minus_Cv_l Cp - Cv for the liquid phase, [J/mol/K].

d2a_alpha_dTdP_g_V Derivative of the temperature derivative of *a_alpha* with respect to pressure at constant volume (varying T) for the gas phase, [J^2/mol^2/Pa^2/K].

d2a_alpha_dTdP_l_V Derivative of the temperature derivative of *a_alpha* with respect to pressure at constant volume (varying T) for the liquid phase, [J^2/mol^2/Pa^2/K].

d2H_dep_dT2_g Second temperature derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K^2].

d2H_dep_dT2_g_P Second temperature derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K^2].

d2H_dep_dT2_g_V Second temperature derivative of departure enthalpy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^2].

d2H_dep_dT2_l Second temperature derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K^2].

d2H_dep_dT2_l_P Second temperature derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K^2].

d2H_dep_dT2_l_V Second temperature derivative of departure enthalpy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^2].

d2H_dep_dTdP_g Temperature and pressure derivative of departure enthalpy at constant pressure then temperature for the gas phase, [(J/mol)/K/Pa].

d2H_dep_dTdP_l Temperature and pressure derivative of departure enthalpy at constant pressure then temperature for the liquid phase, [(J/mol)/K/Pa].

d2P_drho2_g Second derivative of pressure with respect to molar density for the gas phase, [Pa/(mol/m^3)^2].

d2P_drho2_l Second derivative of pressure with respect to molar density for the liquid phase, [Pa/(mol/m^3)^2].

d2P_dT2_PV_g Second derivative of pressure with respect to temperature twice, but with pressure held constant the first time and volume held constant the second time for the gas phase, [Pa/K^2].

d2P_dT2_PV_l Second derivative of pressure with respect to temperature twice, but with pressure held constant the first time and volume held constant the second time for the liquid phase, [Pa/K^2].

d2P_dTdP_g Second derivative of pressure with respect to temperature and, then pressure; and with volume held constant at first, then temperature, for the gas phase, [1/K].

d2P_dTdP_l Second derivative of pressure with respect to temperature and, then pressure; and with volume held constant at first, then temperature, for the liquid phase, [1/K].

d2P_dTdrho_g Derivative of pressure with respect to molar density, and temperature for the gas phase, [Pa/(K*mol/m^3)].

d2P_dTdrho_l Derivative of pressure with respect to molar density, and temperature for the liquid phase, [Pa/(K*mol/m^3)].

d2P_dVdP_g Second derivative of pressure with respect to molar volume and then pressure for the gas phase, [mol/m^3].

d2P_dVdP_l Second derivative of pressure with respect to molar volume and then pressure for the liquid phase, [mol/m^3].

d2P_dVdT_g Alias of `GCEOS.d2P_dTdV_g`

d2P_dVdT_l Alias of `GCEOS.d2P_dTdV_l`

d2P_dVdT_TP_g Second derivative of pressure with respect to molar volume and then temperature at constant temperature then pressure for the gas phase, [Pa*mol/m^3/K].

d2P_dVdT_TP_l Second derivative of pressure with respect to molar volume and then temperature at constant temperature then pressure for the liquid phase, [Pa*mol/m^3/K].

d2rho_dP2_g Second derivative of molar density with respect to pressure for the gas phase, [(mol/m^3)/Pa^2].

d2rho_dP2_l Second derivative of molar density with respect to pressure for the liquid phase, [(mol/m^3)/Pa^2].

d2rho_dPdT_g Second derivative of molar density with respect to pressure and temperature for the gas phase, [(mol/m^3)/(K*Pa)].

d2rho_dPdT_l Second derivative of molar density with respect to pressure and temperature for the liquid phase, [(mol/m^3)/(K*Pa)].

d2rho_dT2_g Second derivative of molar density with respect to temperature for the gas phase, [(mol/m^3)/K^2].

d2rho_dT2_l Second derivative of molar density with respect to temperature for the liquid phase, [(mol/m^3)/K^2].

d2S_dep_dT2_g Second temperature derivative of departure entropy with respect to temperature for the gas phase, [(J/mol)/K^3].

d2S_dep_dT2_g_V Second temperature derivative of departure entropy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^3].

d2S_dep_dT2_l Second temperature derivative of departure entropy with respect to temperature for the liquid phase, [(J/mol)/K^3].

d2S_dep_dT2_l_V Second temperature derivative of departure entropy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^3].

d2S_dep_dTdP_g Temperature and pressure derivative of departure entropy at constant pressure then temperature for the gas phase, [(J/mol)/K^2/Pa].

d2S_dep_dTdP_l Temperature and pressure derivative of departure entropy at constant pressure then temperature for the liquid phase, [(J/mol)/K^2/Pa].

d2T_dP2_g Second partial derivative of temperature with respect to pressure (constant volume) for the gas phase, [K/Pa^2].

d2T_dP2_l Second partial derivative of temperature with respect to pressure (constant temperature) for the liquid phase, [K/Pa^2].

d2T_dPdrho_g Derivative of temperature with respect to molar density, and pressure for the gas phase, [K/(Pa*mol/m^3)].

d2T_dPdrho_l Derivative of temperature with respect to molar density, and pressure for the liquid phase, [K/(Pa*mol/m^3)].

d2T_dPdV_g Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the gas phase, [K*mol/(Pa*m^3)].

d2T_dPdV_l Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the liquid phase, [K*mol/(Pa*m^3)].

d2T_drho2_g Second derivative of temperature with respect to molar density for the gas phase, [K/(mol/m^3)^2].

d2T_drho2_l Second derivative of temperature with respect to molar density for the liquid phase, [K/(mol/m^3)^2].

d2T_dV2_g Second partial derivative of temperature with respect to volume (constant pressure) for the gas phase, [K*mol^2/m^6].

d2T_dV2_l Second partial derivative of temperature with respect to volume (constant pressure) for the liquid phase, [K*mol^2/m^6].

d2T_dVdP_g Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the gas phase, [K*mol/(Pa*m^3)].

d2T_dVdP_l Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the liquid phase, [K*mol/(Pa*m^3)].

d2V_dP2_g Second partial derivative of volume with respect to pressure (constant temperature) for the gas phase, [m^3/(Pa^2*mol)].

d2V_dP2_l Second partial derivative of volume with respect to pressure (constant temperature) for the liquid phase, [m^3/(Pa^2*mol)].

d2V_dPdT_g Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the gas phase, [m^3/(K*Pa*mol)].

d2V_dPdT_l Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the liquid phase, [m^3/(K*Pa*mol)].

d2V_dT2_g Second partial derivative of volume with respect to temperature (constant pressure) for the gas phase, [m^3/(mol*K^2)].

d2V_dT2_l Second partial derivative of volume with respect to temperature (constant pressure) for the liquid phase, [m^3/(mol*K^2)].

d2V_dTdP_g Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the gas phase, [m^3/(K*Pa*mol)].

d2V_dTdP_l Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the liquid phase, [m^3/(K*Pa*mol)].

d3a_alpha_dT3 Method to calculate the third temperature derivative of $a\alpha$, [J^2/mol^2/Pa/K^3].

da_alpha_dP_g_V Derivative of the *a_alpha* with respect to pressure at constant volume (varying T) for the gas phase, [J^2/mol^2/Pa^2].

da_alpha_dP_l_V Derivative of the *a_alpha* with respect to pressure at constant volume (varying T) for the liquid phase, [J^2/mol^2/Pa^2].

dbeta_dP_g Derivative of isobaric expansion coefficient with respect to pressure for the gas phase, [1/(Pa*K)].

dbeta_dP_l Derivative of isobaric expansion coefficient with respect to pressure for the liquid phase, [1/(Pa*K)].

dbeta_dT_g Derivative of isobaric expansion coefficient with respect to temperature for the gas phase, [1/K^2].

dbeta_dT_l Derivative of isobaric expansion coefficient with respect to temperature for the liquid phase, [1/K^2].

dfugacity_dP_g Derivative of fugacity with respect to pressure for the gas phase, [-].

dfugacity_dP_l Derivative of fugacity with respect to pressure for the liquid phase, [-].

dfugacity_dT_g Derivative of fugacity with respect to temperature for the gas phase, [Pa/K].

dfugacity_dT_l Derivative of fugacity with respect to temperature for the liquid phase, [Pa/K].

dH_dep_dP_g Derivative of departure enthalpy with respect to pressure for the gas phase, [(J/mol)/Pa].

dH_dep_dP_g_V Derivative of departure enthalpy with respect to pressure at constant volume for the liquid phase, [(J/mol)/Pa].

dH_dep_dP_l Derivative of departure enthalpy with respect to pressure for the liquid phase, [(J/mol)/Pa].

dH_dep_dP_l_V Derivative of departure enthalpy with respect to pressure at constant volume for the gas phase, [(J/mol)/Pa].

dH_dep_dT_g Derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K].

dH_dep_dT_g_V Derivative of departure enthalpy with respect to temperature at constant volume for the gas phase, [(J/mol)/K].

dH_dep_dT_l Derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K].

dH_dep_dT_l_V Derivative of departure enthalpy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K].

dH_dep_dV_g_P Derivative of departure enthalpy with respect to volume at constant pressure for the gas phase, [J/m^3].

dH_dep_dV_g_T Derivative of departure enthalpy with respect to volume at constant temperature for the gas phase, [J/m^3].

dH_dep_dV_l_P Derivative of departure enthalpy with respect to volume at constant pressure for the liquid phase, [J/m^3].

dH_dep_dV_l_T Derivative of departure enthalpy with respect to volume at constant temperature for the gas phase, [J/m^3].

dP_drho_g Derivative of pressure with respect to molar density for the gas phase, [Pa/(mol/m^3)].

dP_drho_l Derivative of pressure with respect to molar density for the liquid phase, [Pa/(mol/m^3)].

dphi_dP_g Derivative of fugacity coefficient with respect to pressure for the gas phase, [1/Pa].

dphi_dP_l Derivative of fugacity coefficient with respect to pressure for the liquid phase, [1/Pa].

dphi_dT_g Derivative of fugacity coefficient with respect to temperature for the gas phase, [1/K].

dphi_dT_l Derivative of fugacity coefficient with respect to temperature for the liquid phase, [1/K].

drho_dP_g Derivative of molar density with respect to pressure for the gas phase, [(mol/m^3)/Pa].

drho_dP_l Derivative of molar density with respect to pressure for the liquid phase, [(mol/m^3)/Pa].

drho_dT_g Derivative of molar density with respect to temperature for the gas phase, [(mol/m^3)/K].

drho_dT_l Derivative of molar density with respect to temperature for the liquid phase, [(mol/m^3)/K].

dS_dep_dP_g Derivative of departure entropy with respect to pressure for the gas phase, [(J/mol)/K/Pa].

dS_dep_dP_g_V Derivative of departure entropy with respect to pressure at constant volume for the gas phase, [(J/mol)/K/Pa].

dS_dep_dP_l Derivative of departure entropy with respect to pressure for the liquid phase, [(J/mol)/K/Pa].

dS_dep_dP_l_V Derivative of departure entropy with respect to pressure at constant volume for the liquid phase, [(J/mol)/K/Pa].

dS_dep_dT_g Derivative of departure entropy with respect to temperature for the gas phase, [(J/mol)/K^2].

dS_dep_dT_g_V Derivative of departure entropy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^2].

dS_dep_dT_l Derivative of departure entropy with respect to temperature for the liquid phase, [(J/mol)/K^2].

dS_dep_dT_l_V Derivative of departure entropy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^2].

dS_dep_dV_g_P Derivative of departure entropy with respect to volume at constant pressure for the gas phase, [J/K/m^3].

dS_dep_dV_g_T Derivative of departure entropy with respect to volume at constant temperature for the gas phase, [J/K/m^3].

dS_dep_dV_l_P Derivative of departure entropy with respect to volume at constant pressure for the liquid phase, [J/K/m^3].

dS_dep_dV_l_T Derivative of departure entropy with respect to volume at constant temperature for the gas phase, [J/K/m^3].

dT_drho_g Derivative of temperature with respect to molar density for the gas phase, [K/(mol/m^3)].

dT_drho_l Derivative of temperature with respect to molar density for the liquid phase, [K/(mol/m^3)].

dZ_dP_g Derivative of compressibility factor with respect to pressure for the gas phase, [1/Pa].

dZ_dP_l Derivative of compressibility factor with respect to pressure for the liquid phase, [1/Pa].

dZ_dT_g Derivative of compressibility factor with respect to temperature for the gas phase, [1/K].

dZ_dT_l Derivative of compressibility factor with respect to temperature for the liquid phase, [1/K].

fugacity_g Fugacity for the gas phase, [Pa].

fugacity_l Fugacity for the liquid phase, [Pa].

kappa_g Isothermal (constant-temperature) expansion coefficient for the gas phase, [1/Pa].

kappa_l Isothermal (constant-temperature) expansion coefficient for the liquid phase, [1/Pa].

lnphi_g The natural logarithm of the fugacity coefficient for the gas phase, [-].

lnphi_l The natural logarithm of the fugacity coefficient for the liquid phase, [-].

more_stable_phase Checks the Gibbs energy of each possible phase, and returns 'l' if the liquid-like phase is more stable, and 'g' if the vapor-like phase is more stable.

mpmath_volume_ratios Method to compare, as ratios, the volumes of the implemented cubic solver versus those calculated using *mpmath*.

mpmath_volumes Method to calculate to a high precision the exact roots to the cubic equation, using *mpmath*.

mpmath_volumes_float Method to calculate real roots of a cubic equation, using *mpmath*, but returned as floats.

phi_g Fugacity coefficient for the gas phase, [Pa].

phi_l Fugacity coefficient for the liquid phase, [Pa].

rho_g Gas molar density, [mol/m^3].

rho_l Liquid molar density, [mol/m^3].

sorted_volumes List of lexicographically-sorted molar volumes available from the root finding algorithm used to solve the PT point.

state_specs Convenience method to return the two specified state specs (T, P, or V) as a dictionary.

U_dep_g Departure molar internal energy from ideal gas behavior for the gas phase, [J/mol].

U_dep_l Departure molar internal energy from ideal gas behavior for the liquid phase, [J/mol].

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

V_dep_g Departure molar volume from ideal gas behavior for the gas phase, [m^3/mol].

V_dep_l Departure molar volume from ideal gas behavior for the liquid phase, [m^3/mol].

V_g_mpmath The molar volume of the gas phase calculated with *mpmath* to a higher precision, [m^3/mol].

V_l_mpmath The molar volume of the liquid phase calculated with *mpmath* to a higher precision, [m^3/mol].

Methods

Hvap(T)	Method to calculate enthalpy of vaporization for a pure fluid from an equation of state, without iteration.
PT_surface_special([Tmin, Tmax, Pmin, Pmax, ...])	Method to create a plot of the special curves of a fluid - vapor pressure, determinant zeros, pseudo critical point, and mechanical critical point.

continues on next page

Table 7 – continued from previous page

P_PIP_transition(T[, low_P_limit])	Method to calculate the pressure which makes the phase identification parameter exactly 1.
P_discriminant_zero_g()	Method to calculate the pressure which zero the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a vapor-like volume; and having a vapor-like volume.
P_discriminant_zero_l()	Method to calculate the pressure which zero the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a liquid-like volume; and having a liquid-like volume.
P_discriminant_zeros()	Method to calculate the pressures which zero the discriminant function of the general cubic eos, at the current temperature.
P_discriminant_zeros_analytical(T, b, delta, ...)	Method to calculate the pressures which zero the discriminant function of the general cubic eos.
P_max_at_V(V)	Dummy method.
Psat(T[, polish, guess])	Generic method to calculate vapor pressure for a specified *T*.
Psat_errors([Tmin, Tmax, pts, plot, show, ...])	Method to create a plot of vapor pressure and the relative error of its calculation vs.
T_discriminant_zero_g([T_guess])	Method to calculate the temperature which zeros the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a vapor-like volume; and having a vapor-like volume.
T_discriminant_zero_l([T_guess])	Method to calculate the temperature which zeros the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a liquid-like volume; and having a liquid-like volume.
T_max_at_V(V[, Pmax])	Method to calculate the maximum temperature the EOS can create at a constant volume, if one exists; returns None otherwise.
T_min_at_V(V[, Pmin])	Returns the minimum temperature for the EOS to have the volume as specified.
Tsat(P[, polish])	Generic method to calculate the temperature for a specified vapor pressure of the pure fluid.
V_g_sat(T)	Method to calculate molar volume of the vapor phase along the saturation line.
V_l_sat(T)	Method to calculate molar volume of the liquid phase along the saturation line.
Vs_mpmath()	Method to calculate real roots of a cubic equation, using *mpmath*.
a_alpha_and_derivatives(T[, full, quick, ...])	Method to calculate $a\alpha$ and its first and second derivatives.
a_alpha_and_derivatives_pure(T)	Dummy method to calculate $a\alpha$ and its first and second derivatives.
a_alpha_for_Psat(T, Psat[, a_alpha_guess])	Method to calculate which value of $a\alpha$ is required for a given *T*, *Psat* pair.
a_alpha_for_V(T, P, V)	Method to calculate which value of $a\alpha$ is required for a given *T*, *P* pair to match a specified *V*.
a_alpha_plot([Tmin, Tmax, pts, plot, show])	Method to create a plot of the $a\alpha$ parameter and its first two derivatives.

continues on next page

Table 7 – continued from previous page

as_json()	Method to create a JSON-friendly serialization of the eos which can be stored, and reloaded later.
check_sufficient_inputs()	Method to an exception if none of the pairs (T, P), (T, V), or (P, V) are given.
d2phi_sat_dT2(T[, polish])	Method to calculate the second temperature derivative of saturation fugacity coefficient of the compound.
dH_dep_dT_sat_g(T[, polish])	Method to calculate and return the temperature derivative of saturation vapor excess enthalpy.
dH_dep_dT_sat_l(T[, polish])	Method to calculate and return the temperature derivative of saturation liquid excess enthalpy.
dPsat_dT(T[, polish, also_Psat])	Generic method to calculate the temperature derivative of vapor pressure for a specified *T*.
dS_dep_dT_sat_g(T[, polish])	Method to calculate and return the temperature derivative of saturation vapor excess entropy.
dS_dep_dT_sat_l(T[, polish])	Method to calculate and return the temperature derivative of saturation liquid excess entropy.
discriminant([T, P])	Method to compute the discriminant of the cubic volume solution with the current EOS parameters, optionally at the same (assumed) *T*, and *P* or at different ones, if values are specified.
dphi_sat_dT(T[, polish])	Method to calculate the temperature derivative of saturation fugacity coefficient of the compound.
from_json(json_repr)	Method to create a eos from a JSON serialization of another eos.
model_hash()	Basic method to calculate a hash of the non-state parts of the model This is useful for comparing to models to determine if they are the same, i.e. in a VLL flash it is important to know if both liquids have the same model.
phi_sat(T[, polish])	Method to calculate the saturation fugacity coefficient of the compound.
resolve_full_alphas()	Generic method to resolve the eos with fully calculated alpha derviatives.
saturation_prop_plot(prop[, Tmin, Tmax, ...])	Method to create a plot of a specified property of the EOS along the (pure component) saturation line.
set_from_PT(Vs[, only_l, only_g])	Counts the number of real volumes in *Vs*, and determines what to do.
set_properties_from_solution(T, P, V, b, ...)	Sets all interesting properties which can be calculated from an EOS alone.
solve([pure_a_alphas, only_l, only_g, ...])	First EOS-generic method; should be called by all specific EOSs.
solve_T(P, V[, solution])	Generic method to calculate *T* from a specified *P* and *V*.
solve_missing_volumes()	Generic method to ensure both volumes, if solutions are physical, have calculated properties.
state_hash()	Basic method to calculate a hash of the state of the model and its model parameters.
to([T, P, V])	Method to construct a new EOS object at two of *T*, *P* or *V*.

continues on next page

Table 7 – continued from previous page

to_PV(P, V)	Method to construct a new EOS object at the spcified P and V.
to_TP(T, P)	Method to construct a new EOS object at the spcified T and P.
to_TV(T, V)	Method to construct a new EOS object at the spcified T and V.
volume_error()	Method to calculate the relative absolute error in the calculated molar volumes.
volume_errors([Tmin, Tmax, Pmin, Pmax, pts, ...])	Method to create a plot of the relative absolute error in the cubic volume solution as compared to a higher-precision calculation.
volume_solutions(T, P, b, delta, epsilon, ...)	Halley's method based solver for cubic EOS volumes based on the idea of initializing from a single liquid-like guess which is solved precisely, deflating the cubic analytically, solving the quadratic equation for the next two volumes, and then performing two halley steps on each of them to obtain the final solutions.
volume_solutions_full(T, P, b, delta, ...[, ...])	Newton-Raphson based solver for cubic EOS volumes based on the idea of initializing from an analytical solver.
volume_solutions_mp(T, P, b, delta, epsilon, ...)	Solution of this form of the cubic EOS in terms of volumes, using the *mpmath* arbitrary precision library.

property A_dep_g

Departure molar Helmholtz energy from ideal gas behavior for the gas phase, [J/mol].

$$A_{dep} = U_{dep} - TS_{dep}$$

property A_dep_l

Departure molar Helmholtz energy from ideal gas behavior for the liquid phase, [J/mol].

$$A_{dep} = U_{dep} - TS_{dep}$$

property Cp_minus_Cv_g

Cp - Cv for the gas phase, [J/mol/K].

$$C_p - C_v = -T\left(\frac{\partial P}{\partial T}\right)_V^2 / \left(\frac{\partial P}{\partial V}\right)_T$$

property Cp_minus_Cv_l

Cp - Cv for the liquid phase, [J/mol/K].

$$C_p - C_v = -T\left(\frac{\partial P}{\partial T}\right)_V^2 / \left(\frac{\partial P}{\partial V}\right)_T$$

Hvap(T)

Method to calculate enthalpy of vaporization for a pure fluid from an equation of state, without iteration.

$$\frac{dP^{sat}}{dT} = \frac{\Delta H_{vap}}{T(V_g - V_l)}$$

Results above the critical temperature are meaningless. A first-order polynomial is used to extrapolate under 0.32 Tc; however, there is normally not a volume solution to the EOS which can produce that low of a pressure.

Parameters

T [float] Temperature, [K]

Returns

Hvap [float] Increase in enthalpy needed for vaporization of liquid phase along the saturation line, [J/mol]

Notes

Calculates vapor pressure and its derivative with *Psat* and *dPsat_dT* as well as molar volumes of the saturation liquid and vapor phase in the process.

Very near the critical point this provides unrealistic results due to *Psat*'s polynomials being insufficiently accurate.

References

[1]

`N = 1`
The number of components in the EOS

`PT_surface_special`(*Tmin=0.0001*, *Tmax=10000.0*, *Pmin=0.01*, *Pmax=1000000000.0*, *pts=50*, *show=False*, *color_map=None*, *mechanical=True*, *pseudo_critical=True*, *Psat=True*, *determinant_zeros=True*, *phase_ID_transition=True*, *base_property='V'*, *base_min=None*, *base_max=None*, *base_selection='Gmin'*)
Method to create a plot of the special curves of a fluid - vapor pressure, determinant zeros, pseudo critical point, and mechanical critical point.

The color background is a plot of the molar volume (by default) which has the minimum Gibbs energy (by default). If shown with a sufficient number of points, the curve between vapor and liquid should be shown smoothly.

Parameters

Tmin [float, optional] Minimum temperature of calculation, [K]

Tmax [float, optional] Maximum temperature of calculation, [K]

Pmin [float, optional] Minimum pressure of calculation, [Pa]

Pmax [float, optional] Maximum pressure of calculation, [Pa]

pts [int, optional] The number of points to include in both the x and y axis [-]

show [bool, optional] Whether or not the plot should be rendered and shown; a handle to it is returned if *plot* is True for other purposes such as saving the plot to a file, [-]

color_map [matplotlib.cm.ListedColormap, optional] Matplotlib colormap object, [-]

mechanical [bool, optional] Whether or not to include the mechanical critical point; this is the same as the critical point for a pure compound but not for a mixture, [-]

pseudo_critical [bool, optional] Whether or not to include the pseudo critical point; this is the same as the critical point for a pure compound but not for a mixture, [-]

Psat [bool, optional] Whether or not to include the vapor pressure curve; for mixtures this is neither the bubble nor dew curve, but rather a hypothetical one which uses the same equation as the pure components, [-]

determinant_zeros [bool, optional] Whether or not to include a curve showing when the EOS's determinant hits zero, [-]

phase_ID_transition [bool, optional] Whether or not to show a curve of where the PIP hits 1 exactly, [-]

base_property [str, optional] The property which should be plotted; '_l' and '_g' are added automatically according to the selected phase, [-]

base_min [float, optional] If specified, the *base* property will values will be limited to this value at the minimum, [-]

base_max [float, optional] If specified, the *base* property will values will be limited to this value at the maximum, [-]

base_selection [str, optional] For the base property, there are often two possible phases and but only one value can be plotted; use 'l' to pefer liquid-like values, 'g' to prefer gas-like values, and 'Gmin' to prefer values of the phase with the lowest Gibbs energy, [-]

Returns

fig [matplotlib.figure.Figure] Plotted figure, only returned if *plot* is True, [-]

P_PIP_transition(*T*, *low_P_limit=0.0*)

Method to calculate the pressure which makes the phase identification parameter exactly 1. There are three regions for this calculation:

- subcritical - PIP = 1 for the gas-like phase at P = 0
- initially supercritical - PIP = 1 on a curve starting at the critical point, increasing for a while, decreasing for a while, and then curving sharply back to a zero pressure.
- later supercritical - PIP = 1 for the liquid-like phase at P = 0

Parameters

T [float] Temperature for the calculation, [K]

low_P_limit [float] What value to return for the subcritical and later region, [Pa]

Returns

P [float] Pressure which makes the PIP = 1, [Pa]

Notes

The transition between the region where this function returns values and the high temperature region that doesn't is the Joule-Thomson inversion point at a pressure of zero and can be directly solved for.

Examples

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=299.,␣
↪P=1E6)
>>> eos.P_PIP_transition(100)
0.0
>>> low_T = eos.to(T=100.0, P=eos.P_PIP_transition(100, low_P_limit=1e-5))
>>> low_T.PIP_l, low_T.PIP_g
(45.778088191, 0.9999999997903)
>>> initial_super = eos.to(T=600.0, P=eos.P_PIP_transition(600))
```

(continues on next page)

(continued from previous page)

```
>>> initial_super.P, initial_super.PIP_g
(6456282.17132, 0.999999999999)
>>> high_T = eos.to(T=900.0, P=eos.P_PIP_transition(900, low_P_limit=1e-5))
>>> high_T.P, high_T.PIP_g
(12536704.763, 0.9999999999)
```

P_discriminant_zero_g()

Method to calculate the pressure which zero the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a vapor-like volume; and having a vapor-like volume.

Returns

P_discriminant_zero_g [float] Pressure which make the discriminants zero at the right condition, [Pa]

Examples

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=299.,␣
↪P=1E6)
>>> P_trans = eos.P_discriminant_zero_g()
>>> P_trans
149960391.7
```

In this case, the discriminant transition does not reveal a transition to two roots being available, only negative roots becoming negative and imaginary.

```
>>> eos.to(T=eos.T, P=P_trans*.99999999).mpmath_volumes_float
((-0.0001037013146195082-1.5043987866732543e-08j), (-0.0001037013146195082+1.
↪5043987866732543e-08j), (0.00011799201928619508+0j))
>>> eos.to(T=eos.T, P=P_trans*1.0000001).mpmath_volumes_float
((-0.00010374888853182635+0j), (-0.00010365374200380354+0j), (0.
↪00011799201875924273+0j))
```

P_discriminant_zero_l()

Method to calculate the pressure which zero the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a liquid-like volume; and having a liquid-like volume.

Returns

P_discriminant_zero_l [float] Pressure which make the discriminants zero at the right condition, [Pa]

Examples

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=299.,␣
↪P=1E6)
>>> P_trans = eos.P_discriminant_zero_l()
>>> P_trans
478346.37289
```

In this case, the discriminant transition shows the change in roots:

```
>>> eos.to(T=eos.T, P=P_trans*.99999999).mpmath_volumes_float
((0.000131179941401770 62+0j), (0.002479717165903531+0j), (0.
↪002480236178570793+0j))
>>> eos.to(T=eos.T, P=P_trans*1.0000001).mpmath_volumes_float
((0.0001311799413872173+0j), (0.002479976386402769-8.206310112063695e-07j), (0.
↪002479976386402769+8.206310112063695e-07j))
```

P_discriminant_zeros()

Method to calculate the pressures which zero the discriminant function of the general cubic eos, at the current temperature.

Returns

P_discriminant_zeros [list[float]] Pressures which make the discriminants zero, [Pa]

Examples

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=299.,␣
↪P=1E6)
>>> eos.P_discriminant_zeros()
[478346.3, 149960391.7]
```

static **P_discriminant_zeros_analytical**(*T*, *b*, *delta*, *epsilon*, *a_alpha*, *valid=False*)

Method to calculate the pressures which zero the discriminant function of the general cubic eos. This is a quartic function solved analytically.

Parameters

T [float] Temperature, [K]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

valid [bool] Whether to filter the calculated pressures so that they are all real, and positive only, [-]

Returns

P_discriminant_zeros [float] Pressures which make the discriminants zero, [Pa]

Notes

Calculated analytically. Derived as follows.

```
>>> from sympy import *
>>> P, T, V, R, b, a, delta, epsilon = symbols('P, T, V, R, b, a, delta, epsilon
↪')
>>> eta = b
>>> B = b*P/(R*T)
>>> deltas = delta*P/(R*T)
>>> thetas = a*P/(R*T)**2
>>> epsilons = epsilon*(P/(R*T))**2
```

(continues on next page)

(continued from previous page)

```
>>> etas = eta*P/(R*T)
>>> a_coeff = 1
>>> b_coeff = (deltas - B - 1)
>>> c = (thetas + epsilons - deltas*(B+1))
>>> d = -(epsilons*(B+1) + thetas*etas)
>>> disc = b_coeff*b_coeff*c*c - 4*a_coeff*c*c*c - 4*b_coeff*b_coeff*b_coeff*d -
↪ 27*a_coeff*a_coeff*d*d + 18*a_coeff*b_coeff*c*d
>>> base = -(expand(disc/P**2*R**3*T**3))
>>> sln = collect(base, P)
```

P_max_at_V(*V*)

Dummy method. The idea behind this method, which is implemented by some subclasses, is to calculate the maximum pressure the EOS can create at a constant volume, if one exists; returns None otherwise. This method, as a dummy method, always returns None.

Parameters

V [float] Constant molar volume, [m^3/mol]

Returns

P [float] Maximum possible isochoric pressure, [Pa]

P_zero_g_cheb_limits = (0.0, 0.0)

P_zero_l_cheb_limits = (0.0, 0.0)

Psat(*T*, *polish=False*, *guess=None*)

Generic method to calculate vapor pressure for a specified *T*.

From Tc to 0.32Tc, uses a 10th order polynomial of the following form:

$$\ln \frac{P_r}{T_r} = \sum_{k=0}^{10} C_k \left(\frac{\alpha}{T_r} - 1\right)^k$$

If *polish* is True, SciPy's *newton* solver is launched with the calculated vapor pressure as an initial guess in an attempt to get more accuracy. This may not converge however.

Results above the critical temperature are meaningless. A first-order polynomial is used to extrapolate under 0.32 Tc; however, there is normally not a volume solution to the EOS which can produce that low of a pressure.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to attempt to use a numerical solver to make the solution more precise or not

Returns

Psat [float] Vapor pressure, [Pa]

Notes

EOSs sharing the same *b*, *delta*, and *epsilon* have the same coefficient sets.

Form for the regression is inspired from [1].

No volume solution is needed when *polish=False*; the only external call is for the value of *a_alpha*.

References

[1]

`Psat_cheb_range = (0.0, 0.0)`

`Psat_errors`(*Tmin=None*, *Tmax=None*, *pts=50*, *plot=False*, *show=False*, *trunc_err_low=1e-18*, *trunc_err_high=1.0*, *Pmin=1e-100*)

Method to create a plot of vapor pressure and the relative error of its calculation vs. the iterative *polish* approach.

Parameters

Tmin [float] Minimum temperature of calculation; if this is too low the saturation routines will stop converging, [K]

Tmax [float] Maximum temperature of calculation; cannot be above the critical temperature, [K]

pts [int, optional] The number of temperature points to include [-]

plot [bool] If False, the solution is returned without plotting the data, [-]

show [bool] Whether or not the plot should be rendered and shown; a handle to it is returned if *plot* is True for other purposes such as saving the plot to a file, [-]

trunc_err_low [float] Minimum plotted error; values under this are rounded to 0, [-]

trunc_err_high [float] Maximum plotted error; values above this are rounded to 1, [-]

Pmin [float] Minimum pressure for the solution to work on, [Pa]

Returns

errors [list[float]] Absolute relative errors, [-]

Psats_num [list[float]] Vapor pressures calculated to full precision, [Pa]

Psats_fit [list[float]] Vapor pressures calculated with the fast solution, [Pa]

fig [matplotlib.figure.Figure] Plotted figure, only returned if *plot* is True, [-]

`T_discriminant_zero_g`(*T_guess=None*)

Method to calculate the temperature which zeros the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a vapor-like volume; and having a vapor-like volume.

Parameters

T_guess [float, optional] Temperature guess, [K]

Returns

T_discriminant_zero_g [float] Temperature which make the discriminants zero at the right condition, [K]

Notes

Significant numerical issues remain in improving this method.

Examples

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=299.,␣
↪P=1E6)
>>> T_trans = eos.T_discriminant_zero_g()
>>> T_trans
644.3023307
```

In this case, the discriminant transition does not reveal a transition to two roots being available, only to there being a double (imaginary) root.

```
>>> eos.to(P=eos.P, T=T_trans).mpmath_volumes_float
((9.309597822372529e-05-0.00015876248805149625j), (9.309597822372529e-05+0.
↪00015876248805149625j), (0.005064847204219234+0j))
```

T_discriminant_zero_l(*T_guess=None*)

Method to calculate the temperature which zeros the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a liquid-like volume; and having a liquid-like volume.

Parameters

T_guess [float, optional] Temperature guess, [K]

Returns

T_discriminant_zero_l [float] Temperature which make the discriminants zero at the right condition, [K]

Notes

Significant numerical issues remain in improving this method.

Examples

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=299.,␣
↪P=1E6)
>>> T_trans = eos.T_discriminant_zero_l()
>>> T_trans
644.3023307
```

In this case, the discriminant transition does not reveal a transition to two roots being available, only to there being a double (imaginary) root.

```
>>> eos.to(P=eos.P, T=T_trans).mpmath_volumes_float
((9.309597822372529e-05-0.00015876248805149625j), (9.309597822372529e-05+0.
↪00015876248805149625j), (0.005064847204219234+0j))
```

T_max_at_V(*V*, *Pmax=None*)

Method to calculate the maximum temperature the EOS can create at a constant volume, if one exists; returns None otherwise.

Parameters

V [float] Constant molar volume, [m^3/mol]

Pmax [float] Maximum possible isochoric pressure, if already known [Pa]

Returns

T [float] Maximum possible temperature, [K]

Examples

```
>>> e = PR(P=1e5, V=0.0001437, Tc=512.5, Pc=8084000.0, omega=0.559)
>>> e.T_max_at_V(e.V)
431155.5
```

T_min_at_V(*V*, *Pmin=1e-15*)

Returns the minimum temperature for the EOS to have the volume as specified. Under this temperature, the pressure will go negative (and the EOS will not solve).

Tsat(*P*, *polish=False*)

Generic method to calculate the temperature for a specified vapor pressure of the pure fluid. This is simply a bounded solver running between *0.2Tc* and *Tc* on the *Psat* method.

Parameters

P [float] Vapor pressure, [Pa]

polish [bool, optional] Whether to attempt to use a numerical solver to make the solution more precise or not

Returns

Tsat [float] Temperature of saturation, [K]

Notes

It is recommended not to run with *polish=True*, as that will make the calculation much slower.

property U_dep_g

Departure molar internal energy from ideal gas behavior for the gas phase, [J/mol].

$$U_{dep} = H_{dep} - PV_{dep}$$

property U_dep_l

Departure molar internal energy from ideal gas behavior for the liquid phase, [J/mol].

$$U_{dep} = H_{dep} - PV_{dep}$$

property V_dep_g

Departure molar volume from ideal gas behavior for the gas phase, [m^3/mol].

$$V_{dep} = V - \frac{RT}{P}$$

property V_dep_l

Departure molar volume from ideal gas behavior for the liquid phase, [m^3/mol].

$$V_{dep} = V - \frac{RT}{P}$$

property V_g_mpmath

The molar volume of the gas phase calculated with *mpmath* to a higher precision, [m^3/mol]. This is useful for validating the cubic root solver(s). It is not quite a true arbitrary solution to the EOS, because the constants *b*,`epsilon`, *delta* and *a_alpha* as well as the input arguments *T* and *P* are not calculated with arbitrary precision. This is a feature when comparing the volume solution algorithms however as they work with the same finite-precision variables.

V_g_sat(*T*)

Method to calculate molar volume of the vapor phase along the saturation line.

Parameters

T [float] Temperature, [K]

Returns

V_g_sat [float] Gas molar volume along the saturation line, [m^3/mol]

Notes

Computes *Psat*, and then uses *volume_solutions* to obtain the three possible molar volumes. The highest value is returned.

property V_l_mpmath

The molar volume of the liquid phase calculated with *mpmath* to a higher precision, [m^3/mol]. This is useful for validating the cubic root solver(s). It is not quite a true arbitrary solution to the EOS, because the constants *b*,`epsilon`, *delta* and *a_alpha* as well as the input arguments *T* and *P* are not calculated with arbitrary precision. This is a feature when comparing the volume solution algorithms however as they work with the same finite-precision variables.

V_l_sat(*T*)

Method to calculate molar volume of the liquid phase along the saturation line.

Parameters

T [float] Temperature, [K]

Returns

V_l_sat [float] Liquid molar volume along the saturation line, [m^3/mol]

Notes

Computes *Psat*, and then uses *volume_solutions* to obtain the three possible molar volumes. The lowest value is returned.

property Vc

Critical volume, [m^3/mol].

$$V_c = \frac{Z_c R T_c}{P_c}$$

Vs_mpmath()

Method to calculate real roots of a cubic equation, using *mpmath*.

Returns

Vs [list[mpf]] Either 1 or 3 real volumes as calculated by *mpmath*, [m^3/mol]

Examples

```
>>> eos = PRTranslatedTwu(T=300, P=1e5, Tc=512.5, Pc=8084000.0, omega=0.559,␣
↪alpha_coeffs=(0.694911, 0.9199, 1.7), c=-1e-6)
>>> eos.Vs_mpmath()
[mpf('0.0000489261705320261435106226558966745'), mpf('0.
↪000541508154451321441068958547812526'), mpf('0.
↪0243149463942697410611501615357228')]
```

__repr__()

Create a string representation of the EOS - by default, include all parameters so as to make it easy to construct new instances from states. Includes the two specified state variables, *Tc*, *Pc*, *omega* and any *kwargs*.

Returns

recreation [str] String which is valid Python and recreates the current state of the object if ran, [-]

Examples

```
>>> eos = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=400.0, P=1e6)
>>> eos
PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=400.0, P=1000000.0)
```

a_alpha_and_derivatives(*T*, *full=True*, *quick=True*, *pure_a_alphas=True*)

Method to calculate $a\alpha$ and its first and second derivatives.

Parameters

T [float] Temperature, [K]

full [bool, optional] If False, calculates and returns only *a_alpha*, [-]

quick [bool, optional] Legary parameter being phased out [-]

pure_a_alphas [bool, optional] Whether or not to recalculate the a_alpha terms of pure components (for the case of mixtures only) which stay the same as the composition changes (i.e in a PT flash); does nothing in the case of pure EOSs [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

a_alpha_and_derivatives_pure(*T*)

Dummy method to calculate $a\alpha$ and its first and second derivatives. Should be implemented with the same function signature in each EOS variant; this only raises a NotImplemented Exception. Should return 'a_alpha', 'da_alpha_dT', and 'd2a_alpha_dT2'.

Parameters

T [float] Temperature, [K]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

a_alpha_for_Psat(*T*, *Psat*, *a_alpha_guess=None*)

Method to calculate which value of $a\alpha$ is required for a given *T*, *Psat* pair. This is a numerical solution, but not a very complicated one.

Parameters

T [float] Temperature, [K]

Psat [float] Vapor pressure specified, [Pa]

a_alpha_guess [float] Optionally, an initial guess for the solver [J^2/mol^2/Pa]

Returns

a_alpha [float] Value calculated to match specified volume for the current EOS, [J^2/mol^2/Pa]

Notes

The implementation of this function is a direct calculation of departure gibbs energy, which is equal in both phases at saturation.

Examples

```
>>> eos = PR(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6)
>>> eos.a_alpha_for_Psat(T=400, Psat=5e5)
3.1565798926
```

a_alpha_for_V(*T*, *P*, *V*)

Method to calculate which value of $a\alpha$ is required for a given *T*, *P* pair to match a specified *V*. This is a straightforward analytical equation.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

Returns

a_alpha [float] Value calculated to match specified volume for the current EOS, [J^2/mol^2/Pa]

Notes

The derivation of the solution is as follows:

```
>>> from sympy import *
>>> P, T, V, R, b, a, delta, epsilon = symbols('P, T, V, R, b, a, delta, epsilon
↪')
>>> a_alpha = symbols('a_alpha')
>>> CUBIC = R*T/(V-b) - a_alpha/(V*V + delta*V + epsilon)
>>> solve(Eq(CUBIC, P), a_alpha)
[(-P*V**3 + P*V**2*b - P*V**2*delta + P*V*b*delta - P*V*epsilon + P*b*epsilon +␣
↪R*T*V**2 + R*T*V*delta + R*T*epsilon)/(V - b)]
```

`a_alpha_plot`(*Tmin=0.0001*, *Tmax=None*, *pts=1000*, *plot=True*, *show=True*)

Method to create a plot of the $a\alpha$ parameter and its first two derivatives. This easily allows identification of EOSs which are displaying inconsistent behavior.

Parameters

Tmin [float] Minimum temperature of calculation, [K]

Tmax [float] Maximum temperature of calculation, [K]

pts [int, optional] The number of temperature points to include [-]

plot [bool] If False, the calculated values and temperatures are returned without plotting the data, [-]

show [bool] Whether or not the plot should be rendered and shown; a handle to it is returned if *plot* is True for other purposes such as saving the plot to a file, [-]

Returns

Ts [list[float]] Logarithmically spaced temperatures in specified range, [K]

a_alpha [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

fig [matplotlib.figure.Figure] Plotted figure, only returned if *plot* is True, [-]

`as_json`()

Method to create a JSON-friendly serialization of the eos which can be stored, and reloaded later.

Returns

json_repr [dict] JSON-friendly representation, [-]

Examples

```
>>> import json
>>> eos = MSRKTranslated(Tc=507.6, Pc=3025000, omega=0.2975, c=22.0561E-6, M=0.
↪7446, N=0.2476, T=250., P=1E6)
>>> assert eos == MSRKTranslated.from_json(json.loads(json.dumps(eos.as_
↪json())))
```

property beta_g

Isobaric (constant-pressure) expansion coefficient for the gas phase, [1/K].

$$\beta = \frac{1}{V}\frac{\partial V}{\partial T}$$

property beta_l

Isobaric (constant-pressure) expansion coefficient for the liquid phase, [1/K].

$$\beta = \frac{1}{V}\frac{\partial V}{\partial T}$$

c1 = None

Parameter used by some equations of state in the *a* calculation

c2 = None

Parameter used by some equations of state in the *b* calculation

check_sufficient_inputs()

Method to an exception if none of the pairs (T, P), (T, V), or (P, V) are given.

property d2H_dep_dT2_g

Second temperature derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K^2].

$$\frac{\partial^2 H_{dep,g}}{\partial T^2} = P\frac{d^2}{dT^2}V(T) - \frac{8T\frac{d}{dT}V(T)\frac{d^2}{dT^2}\operatorname{a\alpha}(T)}{(\delta^2-4\epsilon)\left(\frac{(\delta+2V(T))^2}{\delta^2-4\epsilon}-1\right)} + \frac{2T\operatorname{atanh}\left(\frac{\delta+2V(T)}{\sqrt{\delta^2-4\epsilon}}\right)\frac{d^3}{dT^3}\operatorname{a\alpha}(T)}{\sqrt{\delta^2-4\epsilon}} + \frac{16(\delta+2V(T))\left(T\frac{d}{dT}\operatorname{a\alpha}\cdots\right.}{(\delta^2-4\epsilon)^2\left(\cdots\right.}$$

property d2H_dep_dT2_g_P

Second temperature derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K^2].

$$\frac{\partial^2 H_{dep,g}}{\partial T^2} = P\frac{d^2}{dT^2}V(T) - \frac{8T\frac{d}{dT}V(T)\frac{d^2}{dT^2}\operatorname{a\alpha}(T)}{(\delta^2-4\epsilon)\left(\frac{(\delta+2V(T))^2}{\delta^2-4\epsilon}-1\right)} + \frac{2T\operatorname{atanh}\left(\frac{\delta+2V(T)}{\sqrt{\delta^2-4\epsilon}}\right)\frac{d^3}{dT^3}\operatorname{a\alpha}(T)}{\sqrt{\delta^2-4\epsilon}} + \frac{16(\delta+2V(T))\left(T\frac{d}{dT}\operatorname{a\alpha}\cdots\right.}{(\delta^2-4\epsilon)^2\left(\cdots\right.}$$

property d2H_dep_dT2_g_V

Second temperature derivative of departure enthalpy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^2].

$$\left(\frac{\partial^2 H_{dep,g}}{\partial T^2}\right)_V = \frac{2T\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)\frac{d^3}{dT^3}\operatorname{a\alpha}(T)}{\sqrt{\delta^2-4\epsilon}} + V\frac{\partial^2}{\partial T^2}P(V,T) + \frac{2\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)\frac{d^2}{dT^2}\operatorname{a\alpha}(T)}{\sqrt{\delta^2-4\epsilon}}$$

property d2H_dep_dT2_l

Second temperature derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K^2].

$$\frac{\partial^2 H_{dep,l}}{\partial T^2} = P \frac{d^2}{dT^2} V(T) - \frac{8T \frac{d}{dT} V(T) \frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{(\delta^2 - 4\epsilon)\left(\frac{(\delta+2V(T))^2}{\delta^2-4\epsilon} - 1\right)} + \frac{2T \operatorname{atanh}\left(\frac{\delta+2V(T)}{\sqrt{\delta^2-4\epsilon}}\right) \frac{d^3}{dT^3} \operatorname{a\alpha}(T)}{\sqrt{\delta^2 - 4\epsilon}} + \frac{16(\delta + 2V(T))\left(T \frac{d}{dT} \operatorname{a\alpha}\right.}{(\delta^2 - 4\epsilon)^2 \left(\right.}$$

property d2H_dep_dT2_l_P

Second temperature derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K^2].

$$\frac{\partial^2 H_{dep,l}}{\partial T^2} = P \frac{d^2}{dT^2} V(T) - \frac{8T \frac{d}{dT} V(T) \frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{(\delta^2 - 4\epsilon)\left(\frac{(\delta+2V(T))^2}{\delta^2-4\epsilon} - 1\right)} + \frac{2T \operatorname{atanh}\left(\frac{\delta+2V(T)}{\sqrt{\delta^2-4\epsilon}}\right) \frac{d^3}{dT^3} \operatorname{a\alpha}(T)}{\sqrt{\delta^2 - 4\epsilon}} + \frac{16(\delta + 2V(T))\left(T \frac{d}{dT} \operatorname{a\alpha}\right.}{(\delta^2 - 4\epsilon)^2 \left(\right.}$$

property d2H_dep_dT2_l_V

Second temperature derivative of departure enthalpy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^2].

$$\left(\frac{\partial^2 H_{dep,l}}{\partial T^2}\right)_V = \frac{2T \operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right) \frac{d^3}{dT^3} \operatorname{a\alpha}(T)}{\sqrt{\delta^2 - 4\epsilon}} + V \frac{\partial^2}{\partial T^2} P(V,T) + \frac{2 \operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right) \frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{\sqrt{\delta^2 - 4\epsilon}}$$

property d2H_dep_dTdP_g

Temperature and pressure derivative of departure enthalpy at constant pressure then temperature for the gas phase, [(J/mol)/K/Pa].

$$\left(\frac{\partial^2 H_{dep,g}}{\partial T \partial P}\right)_{T,P} = P \frac{\partial^2}{\partial T \partial P} V(T,P) - \frac{4T \frac{\partial}{\partial P} V(T,P) \frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{(\delta^2 - 4\epsilon)\left(\frac{(\delta+2V(T,P))^2}{\delta^2-4\epsilon} - 1\right)} + \frac{16(\delta + 2V(T,P))\left(T \frac{d}{dT} \operatorname{a\alpha}(T) - \operatorname{a\alpha}(T)\right) \frac{\partial}{\partial P}}{(\delta^2 - 4\epsilon)^2 \left(\frac{(\delta+2V(T,P))^2}{\delta^2-4\epsilon} - 1\right.}$$

property d2H_dep_dTdP_l

Temperature and pressure derivative of departure enthalpy at constant pressure then temperature for the liquid phase, [(J/mol)/K/Pa].

$$\left(\frac{\partial^2 H_{dep,l}}{\partial T \partial P}\right)_V = P \frac{\partial^2}{\partial T \partial P} V(T,P) - \frac{4T \frac{\partial}{\partial P} V(T,P) \frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{(\delta^2 - 4\epsilon)\left(\frac{(\delta+2V(T,P))^2}{\delta^2-4\epsilon} - 1\right)} + \frac{16(\delta + 2V(T,P))\left(T \frac{d}{dT} \operatorname{a\alpha}(T) - \operatorname{a\alpha}(T)\right) \frac{\partial}{\partial P} V}{(\delta^2 - 4\epsilon)^2 \left(\frac{(\delta+2V(T,P))^2}{\delta^2-4\epsilon} - 1\right)}$$

property d2P_dT2_PV_g

Second derivative of pressure with respect to temperature twice, but with pressure held constant the first time and volume held constant the second time for the gas phase, [Pa/K^2].

$$\left(\frac{\partial^2 P}{\partial T \partial T}\right)_{P,V} = -\frac{R \frac{d}{dT} V(T)}{(-b + V(T))^2} - \frac{\left(-\delta \frac{d}{dT} V(T) - 2V(T) \frac{d}{dT} V(T)\right) \frac{d}{dT} \operatorname{a\alpha}(T)}{(\delta V(T) + \epsilon + V^2(T))^2} - \frac{\frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{\delta V(T) + \epsilon + V^2(T)}$$

property d2P_dT2_PV_l

Second derivative of pressure with respect to temperature twice, but with pressure held constant the first time and volume held constant the second time for the liquid phase, [Pa/K^2].

$$\left(\frac{\partial^2 P}{\partial T \partial T}\right)_{P,V} = -\frac{R \frac{d}{dT} V(T)}{(-b + V(T))^2} - \frac{\left(-\delta \frac{d}{dT} V(T) - 2V(T) \frac{d}{dT} V(T)\right) \frac{d}{dT} \operatorname{a\alpha}(T)}{(\delta V(T) + \epsilon + V^2(T))^2} - \frac{\frac{d^2}{dT^2} \operatorname{a\alpha}(T)}{\delta V(T) + \epsilon + V^2(T)}$$

property d2P_dTdP_g

Second derivative of pressure with respect to temperature and, then pressure; and with volume held constant at first, then temperature, for the gas phase, [1/K].

$$\left(\frac{\partial^2 P}{\partial T \partial P}\right)_{V,T} = -\frac{R\frac{d}{dP}V(P)}{(-b+V(P))^2} - \frac{\left(-\delta\frac{d}{dP}V(P) - 2V(P)\frac{d}{dP}V(P)\right)\frac{d}{dT}\operatorname{a\alpha}(T)}{(\delta V(P)+\epsilon+V^2(P))^2}$$

property d2P_dTdP_l

Second derivative of pressure with respect to temperature and, then pressure; and with volume held constant at first, then temperature, for the liquid phase, [1/K].

$$\left(\frac{\partial^2 P}{\partial T \partial P}\right)_{V,T} = -\frac{R\frac{d}{dP}V(P)}{(-b+V(P))^2} - \frac{\left(-\delta\frac{d}{dP}V(P) - 2V(P)\frac{d}{dP}V(P)\right)\frac{d}{dT}\operatorname{a\alpha}(T)}{(\delta V(P)+\epsilon+V^2(P))^2}$$

property d2P_dTdrho_g

Derivative of pressure with respect to molar density, and temperature for the gas phase, [Pa/(K*mol/m^3)].

$$\frac{\partial^2 P}{\partial \rho \partial T} = -V^2\frac{\partial^2 P}{\partial T \partial V}$$

property d2P_dTdrho_l

Derivative of pressure with respect to molar density, and temperature for the liquid phase, [Pa/(K*mol/m^3)].

$$\frac{\partial^2 P}{\partial \rho \partial T} = -V^2\frac{\partial^2 P}{\partial T \partial V}$$

property d2P_dVdP_g

Second derivative of pressure with respect to molar volume and then pressure for the gas phase, [mol/m^3].

$$\frac{\partial^2 P}{\partial V \partial P} = \frac{2RT\frac{d}{dP}V(P)}{(-b+V(P))^3} - \frac{(-\delta-2V(P))\left(-2\delta\frac{d}{dP}V(P) - 4V(P)\frac{d}{dP}V(P)\right)\operatorname{a\alpha}(T)}{(\delta V(P)+\epsilon+V^2(P))^3} + \frac{2\operatorname{a\alpha}(T)\frac{d}{dP}V(P)}{(\delta V(P)+\epsilon+V^2(P))^2}$$

property d2P_dVdP_l

Second derivative of pressure with respect to molar volume and then pressure for the liquid phase, [mol/m^3].

$$\frac{\partial^2 P}{\partial V \partial P} = \frac{2RT\frac{d}{dP}V(P)}{(-b+V(P))^3} - \frac{(-\delta-2V(P))\left(-2\delta\frac{d}{dP}V(P) - 4V(P)\frac{d}{dP}V(P)\right)\operatorname{a\alpha}(T)}{(\delta V(P)+\epsilon+V^2(P))^3} + \frac{2\operatorname{a\alpha}(T)\frac{d}{dP}V(P)}{(\delta V(P)+\epsilon+V^2(P))^2}$$

property d2P_dVdT_TP_g

Second derivative of pressure with respect to molar volume and then temperature at constant temperature then pressure for the gas phase, [Pa*mol/m^3/K].

$$\left(\frac{\partial^2 P}{\partial V \partial T}\right)_{T,P} = \frac{2RT\frac{d}{dT}V(T)}{(-b+V(T))^3} - \frac{R}{(-b+V(T))^2} - \frac{(-\delta-2V(T))\left(-2\delta\frac{d}{dT}V(T) - 4V(T)\frac{d}{dT}V(T)\right)\operatorname{a\alpha}(T)}{(\delta V(T)+\epsilon+V^2(T))^3} - \frac{(-\delta -}{(\delta V(T}$$

property d2P_dVdT_TP_l

Second derivative of pressure with respect to molar volume and then temperature at constant temperature then pressure for the liquid phase, [Pa*mol/m^3/K].

$$\left(\frac{\partial^2 P}{\partial V \partial T}\right)_{T,P} = \frac{2RT\frac{d}{dT}V(T)}{(-b+V(T))^3} - \frac{R}{(-b+V(T))^2} - \frac{(-\delta-2V(T))\left(-2\delta\frac{d}{dT}V(T) - 4V(T)\frac{d}{dT}V(T)\right)\operatorname{a\alpha}(T)}{(\delta V(T)+\epsilon+V^2(T))^3} - \frac{(-\delta -}{(\delta V(T}$$

property d2P_dVdT_g
Alias of `GCEOS.d2P_dTdV_g`

property d2P_dVdT_l
Alias of `GCEOS.d2P_dTdV_l`

property d2P_drho2_g
Second derivative of pressure with respect to molar density for the gas phase, [Pa/(mol/m^3)^2].

$$\frac{\partial^2 P}{\partial \rho^2} = -V^2\left(-V^2\frac{\partial^2 P}{\partial V^2} - 2V\frac{\partial P}{\partial V}\right)$$

property d2P_drho2_l
Second derivative of pressure with respect to molar density for the liquid phase, [Pa/(mol/m^3)^2].

$$\frac{\partial^2 P}{\partial \rho^2} = -V^2\left(-V^2\frac{\partial^2 P}{\partial V^2} - 2V\frac{\partial P}{\partial V}\right)$$

property d2S_dep_dT2_g
Second temperature derivative of departure entropy with respect to temperature for the gas phase, [(J/mol)/K^3].

$$\frac{\partial^2 S_{dep,g}}{\partial T^2} = -\frac{R\left(\frac{d}{dT}V(T) - \frac{V(T)}{T}\right)\frac{d}{dT}V(T)}{V^2(T)} + \frac{R\left(\frac{d^2}{dT^2}V(T) - \frac{2\frac{d}{dT}V(T)}{T} + \frac{2V(T)}{T^2}\right)}{V(T)} - \frac{R\frac{d^2}{dT^2}V(T)}{V(T)} + \frac{R\left(\frac{d}{dT}V(T)\right)^2}{V^2(T)} -$$

property d2S_dep_dT2_g_V
Second temperature derivative of departure entropy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^3].

$$\left(\frac{\partial^2 S_{dep,g}}{\partial T^2}\right)_V = -\frac{R\left(\frac{\partial}{\partial T}P(V,T) - \frac{P(V,T)}{T}\right)\frac{\partial}{\partial T}P(V,T)}{P^2(V,T)} + \frac{R\left(\frac{\partial^2}{\partial T^2}P(V,T) - \frac{2\frac{\partial}{\partial T}P(V,T)}{T} + \frac{2P(V,T)}{T^2}\right)}{P(V,T)} + \frac{R\left(\frac{\partial}{\partial T}P(V,T\right.}{TP(V}$$

property d2S_dep_dT2_l
Second temperature derivative of departure entropy with respect to temperature for the liquid phase, [(J/mol)/K^3].

$$\frac{\partial^2 S_{dep,l}}{\partial T^2} = -\frac{R\left(\frac{d}{dT}V(T) - \frac{V(T)}{T}\right)\frac{d}{dT}V(T)}{V^2(T)} + \frac{R\left(\frac{d^2}{dT^2}V(T) - \frac{2\frac{d}{dT}V(T)}{T} + \frac{2V(T)}{T^2}\right)}{V(T)} - \frac{R\frac{d^2}{dT^2}V(T)}{V(T)} + \frac{R\left(\frac{d}{dT}V(T)\right)^2}{V^2(T)} -$$

property d2S_dep_dT2_l_V
Second temperature derivative of departure entropy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^3].

$$\left(\frac{\partial^2 S_{dep,l}}{\partial T^2}\right)_V = -\frac{R\left(\frac{\partial}{\partial T}P(V,T) - \frac{P(V,T)}{T}\right)\frac{\partial}{\partial T}P(V,T)}{P^2(V,T)} + \frac{R\left(\frac{\partial^2}{\partial T^2}P(V,T) - \frac{2\frac{\partial}{\partial T}P(V,T)}{T} + \frac{2P(V,T)}{T^2}\right)}{P(V,T)} + \frac{R\left(\frac{\partial}{\partial T}P(V,T\right.}{TP(V}$$

property d2S_dep_dTdP_g
Temperature and pressure derivative of departure entropy at constant pressure then temperature for the gas phase, [(J/mol)/K^2/Pa].

$$\left(\frac{\partial^2 S_{dep,g}}{\partial T \partial P}\right)_{T,P} = -\frac{R\frac{\partial^2}{\partial T\partial P}V(T,P)}{V(T,P)} + \frac{R\frac{\partial}{\partial P}V(T,P)\frac{\partial}{\partial T}V(T,P)}{V^2(T,P)} - \frac{R\frac{\partial^2}{\partial T\partial P}V(T,P)}{b - V(T,P)} - \frac{R\frac{\partial}{\partial P}V(T,P)\frac{\partial}{\partial T}V(T,P)}{\left(b - V(T,P)\right)^2} + \frac{16}{}$$

property d2S_dep_dTdP_l

Temperature and pressure derivative of departure entropy at constant pressure then temperature for the liquid phase, [(J/mol)/K^2/Pa].

$$\left(\frac{\partial^2 S_{dep,l}}{\partial T \partial P}\right)_{T,P} = -\frac{R\frac{\partial^2}{\partial T\partial P}V(T,P)}{V(T,P)} + \frac{R\frac{\partial}{\partial P}V(T,P)\frac{\partial}{\partial T}V(T,P)}{V^2(T,P)} - \frac{R\frac{\partial^2}{\partial T\partial P}V(T,P)}{b-V(T,P)} - \frac{R\frac{\partial}{\partial P}V(T,P)\frac{\partial}{\partial T}V(T,P)}{(b-V(T,P))^2} + \frac{16}{}$$

property d2T_dP2_g

Second partial derivative of temperature with respect to pressure (constant volume) for the gas phase, [K/Pa^2].

$$\left(\frac{\partial^2 T}{\partial P^2}\right)_V = -\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial P}{\partial T}\right)^{-3}_V$$

property d2T_dP2_l

Second partial derivative of temperature with respect to pressure (constant temperature) for the liquid phase, [K/Pa^2].

$$\left(\frac{\partial^2 T}{\partial P^2}\right)_V = -\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial P}{\partial T}\right)^{-3}_V$$

property d2T_dPdV_g

Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the gas phase, [K*mol/(Pa*m^3)].

$$\left(\frac{\partial^2 T}{\partial P\partial V}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T\left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]\left(\frac{\partial P}{\partial T}\right)_V^{-3}$$

property d2T_dPdV_l

Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the liquid phase, [K*mol/(Pa*m^3)].

$$\left(\frac{\partial^2 T}{\partial P\partial V}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T\left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]\left(\frac{\partial P}{\partial T}\right)_V^{-3}$$

property d2T_dPdrho_g

Derivative of temperature with respect to molar density, and pressure for the gas phase, [K/(Pa*mol/m^3)].

$$\frac{\partial^2 T}{\partial \rho \partial P} = -V^2\frac{\partial^2 T}{\partial P \partial V}$$

property d2T_dPdrho_l

Derivative of temperature with respect to molar density, and pressure for the liquid phase, [K/(Pa*mol/m^3)].

$$\frac{\partial^2 T}{\partial \rho \partial P} = -V^2\frac{\partial^2 T}{\partial P \partial V}$$

property d2T_dV2_g

Second partial derivative of temperature with respect to volume (constant pressure) for the gas phase, [K*mol^2/m^6].

$$\left(\frac{\partial^2 T}{\partial V^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial V^2}\right)_T\left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T\left(\frac{\partial^2 P}{\partial T \partial V}\right)\right]\left(\frac{\partial P}{\partial T}\right)^{-2}_V + \left[\left(\frac{\partial^2 P}{\partial T\partial V}\right)\left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T\left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]$$

property d2T_dV2_l

Second partial derivative of temperature with respect to volume (constant pressure) for the liquid phase, [K*mol^2/m^6].

$$\left(\frac{\partial^2 T}{\partial V^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial V^2}\right)_T \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right] \left(\frac{\partial P}{\partial T}\right)_V^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]$$

property d2T_dVdP_g

Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the gas phase, [K*mol/(Pa*m^3)].

$$\left(\frac{\partial^2 T}{\partial P \partial V}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right] \left(\frac{\partial P}{\partial T}\right)_V^{-3}$$

property d2T_dVdP_l

Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the liquid phase, [K*mol/(Pa*m^3)].

$$\left(\frac{\partial^2 T}{\partial P \partial V}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right] \left(\frac{\partial P}{\partial T}\right)_V^{-3}$$

property d2T_drho2_g

Second derivative of temperature with respect to molar density for the gas phase, [K/(mol/m^3)^2].

$$\frac{\partial^2 T}{\partial \rho^2} = -V^2(-V^2\frac{\partial^2 T}{\partial V^2} - 2V\frac{\partial T}{\partial V})$$

property d2T_drho2_l

Second derivative of temperature with respect to molar density for the liquid phase, [K/(mol/m^3)^2].

$$\frac{\partial^2 T}{\partial \rho^2} = -V^2(-V^2\frac{\partial^2 T}{\partial V^2} - 2V\frac{\partial T}{\partial V})$$

property d2V_dP2_g

Second partial derivative of volume with respect to pressure (constant temperature) for the gas phase, [m^3/(Pa^2*mol)].

$$\left(\frac{\partial^2 V}{\partial P^2}\right)_T = -\left(\frac{\partial^2 P}{\partial V^2}\right)_T \left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

property d2V_dP2_l

Second partial derivative of volume with respect to pressure (constant temperature) for the liquid phase, [m^3/(Pa^2*mol)].

$$\left(\frac{\partial^2 V}{\partial P^2}\right)_T = -\left(\frac{\partial^2 P}{\partial V^2}\right)_T \left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

property d2V_dPdT_g

Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the gas phase, [m^3/(K*Pa*mol)].

$$\left(\frac{\partial^2 V}{\partial T \partial P}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right] \left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

property d2V_dPdT_l

Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the liquid phase, [m^3/(K*Pa*mol)].

$$\left(\frac{\partial^2 V}{\partial T \partial P}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]\left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

property d2V_dT2_g

Second partial derivative of volume with respect to temperature (constant pressure) for the gas phase, [m^3/(mol*K^2)].

$$\left(\frac{\partial^2 V}{\partial T^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right]\left(\frac{\partial P}{\partial V}\right)_T^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]$$

property d2V_dT2_l

Second partial derivative of volume with respect to temperature (constant pressure) for the liquid phase, [m^3/(mol*K^2)].

$$\left(\frac{\partial^2 V}{\partial T^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right]\left(\frac{\partial P}{\partial V}\right)_T^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]$$

property d2V_dTdP_g

Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the gas phase, [m^3/(K*Pa*mol)].

$$\left(\frac{\partial^2 V}{\partial T \partial P}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]\left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

property d2V_dTdP_l

Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the liquid phase, [m^3/(K*Pa*mol)].

$$\left(\frac{\partial^2 V}{\partial T \partial P}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]\left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

property d2a_alpha_dTdP_g_V

Derivative of the temperature derivative of *a_alpha* with respect to pressure at constant volume (varying T) for the gas phase, [J^2/mol^2/Pa^2/K].

$$\left(\frac{\partial \left(\frac{\partial a\alpha}{\partial T}\right)_P}{\partial P}\right)_V = \left(\frac{\partial^2 a\alpha}{\partial T^2}\right)_P \cdot \left(\frac{\partial T}{\partial P}\right)_V$$

property d2a_alpha_dTdP_l_V

Derivative of the temperature derivative of *a_alpha* with respect to pressure at constant volume (varying T) for the liquid phase, [J^2/mol^2/Pa^2/K].

$$\left(\frac{\partial \left(\frac{\partial a\alpha}{\partial T}\right)_P}{\partial P}\right)_V = \left(\frac{\partial^2 a\alpha}{\partial T^2}\right)_P \cdot \left(\frac{\partial T}{\partial P}\right)_V$$

d2phi_sat_dT2(*T*, *polish=True*)

Method to calculate the second temperature derivative of saturation fugacity coefficient of the compound. This does require solving the EOS itself.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

d2phi_sat_dT2 [float] Second temperature derivative of fugacity coefficient along the liquid-vapor saturation line, [1/K^2]

Notes

This is presently a numerical calculation.

`property d2rho_dP2_g`

Second derivative of molar density with respect to pressure for the gas phase, [(mol/m^3)/Pa^2].

$$\frac{\partial^2 \rho}{\partial P^2} = -\frac{\partial^2 V}{\partial P^2}\frac{1}{V^2} + 2\left(\frac{\partial V}{\partial P}\right)^2\frac{1}{V^3}$$

`property d2rho_dP2_l`

Second derivative of molar density with respect to pressure for the liquid phase, [(mol/m^3)/Pa^2].

$$\frac{\partial^2 \rho}{\partial P^2} = -\frac{\partial^2 V}{\partial P^2}\frac{1}{V^2} + 2\left(\frac{\partial V}{\partial P}\right)^2\frac{1}{V^3}$$

`property d2rho_dPdT_g`

Second derivative of molar density with respect to pressure and temperature for the gas phase, [(mol/m^3)/(K*Pa)].

$$\frac{\partial^2 \rho}{\partial T \partial P} = -\frac{\partial^2 V}{\partial T \partial P}\frac{1}{V^2} + 2\left(\frac{\partial V}{\partial T}\right)\left(\frac{\partial V}{\partial P}\right)\frac{1}{V^3}$$

`property d2rho_dPdT_l`

Second derivative of molar density with respect to pressure and temperature for the liquid phase, [(mol/m^3)/(K*Pa)].

$$\frac{\partial^2 \rho}{\partial T \partial P} = -\frac{\partial^2 V}{\partial T \partial P}\frac{1}{V^2} + 2\left(\frac{\partial V}{\partial T}\right)\left(\frac{\partial V}{\partial P}\right)\frac{1}{V^3}$$

`property d2rho_dT2_g`

Second derivative of molar density with respect to temperature for the gas phase, [(mol/m^3)/K^2].

$$\frac{\partial^2 \rho}{\partial T^2} = -\frac{\partial^2 V}{\partial T^2}\frac{1}{V^2} + 2\left(\frac{\partial V}{\partial T}\right)^2\frac{1}{V^3}$$

`property d2rho_dT2_l`

Second derivative of molar density with respect to temperature for the liquid phase, [(mol/m^3)/K^2].

$$\frac{\partial^2 \rho}{\partial T^2} = -\frac{\partial^2 V}{\partial T^2}\frac{1}{V^2} + 2\left(\frac{\partial V}{\partial T}\right)^2\frac{1}{V^3}$$

property d3a_alpha_dT3

Method to calculate the third temperature derivative of $a\alpha$, [J^2/mol^2/Pa/K^3]. This parameter is needed for some higher derivatives that are needed in some flash calculations.

Returns

d3a_alpha_dT3 [float] Third temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^3]

property dH_dep_dP_g

Derivative of departure enthalpy with respect to pressure for the gas phase, [(J/mol)/Pa].

$$\frac{\partial H_{dep,g}}{\partial P} = P\frac{d}{dP}V(P) + V(P) + \frac{4\left(T\frac{d}{dT}\operatorname{a\alpha}(T) - \operatorname{a\alpha}(T)\right)\frac{d}{dP}V(P)}{\left(\delta^2 - 4\epsilon\right)\left(-\frac{(\delta + 2V(P))^2}{\delta^2 - 4\epsilon} + 1\right)}$$

property dH_dep_dP_g_V

Derivative of departure enthalpy with respect to pressure at constant volume for the liquid phase, [(J/mol)/Pa].

$$\left(\frac{\partial H_{dep,g}}{\partial P}\right)_V = -R\left(\frac{\partial T}{\partial P}\right)_V + V + \frac{2\left(T\left(\frac{\partial\left(\frac{\partial a\alpha}{\partial T}\right)_P}{\partial P}\right)_V + \left(\frac{\partial a\alpha}{\partial T}\right)_P\left(\frac{\partial T}{\partial P}\right)_V - \left(\frac{\partial a\alpha}{\partial P}\right)_V\right)\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)}{\sqrt{\delta^2 - 4\epsilon}}$$

property dH_dep_dP_l

Derivative of departure enthalpy with respect to pressure for the liquid phase, [(J/mol)/Pa].

$$\frac{\partial H_{dep,l}}{\partial P} = P\frac{d}{dP}V(P) + V(P) + \frac{4\left(T\frac{d}{dT}\operatorname{a\alpha}(T) - \operatorname{a\alpha}(T)\right)\frac{d}{dP}V(P)}{\left(\delta^2 - 4\epsilon\right)\left(-\frac{(\delta + 2V(P))^2}{\delta^2 - 4\epsilon} + 1\right)}$$

property dH_dep_dP_l_V

Derivative of departure enthalpy with respect to pressure at constant volume for the gas phase, [(J/mol)/Pa].

$$\left(\frac{\partial H_{dep,g}}{\partial P}\right)_V = -R\left(\frac{\partial T}{\partial P}\right)_V + V + \frac{2\left(T\left(\frac{\partial\left(\frac{\partial a\alpha}{\partial T}\right)_P}{\partial P}\right)_V + \left(\frac{\partial a\alpha}{\partial T}\right)_P\left(\frac{\partial T}{\partial P}\right)_V - \left(\frac{\partial a\alpha}{\partial P}\right)_V\right)\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)}{\sqrt{\delta^2 - 4\epsilon}}$$

property dH_dep_dT_g

Derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K].

$$\frac{\partial H_{dep,g}}{\partial T} = P\frac{d}{dT}V(T) - R + \frac{2T}{\sqrt{\delta^2 - 4\epsilon}}\operatorname{atanh}\left(\frac{\delta + 2V(T)}{\sqrt{\delta^2 - 4\epsilon}}\right)\frac{d^2}{dT^2}\operatorname{a\alpha}(T) + \frac{4\left(T\frac{d}{dT}\operatorname{a\alpha}(T) - \operatorname{a\alpha}(T)\right)\frac{d}{dT}V(T)}{\left(\delta^2 - 4\epsilon\right)\left(-\frac{(\delta + 2V(T))^2}{\delta^2 - 4\epsilon} + 1\right)}$$

property dH_dep_dT_g_V

Derivative of departure enthalpy with respect to temperature at constant volume for the gas phase, [(J/mol)/K].

$$\left(\frac{\partial H_{dep,g}}{\partial T}\right)_V = -R + \frac{2T\operatorname{atanh}\left(\frac{2V_g+\delta}{\sqrt{\delta^2-4\epsilon}}\right)\frac{d^2}{dT^2}\operatorname{a}_\alpha(T)}{\sqrt{\delta^2 - 4\epsilon}} + V_g\frac{\partial}{\partial T}P(T, V)$$

property dH_dep_dT_l

Derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K].

$$\frac{\partial H_{dep,l}}{\partial T} = P\frac{d}{dT}V(T) - R + \frac{2T}{\sqrt{\delta^2 - 4\epsilon}}\operatorname{atanh}\left(\frac{\delta + 2V(T)}{\sqrt{\delta^2 - 4\epsilon}}\right)\frac{d^2}{dT^2}\operatorname{a\alpha}(T) + \frac{4\left(T\frac{d}{dT}\operatorname{a\alpha}(T) - \operatorname{a\alpha}(T)\right)\frac{d}{dT}V(T)}{\left(\delta^2 - 4\epsilon\right)\left(-\frac{(\delta + 2V(T))^2}{\delta^2 - 4\epsilon} + 1\right)}$$

property dH_dep_dT_l_V

Derivative of departure enthalpy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K].

$$\left(\frac{\partial H_{dep,l}}{\partial T}\right)_V = -R + \frac{2T \operatorname{atanh}\left(\frac{2V_l+\delta}{\sqrt{\delta^2-4\epsilon}}\right)\frac{d^2}{dT^2}\operatorname{a_{\alpha}}(T)}{\sqrt{\delta^2-4\epsilon}} + V_l\frac{\partial}{\partial T}P(T,V)$$

dH_dep_dT_sat_g(*T*, *polish=False*)

Method to calculate and return the temperature derivative of saturation vapor excess enthalpy.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

dH_dep_dT_sat_g [float] Vapor phase temperature derivative of excess enthalpy along the liquid-vapor saturation line, [J/mol/K]

dH_dep_dT_sat_l(*T*, *polish=False*)

Method to calculate and return the temperature derivative of saturation liquid excess enthalpy.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

dH_dep_dT_sat_l [float] Liquid phase temperature derivative of excess enthalpy along the liquid-vapor saturation line, [J/mol/K]

property dH_dep_dV_g_P

Derivative of departure enthalpy with respect to volume at constant pressure for the gas phase, [J/m^3].

$$\left(\frac{\partial H_{dep,g}}{\partial V}\right)_P = \left(\frac{\partial H_{dep,g}}{\partial T}\right)_P \cdot \left(\frac{\partial T}{\partial V}\right)_P$$

property dH_dep_dV_g_T

Derivative of departure enthalpy with respect to volume at constant temperature for the gas phase, [J/m^3].

$$\left(\frac{\partial H_{dep,g}}{\partial V}\right)_T = \left(\frac{\partial H_{dep,g}}{\partial P}\right)_T \cdot \left(\frac{\partial P}{\partial V}\right)_T$$

property dH_dep_dV_l_P

Derivative of departure enthalpy with respect to volume at constant pressure for the liquid phase, [J/m^3].

$$\left(\frac{\partial H_{dep,l}}{\partial V}\right)_P = \left(\frac{\partial H_{dep,l}}{\partial T}\right)_P \cdot \left(\frac{\partial T}{\partial V}\right)_P$$

property dH_dep_dV_l_T

Derivative of departure enthalpy with respect to volume at constant temperature for the gas phase, [J/m^3].

$$\left(\frac{\partial H_{dep,l}}{\partial V}\right)_T = \left(\frac{\partial H_{dep,l}}{\partial P}\right)_T \cdot \left(\frac{\partial P}{\partial V}\right)_T$$

property dP_drho_g

Derivative of pressure with respect to molar density for the gas phase, [Pa/(mol/m^3)].

$$\frac{\partial P}{\partial \rho} = -V^2 \frac{\partial P}{\partial V}$$

property dP_drho_l

Derivative of pressure with respect to molar density for the liquid phase, [Pa/(mol/m^3)].

$$\frac{\partial P}{\partial \rho} = -V^2 \frac{\partial P}{\partial V}$$

dPsat_dT(*T*, *polish=False*, *also_Psat=False*)

Generic method to calculate the temperature derivative of vapor pressure for a specified *T*. Implements the analytical derivative of the three polynomials described in *Psat*.

As with *Psat*, results above the critical temperature are meaningless. The first-order polynomial which is used to calculate it under 0.32 Tc may not be physicall meaningful, due to there normally not being a volume solution to the EOS which can produce that low of a pressure.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to attempt to use a numerical solver to make the solution more precise or not

also_Psat [bool, optional] Calculating *dPsat_dT* necessarily involves calculating *Psat*; when this is set to True, a second return value is added, whic is the actual *Psat* value.

Returns

dPsat_dT [float] Derivative of vapor pressure with respect to temperature, [Pa/K]

Psat [float, returned if *also_Psat* is *True*] Vapor pressure, [Pa]

Notes

There is a small step change at 0.32 Tc for all EOS due to the two switch between polynomials at that point.

Useful for calculating enthalpy of vaporization with the Clausius Clapeyron Equation. Derived with SymPy's diff and cse.

property dS_dep_dP_g

Derivative of departure entropy with respect to pressure for the gas phase, [(J/mol)/K/Pa].

$$\frac{\partial S_{dep,g}}{\partial P} = -\frac{R\frac{d}{dP}V(P)}{V(P)} + \frac{R\frac{d}{dP}V(P)}{-b+V(P)} + \frac{4\frac{d}{dP}V(P)\frac{d}{dT}\operatorname{a\alpha}(T)}{(\delta^2-4\epsilon)\left(-\frac{(\delta+2V(P))^2}{\delta^2-4\epsilon}+1\right)} + \frac{R^2T}{PV(P)}\left(\frac{P}{RT}\frac{d}{dP}V(P)+\frac{V(P)}{RT}\right)$$

property dS_dep_dP_g_V

Derivative of departure entropy with respect to pressure at constant volume for the gas phase, [(J/mol)/K/Pa].

$$\left(\frac{\partial S_{dep,g}}{\partial P}\right)_V = \frac{2\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)\left(\frac{\partial\left(\frac{\partial a\alpha}{\partial T}\right)_P}{\partial P}\right)_V}{\sqrt{\delta^2-4\epsilon}} + \frac{R^2\left(-\frac{PV\frac{d}{dP}T(P)}{RT^2(P)}+\frac{V}{RT(P)}\right)T(P)}{PV}$$

property dS_dep_dP_l

Derivative of departure entropy with respect to pressure for the liquid phase, [(J/mol)/K/Pa].

$$\frac{\partial S_{dep,l}}{\partial P} = -\frac{R\frac{d}{dP}V(P)}{V(P)} + \frac{R\frac{d}{dP}V(P)}{-b+V(P)} + \frac{4\frac{d}{dP}V(P)\frac{d}{dT}\operatorname{a\alpha}(T)}{\left(\delta^{2}-4\epsilon\right)\left(-\frac{(\delta+2V(P))^{2}}{\delta^{2}-4\epsilon}+1\right)} + \frac{R^{2}T}{PV(P)}\left(\frac{P}{RT}\frac{d}{dP}V(P)+\frac{V(P)}{RT}\right)$$

property dS_dep_dP_l_V

Derivative of departure entropy with respect to pressure at constant volume for the liquid phase, [(J/mol)/K/Pa].

$$\left(\frac{\partial S_{dep,l}}{\partial P}\right)_{V} = \frac{2\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^{2}-4\epsilon}}\right)\left(\frac{\partial\left(\frac{\partial a\alpha}{\partial T}\right)_{P}}{\partial P}\right)_{V}}{\sqrt{\delta^{2}-4\epsilon}} + \frac{R^{2}\left(-\frac{PV\frac{d}{dP}T(P)}{RT^{2}(P)}+\frac{V}{RT(P)}\right)T(P)}{PV}$$

property dS_dep_dT_g

Derivative of departure entropy with respect to temperature for the gas phase, [(J/mol)/K^2].

$$\frac{\partial S_{dep,g}}{\partial T} = -\frac{R\frac{d}{dT}V(T)}{V(T)} + \frac{R\frac{d}{dT}V(T)}{-b+V(T)} + \frac{4\frac{d}{dT}V(T)\frac{d}{dT}\operatorname{a\alpha}(T)}{\left(\delta^{2}-4\epsilon\right)\left(-\frac{(\delta+2V(T))^{2}}{\delta^{2}-4\epsilon}+1\right)} + \frac{2\frac{d^{2}}{dT^{2}}\operatorname{a\alpha}(T)}{\sqrt{\delta^{2}-4\epsilon}}\operatorname{atanh}\left(\frac{\delta+2V(T)}{\sqrt{\delta^{2}-4\epsilon}}\right) + \frac{R^{2}T}{PV(T}$$

property dS_dep_dT_g_V

Derivative of departure entropy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^2].

$$\left(\frac{\partial S_{dep,g}}{\partial T}\right)_{V} = \frac{R^{2}T\left(\frac{V\frac{\partial}{\partial T}P(T,V)}{RT}-\frac{VP(T,V)}{RT^{2}}\right)}{VP(T,V)} + \frac{2\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^{2}-4\epsilon}}\right)\frac{d^{2}}{dT^{2}}\operatorname{a\alpha}(T)}{\sqrt{\delta^{2}-4\epsilon}}$$

property dS_dep_dT_l

Derivative of departure entropy with respect to temperature for the liquid phase, [(J/mol)/K^2].

$$\frac{\partial S_{dep,l}}{\partial T} = -\frac{R\frac{d}{dT}V(T)}{V(T)} + \frac{R\frac{d}{dT}V(T)}{-b+V(T)} + \frac{4\frac{d}{dT}V(T)\frac{d}{dT}\operatorname{a\alpha}(T)}{\left(\delta^{2}-4\epsilon\right)\left(-\frac{(\delta+2V(T))^{2}}{\delta^{2}-4\epsilon}+1\right)} + \frac{2\frac{d^{2}}{dT^{2}}\operatorname{a\alpha}(T)}{\sqrt{\delta^{2}-4\epsilon}}\operatorname{atanh}\left(\frac{\delta+2V(T)}{\sqrt{\delta^{2}-4\epsilon}}\right) + \frac{R^{2}T}{PV(T}$$

property dS_dep_dT_l_V

Derivative of departure entropy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^2].

$$\left(\frac{\partial S_{dep,l}}{\partial T}\right)_{V} = \frac{R^{2}T\left(\frac{V\frac{\partial}{\partial T}P(T,V)}{RT}-\frac{VP(T,V)}{RT^{2}}\right)}{VP(T,V)} + \frac{2\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^{2}-4\epsilon}}\right)\frac{d^{2}}{dT^{2}}\operatorname{a\alpha}(T)}{\sqrt{\delta^{2}-4\epsilon}}$$

dS_dep_dT_sat_g(*T*, *polish=False*)

Method to calculate and return the temperature derivative of saturation vapor excess entropy.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

dS_dep_dT_sat_g [float] Vapor phase temperature derivative of excess entropy along the liquid-vapor saturation line, [J/mol/K^2]

`dS_dep_dT_sat_l`(*T*, *polish=False*)

Method to calculate and return the temperature derivative of saturation liquid excess entropy.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

dS_dep_dT_sat_l [float] Liquid phase temperature derivative of excess entropy along the liquid-vapor saturation line, [J/mol/K^2]

`property dS_dep_dV_g_P`

Derivative of departure entropy with respect to volume at constant pressure for the gas phase, [J/K/m^3].

$$\left(\frac{\partial S_{dep,g}}{\partial V}\right)_P = \left(\frac{\partial S_{dep,g}}{\partial T}\right)_P \cdot \left(\frac{\partial T}{\partial V}\right)_P$$

`property dS_dep_dV_g_T`

Derivative of departure entropy with respect to volume at constant temperature for the gas phase, [J/K/m^3].

$$\left(\frac{\partial S_{dep,g}}{\partial V}\right)_T = \left(\frac{\partial S_{dep,g}}{\partial P}\right)_T \cdot \left(\frac{\partial P}{\partial V}\right)_T$$

`property dS_dep_dV_l_P`

Derivative of departure entropy with respect to volume at constant pressure for the liquid phase, [J/K/m^3].

$$\left(\frac{\partial S_{dep,l}}{\partial V}\right)_P = \left(\frac{\partial S_{dep,l}}{\partial T}\right)_P \cdot \left(\frac{\partial T}{\partial V}\right)_P$$

`property dS_dep_dV_l_T`

Derivative of departure entropy with respect to volume at constant temperature for the gas phase, [J/K/m^3].

$$\left(\frac{\partial S_{dep,l}}{\partial V}\right)_T = \left(\frac{\partial S_{dep,l}}{\partial P}\right)_T \cdot \left(\frac{\partial P}{\partial V}\right)_T$$

`property dT_drho_g`

Derivative of temperature with respect to molar density for the gas phase, [K/(mol/m^3)].

$$\frac{\partial T}{\partial \rho} = V^2 \frac{\partial T}{\partial V}$$

`property dT_drho_l`

Derivative of temperature with respect to molar density for the liquid phase, [K/(mol/m^3)].

$$\frac{\partial T}{\partial \rho} = V^2 \frac{\partial T}{\partial V}$$

`property dZ_dP_g`

Derivative of compressibility factor with respect to pressure for the gas phase, [1/Pa].

$$\frac{\partial Z}{\partial P} = \frac{1}{RT}\left(V - \frac{\partial V}{\partial P}\right)$$

property dZ_dP_l

Derivative of compressibility factor with respect to pressure for the liquid phase, [1/Pa].

$$\frac{\partial Z}{\partial P} = \frac{1}{RT}\left(V - \frac{\partial V}{\partial P}\right)$$

property dZ_dT_g

Derivative of compressibility factor with respect to temperature for the gas phase, [1/K].

$$\frac{\partial Z}{\partial T} = \frac{P}{RT}\left(\frac{\partial V}{\partial T} - \frac{V}{T}\right)$$

property dZ_dT_l

Derivative of compressibility factor with respect to temperature for the liquid phase, [1/K].

$$\frac{\partial Z}{\partial T} = \frac{P}{RT}\left(\frac{\partial V}{\partial T} - \frac{V}{T}\right)$$

property da_alpha_dP_g_V

Derivative of the *a_alpha* with respect to pressure at constant volume (varying T) for the gas phase, [J^2/mol^2/Pa^2].

$$\left(\frac{\partial a\alpha}{\partial P}\right)_V = \left(\frac{\partial a\alpha}{\partial T}\right)_P \cdot \left(\frac{\partial T}{\partial P}\right)_V$$

property da_alpha_dP_l_V

Derivative of the *a_alpha* with respect to pressure at constant volume (varying T) for the liquid phase, [J^2/mol^2/Pa^2].

$$\left(\frac{\partial a\alpha}{\partial P}\right)_V = \left(\frac{\partial a\alpha}{\partial T}\right)_P \cdot \left(\frac{\partial T}{\partial P}\right)_V$$

property dbeta_dP_g

Derivative of isobaric expansion coefficient with respect to pressure for the gas phase, [1/(Pa*K)].

$$\frac{\partial \beta_g}{\partial P} = \frac{\frac{\partial^2}{\partial T \partial P} V{\left(T,P \right)}_g}{V{\left(T,P \right)}_g} - \frac{\frac{\partial}{\partial P} V{\left(T,P \right)}_g \frac{\partial}{\partial T} V{\left(T,P \right)}_g}{V^{2}{\left(T,P \right)}_g}$$

property dbeta_dP_l

Derivative of isobaric expansion coefficient with respect to pressure for the liquid phase, [1/(Pa*K)].

$$\frac{\partial \beta_g}{\partial P} = \frac{\frac{\partial^2}{\partial T \partial P} V{\left(T,P \right)}_l}{V{\left(T,P \right)}_l} - \frac{\frac{\partial}{\partial P} V{\left(T,P \right)}_l \frac{\partial}{\partial T} V{\left(T,P \right)}_l}{V^{2}{\left(T,P \right)}_l}$$

property dbeta_dT_g

Derivative of isobaric expansion coefficient with respect to temperature for the gas phase, [1/K^2].

$$\frac{\partial \beta_g}{\partial T} = \frac{\frac{\partial^2}{\partial T^2} V{\left(T,P \right)}_g}{V{\left(T,P \right)}_g} - \frac{\left(\frac{\partial}{\partial T} V{\left(T,P \right)}_g\right)^{2}}{V^{2}{\left(T,P \right)}_g}$$

property dbeta_dT_l

Derivative of isobaric expansion coefficient with respect to temperature for the liquid phase, [1/K^2].

$$\frac{\partial \beta_l}{\partial T} = \frac{\frac{\partial^2}{\partial T^2} V(T,P)_l}{V(T,P)_l} - \frac{\left(\frac{\partial}{\partial T} V(T,P)_l\right)^2}{V^2(T,P)_l}$$

property dfugacity_dP_g

Derivative of fugacity with respect to pressure for the gas phase, [-].

$$\frac{\partial (\text{fugacity})_g}{\partial P} = \frac{P}{RT}\left(-T\frac{\partial}{\partial P}\operatorname{S_{dep}}(T,P) + \frac{\partial}{\partial P}\operatorname{H_{dep}}(T,P)\right)e^{\frac{1}{RT}\left(-T\operatorname{S_{dep}}(T,P)+\operatorname{H_{dep}}(T,P)\right)} + e^{\frac{1}{RT}\left(-T\operatorname{S_{dep}}(T,P)+\operatorname{H_{dep}}(T\right.}$$

property dfugacity_dP_l

Derivative of fugacity with respect to pressure for the liquid phase, [-].

$$\frac{\partial (\text{fugacity})_l}{\partial P} = \frac{P}{RT}\left(-T\frac{\partial}{\partial P}\operatorname{S_{dep}}(T,P) + \frac{\partial}{\partial P}\operatorname{H_{dep}}(T,P)\right)e^{\frac{1}{RT}\left(-T\operatorname{S_{dep}}(T,P)+\operatorname{H_{dep}}(T,P)\right)} + e^{\frac{1}{RT}\left(-T\operatorname{S_{dep}}(T,P)+\operatorname{H_{dep}}(T\right.}$$

property dfugacity_dT_g

Derivative of fugacity with respect to temperature for the gas phase, [Pa/K].

$$\frac{\partial (\text{fugacity})_g}{\partial T} = P\left(\frac{1}{RT}\left(-T\frac{\partial}{\partial T}\operatorname{S_{dep}}(T,P) - \operatorname{S_{dep}}(T,P) + \frac{\partial}{\partial T}\operatorname{H_{dep}}(T,P)\right) - \frac{1}{RT^2}\left(-T\operatorname{S_{dep}}(T,P) + \operatorname{H_{dep}}(T,P\right.\right.$$

property dfugacity_dT_l

Derivative of fugacity with respect to temperature for the liquid phase, [Pa/K].

$$\frac{\partial (\text{fugacity})_l}{\partial T} = P\left(\frac{1}{RT}\left(-T\frac{\partial}{\partial T}\operatorname{S_{dep}}(T,P) - \operatorname{S_{dep}}(T,P) + \frac{\partial}{\partial T}\operatorname{H_{dep}}(T,P)\right) - \frac{1}{RT^2}\left(-T\operatorname{S_{dep}}(T,P) + \operatorname{H_{dep}}(T,P)\right.\right.$$

discriminant(*T=None*, *P=None*)

Method to compute the discriminant of the cubic volume solution with the current EOS parameters, optionally at the same (assumed) *T*, and *P* or at different ones, if values are specified.

Parameters

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

Returns

discriminant [float] Discriminant, [-]

Notes

This call is quite quick; only $a\alpha$ is needed and if T is the same as the current object than it has already been computed.

The formula is as follows:

$$\text{discriminant} = -\left(-\frac{27P^2\epsilon\left(\frac{Pb}{RT}+1\right)}{R^2T^2} - \frac{27P^2b\operatorname{a\alpha}(T)}{R^3T^3}\right)\left(-\frac{P^2\epsilon\left(\frac{Pb}{RT}+1\right)}{R^2T^2} - \frac{P^2b\operatorname{a\alpha}(T)}{R^3T^3}\right) + \left(-\frac{P^2\epsilon\left(\frac{Pb}{RT}+1\right)}{R^2T^2} - \frac{P}{}\right.$$

The formula is derived as follows:

```
>>> from sympy import *
>>> P, T, R, b = symbols('P, T, R, b')
>>> a_alpha = symbols(r'a\ \alpha', cls=Function)
>>> delta, epsilon = symbols('delta, epsilon')
>>> eta = b
>>> B = b*P/(R*T)
>>> deltas = delta*P/(R*T)
>>> thetas = a_alpha(T)*P/(R*T)**2
>>> epsilons = epsilon*(P/(R*T))**2
>>> etas = eta*P/(R*T)
>>> a = 1
>>> b = (deltas - B - 1)
>>> c = (thetas + epsilons - deltas*(B+1))
>>> d = -(epsilons*(B+1) + thetas*etas)
>>> disc = b*b*c*c - 4*a*c*c*c - 4*b*b*b*d - 27*a*a*d*d + 18*a*b*c*d
```

Examples

```
>>> base = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=500.0, P=1E6)
>>> base.discriminant()
-0.001026390999
>>> base.discriminant(T=400)
 0.0010458828
>>> base.discriminant(T=400, P=1e9)
12584660355.4
```

property dphi_dP_g

Derivative of fugacity coefficient with respect to pressure for the gas phase, [1/Pa].

$$\frac{\partial \phi}{\partial P} = \frac{\left(-T \frac{\partial}{\partial P} \operatorname{S_{dep}}{\left(T,P \right)} + \frac{\partial}{\partial P} \operatorname{H_{dep}}{\left(T,P \right)}\right) e^{\frac{-T \operatorname{S_{dep}}{\left(T,P \right)} + \operatorname{H_{dep}}{\left(T,P \right)}}{RT}}}{RT}$$

property dphi_dP_l

Derivative of fugacity coefficient with respect to pressure for the liquid phase, [1/Pa].

$$\frac{\partial \phi}{\partial P} = \frac{\left(-T \frac{\partial}{\partial P} \operatorname{S_{dep}}{\left(T,P \right)} + \frac{\partial}{\partial P} \operatorname{H_{dep}}{\left(T,P \right)}\right) e^{\frac{-T \operatorname{S_{dep}}{\left(T,P \right)} + \operatorname{H_{dep}}{\left(T,P \right)}}{RT}}}{RT}$$

property dphi_dT_g

Derivative of fugacity coefficient with respect to temperature for the gas phase, [1/K].

$$\frac{\partial \phi}{\partial T} = \left(\frac{-T \frac{\partial}{\partial T} \operatorname{S_{dep}}{\left(T,P \right)} - \operatorname{S_{dep}}{\left(T,P \right)} + \frac{\partial}{\partial T} \operatorname{H_{dep}}{\left(T,P \right)}}{RT} - \frac{-T \operatorname{S_{dep}}{\left(T,P \right)} + \operatorname{H_{dep}}{\left(T,P \right)}}{RT^{2}}\right) e^{\frac{-T \operatorname{S_{dep}}{\left(T,P \right)} + \operatorname{H_{dep}}{\left(T,P \right)}}{RT}}$$

property dphi_dT_l

Derivative of fugacity coefficient with respect to temperature for the liquid phase, [1/K].

$$\frac{\partial \phi}{\partial T} = \left(\frac{-T \frac{\partial}{\partial T} \operatorname{S_{dep}}{\left(T,P \right)} - \operatorname{S_{dep}}{\left(T,P \right)} + \frac{\partial}{\partial T} \operatorname{H_{dep}}{\left(T,P \right)}}{RT} - \frac{-T \operatorname{S_{dep}}{\left(T,P \right)} + \operatorname{H_{dep}}{\left(T,P \right)}}{RT^{2}}\right) e^{\frac{-T \operatorname{S_{dep}}{\left(T,P \right)} + \operatorname{H_{dep}}{\left(T,P \right)}}{RT}}$$

dphi_sat_dT(*T*, *polish=True*)

Method to calculate the temperature derivative of saturation fugacity coefficient of the compound. This does require solving the EOS itself.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

dphi_sat_dT [float] First temperature derivative of fugacity coefficient along the liquid-vapor saturation line, [1/K]

property drho_dP_g

Derivative of molar density with respect to pressure for the gas phase, [(mol/m^3)/Pa].

$$\frac{\partial \rho}{\partial P} = \frac{-1}{V^2}\frac{\partial V}{\partial P}$$

property drho_dP_l

Derivative of molar density with respect to pressure for the liquid phase, [(mol/m^3)/Pa].

$$\frac{\partial \rho}{\partial P} = \frac{-1}{V^2}\frac{\partial V}{\partial P}$$

property drho_dT_g

Derivative of molar density with respect to temperature for the gas phase, [(mol/m^3)/K].

$$\frac{\partial \rho}{\partial T} = -\frac{1}{V^2}\frac{\partial V}{\partial T}$$

property drho_dT_l

Derivative of molar density with respect to temperature for the liquid phase, [(mol/m^3)/K].

$$\frac{\partial \rho}{\partial T} = -\frac{1}{V^2}\frac{\partial V}{\partial T}$$

classmethod from_json(*json_repr*)

Method to create a eos from a JSON serialization of another eos.

Parameters

json_repr [dict] JSON-friendly representation, [-]

Returns

eos [`GCEOS`] Newly created object from the json serialization, [-]

Notes

It is important that the input string be in the same format as that created by `GCEOS.as_json`.

Examples

```
>>> eos = MSRKTranslated(Tc=507.6, Pc=3025000, omega=0.2975, c=22.0561E-6, M=0.
↪7446, N=0.2476, T=250., P=1E6)
>>> string = eos.as_json()
>>> new_eos = GCEOS.from_json(string)
>>> assert eos.__dict__ == new_eos.__dict__
```

property fugacity_g
Fugacity for the gas phase, [Pa].

$$\text{fugacity} = P \exp\left(\frac{G_{dep}}{RT}\right)$$

property fugacity_l
Fugacity for the liquid phase, [Pa].

$$\text{fugacity} = P \exp\left(\frac{G_{dep}}{RT}\right)$$

property kappa_g
Isothermal (constant-temperature) expansion coefficient for the gas phase, [1/Pa].

$$\kappa = \frac{-1}{V}\frac{\partial V}{\partial P}$$

property kappa_l
Isothermal (constant-temperature) expansion coefficient for the liquid phase, [1/Pa].

$$\kappa = \frac{-1}{V}\frac{\partial V}{\partial P}$$

kwargs = {}
Dictionary which holds input parameters to an EOS which are non-standard; this excludes *T*, *P*, *V*, *omega*, *Tc*, *Pc*, *Vc* but includes EOS specific parameters like *S1* and *alpha_coeffs*.

kwargs_keys = ()

property lnphi_g
The natural logarithm of the fugacity coefficient for the gas phase, [-].

property lnphi_l
The natural logarithm of the fugacity coefficient for the liquid phase, [-].

model_hash()
Basic method to calculate a hash of the non-state parts of the model This is useful for comparing to models to determine if they are the same, i.e. in a VLL flash it is important to know if both liquids have the same model.

Note that the hashes should only be compared on the same system running in the same process!

Returns

model_hash [int] Hash of the object's model parameters, [-]

property more_stable_phase
Checks the Gibbs energy of each possible phase, and returns 'l' if the liquid-like phase is more stable, and 'g' if the vapor-like phase is more stable.

Examples

```
>>> PR(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6).more_stable_phase
'l'
```

property mpmath_volume_ratios

Method to compare, as ratios, the volumes of the implemented cubic solver versus those calculated using *mpmath*.

Returns

ratios [list[mpc]] Either 1 or 3 volume ratios as calculated by *mpmath*, [-]

Examples

```
>>> eos = PRTranslatedTwu(T=300, P=1e5, Tc=512.5, Pc=8084000.0, omega=0.559,␣
↪alpha_coeffs=(0.694911, 0.9199, 1.7), c=-1e-6)
>>> eos.mpmath_volume_ratios
(mpc(real='0.99999999999999995', imag='0.0'), mpc(real='0.999999999999999965',␣
↪imag='0.0'), mpc(real='1.00000000000000005', imag='0.0'))
```

property mpmath_volumes

Method to calculate to a high precision the exact roots to the cubic equation, using *mpmath*.

Returns

Vs [tuple[mpf]] 3 Real or not real volumes as calculated by *mpmath*, [m^3/mol]

Examples

```
>>> eos = PRTranslatedTwu(T=300, P=1e5, Tc=512.5, Pc=8084000.0, omega=0.559,␣
↪alpha_coeffs=(0.694911, 0.9199, 1.7), c=-1e-6)
>>> eos.mpmath_volumes
(mpf('0.0000489261705320261435106226558966745'), mpf('0.
↪000541508154451321441068958547812526'), mpf('0.
↪0243149463942697410611501615357228'))
```

property mpmath_volumes_float

Method to calculate real roots of a cubic equation, using *mpmath*, but returned as floats.

Returns

Vs [list[float]] All volumes calculated by *mpmath*, [m^3/mol]

Examples

```
>>> eos = PRTranslatedTwu(T=300, P=1e5, Tc=512.5, Pc=8084000.0, omega=0.559,␣
↪alpha_coeffs=(0.694911, 0.9199, 1.7), c=-1e-6)
>>> eos.mpmath_volumes_float
((4.892617053202614e-05+0j), (0.0005415081544513214+0j), (0.
↪024314946394269742+0j))
```

multicomponent = False

Whether or not the EOS is multicomponent or not

nonstate_constants = ('Tc', 'Pc', 'omega', 'kwargs', 'a', 'b', 'delta', 'epsilon')

property phi_g

Fugacity coefficient for the gas phase, [Pa].

$$\phi = \frac{\text{fugacity}}{P}$$

property phi_l

Fugacity coefficient for the liquid phase, [Pa].

$$\phi = \frac{\text{fugacity}}{P}$$

phi_sat(*T*, *polish=True*)

Method to calculate the saturation fugacity coefficient of the compound. This does not require solving the EOS itself.

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to perform a rigorous calculation or to use a polynomial fit, [-]

Returns

phi_sat [float] Fugacity coefficient along the liquid-vapor saturation line, [-]

Notes

Accuracy is generally around 1e-7. If Tr is under 0.32, the rigorous method is always used, but a solution may not exist if both phases cannot coexist. If Tr is above 1, likewise a solution does not exist.

resolve_full_alphas()

Generic method to resolve the eos with fully calculated alpha derviatives. Re-calculates properties with the new alpha derivatives for any previously solved roots.

property rho_g

Gas molar density, [mol/m^3].

$$\rho_g = \frac{1}{V_g}$$

property rho_l

Liquid molar density, [mol/m^3].

$$\rho_l = \frac{1}{V_l}$$

saturation_prop_plot(*prop*, *Tmin=None*, *Tmax=None*, *pts=100*, *plot=False*, *show=False*, *both=False*)

Method to create a plot of a specified property of the EOS along the (pure component) saturation line.

Parameters

prop [str] Property to be used; such as 'H_dep_l' (when *both* is False) or 'H_dep' (when *both* is True), [-]

Tmin [float] Minimum temperature of calculation; if this is too low the saturation routines will stop converging, [K]

Tmax [float] Maximum temperature of calculation; cannot be above the critical temperature, [K]

pts [int, optional] The number of temperature points to include [-]

plot [bool] If False, the calculated values and temperatures are returned without plotting the data, [-]

show [bool] Whether or not the plot should be rendered and shown; a handle to it is returned if *plot* is True for other purposes such as saving the plot to a file, [-]

both [bool] When true, append '_l' and '_g' and draw both the liquid and vapor property specified and return two different sets of values.

Returns

Ts [list[float]] Logarithmically spaced temperatures in specified range, [K]

props [list[float]] The property specified if *both* is False; otherwise, the liquid properties, [various]

props_g [list[float]] The gas properties, only returned if *both* is True, [various]

fig [matplotlib.figure.Figure] Plotted figure, only returned if *plot* is True, [-]

`scalar = True`

`set_from_PT`(*Vs*, *only_l=False*, *only_g=False*)

Counts the number of real volumes in *Vs*, and determines what to do. If there is only one real volume, the method *set_properties_from_solution* is called with it. If there are two real volumes, *set_properties_from_solution* is called once with each volume. The phase is returned by *set_properties_from_solution*, and the volumes is set to either *V_l* or *V_g* as appropriate.

Parameters

Vs [list[float]] Three possible molar volumes, [m^3/mol]

only_l [bool] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set.

only_g [bool] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set.

Notes

An optimization attempt was made to remove min() and max() from this function; that is indeed possible, but the check for handling if there are two or three roots makes it not worth it.

`set_properties_from_solution`(*T*, *P*, *V*, *b*, *delta*, *epsilon*, *a_alpha*, *da_alpha_dT*, *d2a_alpha_dT2*, *quick=True*, *force_l=False*, *force_g=False*)

Sets all interesting properties which can be calculated from an EOS alone. Determines which phase the fluid is on its own; for details, see *phase_identification_parameter*.

The list of properties set is as follows, with all properties suffixed with '_l' or '_g'.

dP_dT, dP_dV, dV_dT, dV_dP, dT_dV, dT_dP, d2P_dT2, d2P_dV2, d2V_dT2, d2V_dP2, d2T_dV2, d2T_dP2, d2V_dPdT, d2P_dTdV, d2T_dPdV, H_dep, S_dep, G_dep and PIP.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

quick [bool, optional] Whether to use a SymPy cse-derived expression (3x faster) or individual formulas

Returns

phase [str] Either 'l' or 'g'

Notes

The individual formulas for the derivatives and excess properties are as follows. For definitions of *beta*, see *isobaric_expansion*; for *kappa*, see isothermal_compressibility; for *Cp_minus_Cv*, see *Cp_minus_Cv*; for *phase_identification_parameter*, see *phase_identification_parameter*.

First derivatives; in part using the Triple Product Rule [2], [3]:

$$\left(\frac{\partial P}{\partial T}\right)_V = \frac{R}{V - b} - \frac{a \frac{d\alpha(T)}{dT}}{V^2 + V\delta + \epsilon}$$

$$\left(\frac{\partial P}{\partial V}\right)_T = - \frac{RT}{(V - b)^2} - \frac{a \left(- 2V - \delta\right) \alpha(T)}{\left(V^2 + V\delta + \epsilon\right)^2}$$

$$\left(\frac{\partial V}{\partial T}\right)_P = - \frac{\left(\frac{\partial P}{\partial T}\right)_V}{\left(\frac{\partial P}{\partial V}\right)_T}$$

$$\left(\frac{\partial V}{\partial P}\right)_T = - \frac{\left(\frac{\partial V}{\partial T}\right)_P}{\left(\frac{\partial P}{\partial T}\right)_V}$$

$$\left(\frac{\partial T}{\partial V}\right)_P = \frac{1}{\left(\frac{\partial V}{\partial T}\right)_P}$$

$$\left(\frac{\partial T}{\partial P}\right)_V = \frac{1}{\left(\frac{\partial P}{\partial T}\right)_V}$$

Second derivatives with respect to one variable; those of T and V use identities shown in [1] and verified numerically:

$$\left(\frac{\partial^2 P}{\partial T^2}\right)_V = - \frac{a \frac{d^2\alpha(T)}{dT^2}}{V^2 + V\delta + \epsilon}$$

$$\left(\frac{\partial^2 P}{\partial V^2}\right)_T = 2\left(\frac{RT}{(V - b)^3} - \frac{a\left(2V + \delta\right)^2 \alpha(T)}{\left(V^2 + V\delta + \epsilon\right)^3} + \frac{a\alpha(T)}{\left(V^2 + V\delta + \epsilon\right)^2}\right)$$

Second derivatives with respect to the other two variables; those of T and V use identities shown in [1] and verified numerically:

$$\left(\frac{\partial^2 P}{\partial T \partial V}\right) = -\frac{R}{(V-b)^2} + \frac{a(2V+\delta)\frac{d\alpha(T)}{dT}}{(V^2+V\delta+\epsilon)^2}$$

Excess properties

$$H_{dep} = \int_{\infty}^{V}\left[T\frac{\partial P}{\partial T}_V - P\right]dV + PV - RT = PV - RT + \frac{2}{\sqrt{\delta^2-4\epsilon}}\left(Ta\frac{d\alpha(T)}{dT} - a\alpha(T)\right)\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)$$

$$S_{dep} = \int_{\infty}^{V}\left[\frac{\partial P}{\partial T} - \frac{R}{V}\right]dV + R\ln\frac{PV}{RT} = -R\ln(V) + R\ln\left(\frac{PV}{RT}\right) + R\ln(V-b) + \frac{2a\frac{d\alpha(T)}{dT}}{\sqrt{\delta^2-4\epsilon}}\operatorname{atanh}\left(\frac{2V+\delta}{\sqrt{\delta^2-4\epsilon}}\right)$$

$$G_{dep} = H_{dep} - TS_{dep}$$

$$C_{v,dep} = T\int_{\infty}^{V}\left(\frac{\partial^2 P}{\partial T^2}\right)dV = -Ta\left(\sqrt{\frac{1}{\delta^2-4\epsilon}}\ln\left(V - \frac{\delta^2}{2}\sqrt{\frac{1}{\delta^2-4\epsilon}} + \frac{\delta}{2} + 2\epsilon\sqrt{\frac{1}{\delta^2-4\epsilon}}\right) - \sqrt{\frac{1}{\delta^2-4\epsilon}}\ln\left(V + \frac{\delta^2}{2}\right.\right.$$

$$C_{p,dep} = (C_p - C_v)_{\text{from EOS}} + C_{v,dep} - R$$

References

[1], [2], [3]

solve(*pure_a_alphas=True*, *only_l=False*, *only_g=False*, *full_alphas=True*)

First EOS-generic method; should be called by all specific EOSs. For solving for *T*, the EOS must provide the method *solve_T*. For all cases, the EOS must provide *a_alpha_and_derivatives*. Calls *set_from_PT* once done.

solve_T(*P*, *V*, *solution=None*)

Generic method to calculate *T* from a specified *P* and *V*. Provides SciPy's *newton* solver, and iterates to solve the general equation for *P*, recalculating *a_alpha* as a function of temperature using *a_alpha_and_derivatives* each iteration.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

solve_missing_volumes()

Generic method to ensure both volumes, if solutions are physical, have calculated properties. This effectively un-does the optimization of the *only_l* and *only_g* keywords.

property sorted_volumes

List of lexicographically-sorted molar volumes available from the root finding algorithm used to solve the PT point. The convention of sorting lexicographically comes from numpy's handling of complex numbers, which python does not define. This method was added to facilitate testing, as the volume solution method changes over time and the ordering does as well.

Examples

```
>>> PR(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6).sorted_volumes
((0.000130222125139+0j), (0.00112363131346-0.00129269672343j), (0.
↪00112363131346+0.00129269672343j))
```

state_hash()

Basic method to calculate a hash of the state of the model and its model parameters.

Note that the hashes should only be compared on the same system running in the same process!

Returns

state_hash [int] Hash of the object's model parameters and state, [-]

property state_specs

Convenience method to return the two specified state specs (*T*, *P*, or *V*) as a dictionary.

Examples

```
>>> PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=500.0, V=1.0).state_specs
{'T': 500.0, 'V': 1.0}
```

to(*T=None*, *P=None*, *V=None*)

Method to construct a new EOS object at two of *T*, *P* or *V*. In the event the specs match those of the current object, it will be returned unchanged.

Parameters

T [float or None, optional] Temperature, [K]

P [float or None, optional] Pressure, [Pa]

V [float or None, optional] Molar volume, [m^3/mol]

Returns

obj [EOS] Pure component EOS at the two specified specs, [-]

Notes

Constructs the object with parameters *Tc*, *Pc*, *omega*, and *kwargs*.

Examples

```
>>> base = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=500.0, P=1E6)
>>> base.to(T=300.0, P=1e9).state_specs
{'T': 300.0, 'P': 1000000000.0}
>>> base.to(T=300.0, V=1.0).state_specs
{'T': 300.0, 'V': 1.0}
>>> base.to(P=1e5, V=1.0).state_specs
{'P': 100000.0, 'V': 1.0}
```

to_PV(*P*, *V*)

Method to construct a new EOS object at the spcified *P* and *V*. In the event the *P* and *V* match the current object's *P* and *V*, it will be returned unchanged.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

Returns

obj [EOS] Pure component EOS at specified *P* and *V*, [-]

Notes

Constructs the object with parameters *Tc*, *Pc*, *omega*, and *kwargs*.

Examples

```
>>> base = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=500.0, P=1E6)
>>> new = base.to_PV(P=1000.0, V=1.0)
>>> base.state_specs, new.state_specs
({'T': 500.0, 'P': 1000000.0}, {'P': 1000.0, 'V': 1.0})
```

to_TP(*T*, *P*)

Method to construct a new EOS object at the spcified *T* and *P*. In the event the *T* and *P* match the current object's *T* and *P*, it will be returned unchanged.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

obj [EOS] Pure component EOS at specified *T* and *P*, [-]

Notes

Constructs the object with parameters *Tc*, *Pc*, *omega*, and *kwargs*.

Examples

```
>>> base = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=500.0, P=1E6)
>>> new = base.to_TP(T=1.0, P=2.0)
>>> base.state_specs, new.state_specs
({'T': 500.0, 'P': 1000000.0}, {'T': 1.0, 'P': 2.0})
```

to_TV(*T*, *V*)

Method to construct a new EOS object at the spcified *T* and *V*. In the event the *T* and *V* match the current object's *T* and *V*, it will be returned unchanged.

Parameters

T [float] Temperature, [K]

V [float] Molar volume, [m^3/mol]

Returns

obj [EOS] Pure component EOS at specified *T* and *V*, [-]

Notes

Constructs the object with parameters *Tc*, *Pc*, *omega*, and *kwargs*.

Examples

```
>>> base = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=500.0, P=1E6)
>>> new = base.to_TV(T=1000000.0, V=1.0)
>>> base.state_specs, new.state_specs
({'T': 500.0, 'P': 1000000.0}, {'T': 1000000.0, 'V': 1.0})
```

volume_error()

Method to calculate the relative absolute error in the calculated molar volumes. This is computed with *mpmath*. If the number of real roots is different between mpmath and the implemented solver, an error of 1 is returned.

Parameters

T [float] Temperature, [K]

Returns

error [float] relative absolute error in molar volumes , [-]

Examples

```
>>> eos = PRTranslatedTwu(T=300, P=1e5, Tc=512.5, Pc=8084000.0, omega=0.559, 
↪alpha_coeffs=(0.694911, 0.9199, 1.7), c=-1e-6)
>>> eos.volume_error()
5.2192e-17
```

volume_errors(*Tmin=0.0001*, *Tmax=10000.0*, *Pmin=0.01*, *Pmax=1000000000.0*, *pts=50*, *plot=False*, *show=False*, *trunc_err_low=1e-18*, *trunc_err_high=1.0*, *color_map=None*, *timing=False*)

Method to create a plot of the relative absolute error in the cubic volume solution as compared to a higher-precision calculation. This method is incredible valuable for the development of more reliable floating-point based cubic solutions.

Parameters

Tmin [float] Minimum temperature of calculation, [K]

Tmax [float] Maximum temperature of calculation, [K]

Pmin [float] Minimum pressure of calculation, [Pa]

Pmax [float] Maximum pressure of calculation, [Pa]

pts [int, optional] The number of points to include in both the *x* and *y* axis; the validation calculation is slow, so increasing this too much is not advisable, [-]

plot [bool] If False, the calculated errors are returned without plotting the data, [-]

show [bool] Whether or not the plot should be rendered and shown; a handle to it is returned if *plot* is True for other purposes such as saving the plot to a file, [-]

trunc_err_low [float] Minimum plotted error; values under this are rounded to 0, [-]

trunc_err_high [float] Maximum plotted error; values above this are rounded to 1, [-]

color_map [matplotlib.cm.ListedColormap] Matplotlib colormap object, [-]

timing [bool] If True, plots the time taken by the volume root calculations themselves; this can reveal whether the solvers are taking fast or slow paths quickly, [-]

Returns

errors [list[list[float]]] Relative absolute errors in the volume calculation (or timings in seconds if *timing* is True), [-]

fig [matplotlib.figure.Figure] Plotted figure, only returned if *plot* is True, [-]

`static volume_solutions`(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Halley's method based solver for cubic EOS volumes based on the idea of initializing from a single liquid-like guess which is solved precisely, deflating the cubic analytically, solving the quadratic equation for the next two volumes, and then performing two halley steps on each of them to obtain the final solutions. This method does not calculate imaginary roots - they are set to zero on detection. This method has been rigorously tested over a wide range of conditions.

The method uses the standard combination of bisection to provide high and low boundaries as well, to keep the iteration always moving forward.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [tuple[float]] Three possible molar volumes, [m^3/mol]

Notes

A sample region where this method works perfectly is shown below:

`static volume_solutions_full`(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*, *tries=0*)

Newton-Raphson based solver for cubic EOS volumes based on the idea of initializing from an analytical solver. This algorithm can only be described as a monstrous mess. It is fairly fast for most cases, but about 3x slower than `volume_solutions_halley`. In the worst case this will fall back to *mpmath*.

Parameters

T [float] Temperature, [K]

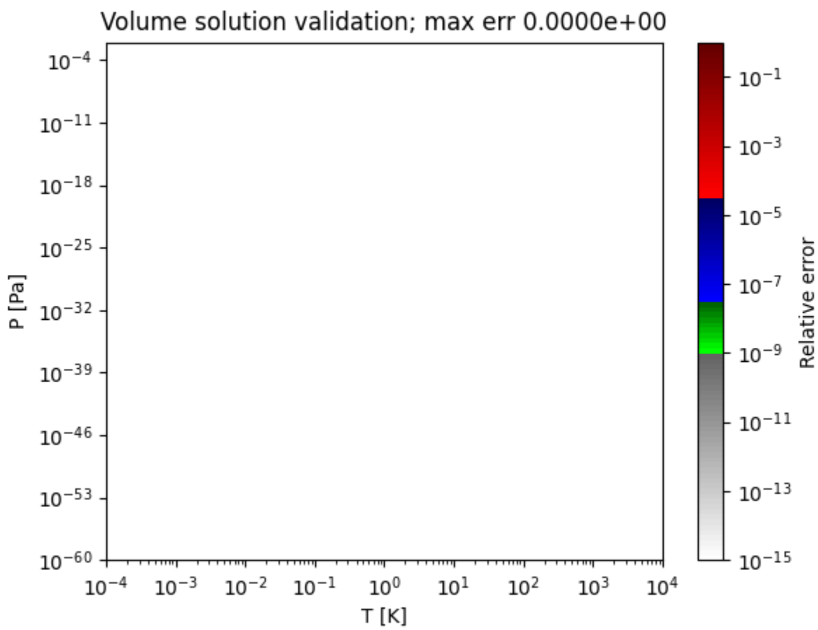
P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]



tries [int, optional] Internal parameter as this function will call itself if it needs to; number of previous solve attempts, [-]

Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Notes

Sample regions where this method works perfectly are shown below:

`static volume_solutions_mp`(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*, *dps=50*)

Solution of this form of the cubic EOS in terms of volumes, using the *mpmath* arbitrary precision library. The number of decimal places returned is controlled by the *dps* parameter.

This function is the reference implementation which provides exactly correct solutions; other algorithms are compared against this one.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

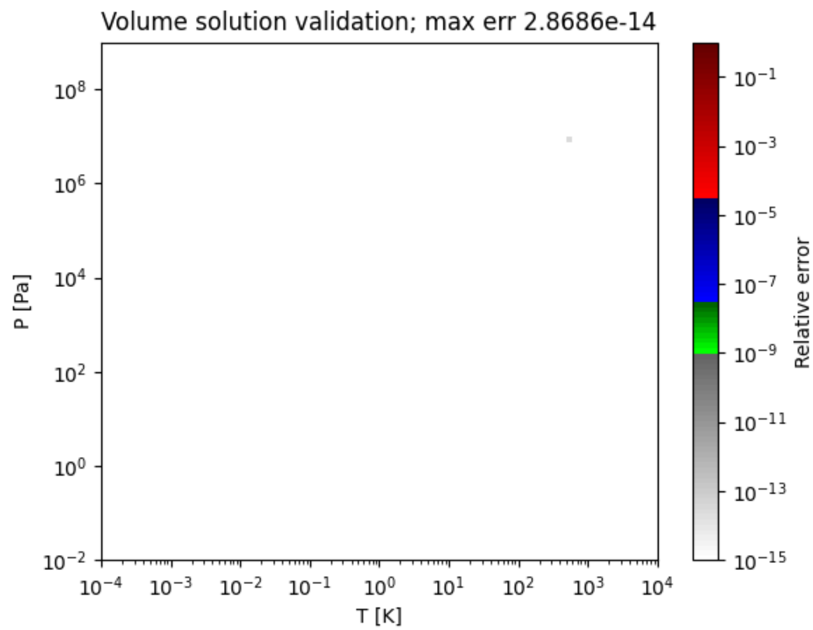
epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

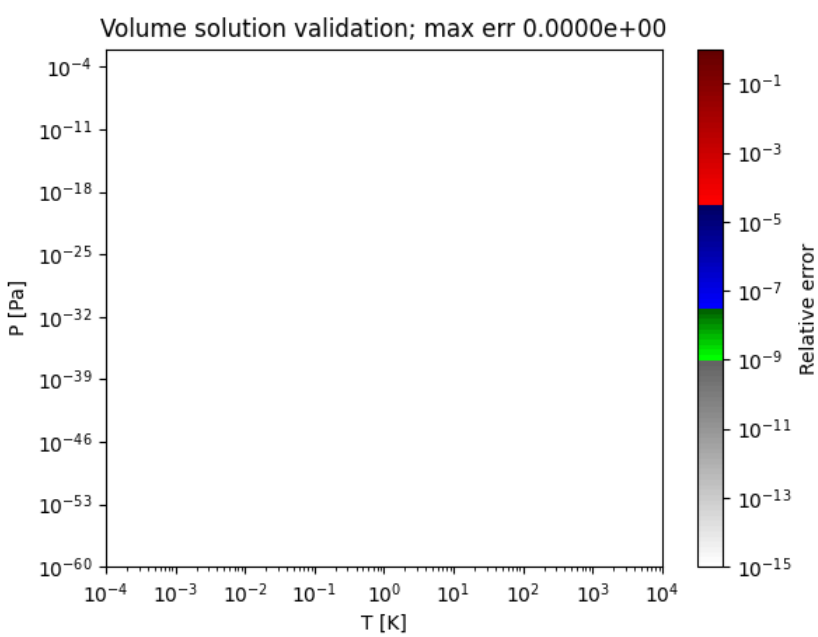
a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

dps [int] Number of decimal places in the result by *mpmath*, [-]

Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]


Volume solution validation; max err 2.8686e-14
P [Pa]
T [K]
Relative error


Volume solution validation; max err 0.0000e+00
P [Pa]
T [K]
Relative error

Notes

Although *mpmath* has a cubic solver, it has been found to fail to solve in some cases. Accordingly, the algorithm is as follows:

Working precision is *dps* plus 40 digits; and if P < 1e-10 Pa, it is *dps* plus 400 digits. The input parameters are converted exactly to *mpf* objects on input.

polyroots from mpmath is used with *maxsteps=2000*, and extra precision of 15 digits. If the solution does not converge, 20 extra digits are added up to 8 times. If no solution is found, mpmath's *findroot* is called on the pressure error function using three initial guesses from another solver.

Needless to say, this function is quite slow.

References

[1]

Examples

Test case which presented issues for PR EOS (three roots were not being returned):

```
>>> volume_solutions_mpmath(0.01, 1e-05, 2.5405184201558786e-05, 5.
↪081036840311757e-05, -6.454233843151321e-10, 0.3872747173781095)
(mpf('0.0000254054613415548712260258773060137'), mpf('4.
↪66038025602155259976574392093252'), mpf('8309.80218708657190094424659859346'))
```

7.7.2 Standard Peng-Robinson Family EOSs

Standard Peng Robinson

class thermo.eos.**PR**(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*)

Bases: `thermo.eos.GCEOS`

Class for solving the Peng-Robinson [1] [2] cubic equation of state for a pure compound. Subclasses `GCEOS`, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

The main methods here are `PR.a_alpha_and_derivatives_pure`, which calculates $a\alpha$ and its first and second derivatives, and `PR.solve_T`, which from a specified *P* and *V* obtains *T*.

Two of (*T*, *P*, *V*) are needed to solve the EOS.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha(T) = [1+\kappa(1-\sqrt{T_r})]^2$$

$$\kappa = 0.37464 + 1.54226\omega - 0.26992\omega^2$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

The constants in the expresions for a and b are given to full precision in the actual code, as derived in [3].

The full expression for critical compressibility is:

$$Z_c = \frac{1}{32}\left(\sqrt[3]{16\sqrt{2}-13} - \frac{7}{\sqrt[3]{16\sqrt{2}-13}} + 11\right)$$

References

[1], [2], [3]

Examples

T-P initialization, and exploring each phase's properties:

```
>>> eos = PR(Tc=507.6, Pc=3025000.0, omega=0.2975, T=400., P=1E6)
>>> eos.V_l, eos.V_g
(0.000156073184785, 0.0021418768167)
>>> eos.phase
'l/g'
>>> eos.H_dep_l, eos.H_dep_g
(-26111.8775716, -3549.30057795)
>>> eos.S_dep_l, eos.S_dep_g
(-58.098447843, -6.4394518931)
>>> eos.U_dep_l, eos.U_dep_g
(-22942.1657091, -2365.3923474)
>>> eos.G_dep_l, eos.G_dep_g
(-2872.49843435, -973.51982071)
>>> eos.A_dep_l, eos.A_dep_g
(297.21342811, 210.38840980)
>>> eos.beta_l, eos.beta_g
(0.00269337091778, 0.0101232239111)
>>> eos.kappa_l, eos.kappa_g
(9.3357215438e-09, 1.97106698097e-06)
>>> eos.Cp_minus_Cv_l, eos.Cp_minus_Cv_g
(48.510162249, 44.544161128)
>>> eos.Cv_dep_l, eos.Cp_dep_l
(18.8921126734, 59.0878123050)
```

P-T initialization, liquid phase, and round robin trip:

```
>>> eos = PR(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000130222125139, -31134.75084, -72.47561931)
```

T-V initialization, liquid phase:

```
>>> eos2 = PR(Tc=507.6, Pc=3025000, omega=0.2975, T=299., V=eos.V_l)
>>> eos2.P, eos2.phase
(1000000.00, 'l')
```

P-V initialization at same state:

```
>>> eos3 = PR(Tc=507.6, Pc=3025000, omega=0.2975, V=eos.V_l, P=1E6)
>>> eos3.T, eos3.phase
(299.0000000000, 'l')
```

Methods

P_max_at_V(V)	Method to calculate the maximum pressure the EOS can create at a constant volume, if one exists; returns None otherwise.
a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for this EOS.
d3a_alpha_dT3_pure(T)	Method to calculate the third temperature derivative of *a_alpha*.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the PR EOS.

P_max_at_V(*V*)

Method to calculate the maximum pressure the EOS can create at a constant volume, if one exists; returns None otherwise.

Parameters

V [float] Constant molar volume, [m^3/mol]

Returns

P [float] Maximum possible isochoric pressure, [Pa]

Notes

The analytical determination of this formula involved some part of the discriminant, and much black magic.

Examples

```
>>> e = PR(P=1e5, V=0.0001437, Tc=512.5, Pc=8084000.0, omega=0.559)
>>> e.P_max_at_V(e.V)
2247886208.7
```

Zc = 0.30740130869870386

Mechanical compressibility of Peng-Robinson EOS

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Uses the set values of *Tc*, *kappa*, and *a*.

$$a\alpha = a\left(\kappa\left(-\frac{T^{0.5}}{Tc^{0.5}}+1\right)+1\right)^2$$

$$\frac{da\alpha}{dT} = -\frac{1.0a\kappa}{T^{0.5}Tc^{0.5}}\left(\kappa\left(-\frac{T^{0.5}}{Tc^{0.5}}+1\right)+1\right)$$

$$\frac{d^2a\alpha}{dT^2} = 0.5a\kappa\left(-\frac{1}{T^{1.5}Tc^{0.5}}\left(\kappa\left(\frac{T^{0.5}}{Tc^{0.5}}-1\right)-1\right)+\frac{\kappa}{T^{1.0}Tc^{1.0}}\right)$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

Examples

Dodecane at 250 K:

```
>>> eos = PR(Tc=658.0, Pc=1820000.0, omega=0.562, T=500., P=1e5)
>>> eos.a_alpha_and_derivatives_pure(250.0)
(15.66839156301, -0.03094091246957, 9.243186769880e-05)
```

a_alpha_pure(T)

Method to calculate $a\alpha$ for this EOS. Uses the set values of *Tc*, *kappa*, and *a*.

$$a\alpha = a\left(\kappa\left(-\frac{T^{0.5}}{Tc^{0.5}} + 1\right) + 1\right)^2$$

Parameters

T [float] Temperature at which to calculate the value, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

Examples

Dodecane at 250 K:

```
>>> eos = PR(Tc=658.0, Pc=1820000.0, omega=0.562, T=500., P=1e5)
>>> eos.a_alpha_pure(250.0)
15.66839156301
```

c1 = 0.4572355289213822

Full value of the constant in the *a* parameter

c2 = 0.07779607390388846

Full value of the constant in the *b* parameter

d3a_alpha_dT3_pure(T)

Method to calculate the third temperature derivative of *a_alpha*. Uses the set values of *Tc*, *kappa*, and *a*. This property is not normally needed.

$$\frac{d^3a\alpha}{dT^3} = \frac{3a\kappa\left(-\frac{\kappa}{T_c} + \frac{\sqrt{\frac{T}{T_c}}\left(\kappa\left(\sqrt{\frac{T}{T_c}}-1\right)-1\right)}{T}\right)}{4T^2}$$

Parameters

T [float] Temperature at which to calculate the derivative, [-]

Returns

d3a_alpha_dT3 [float] Third temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^3]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

Examples

Dodecane at 500 K:

```
>>> eos = PR(Tc=658.0, Pc=1820000.0, omega=0.562, T=500., P=1e5)
>>> eos.d3a_alpha_dT3_pure(500.0)
-9.8038800671e-08
```

solve_T(*P*, *V*, *solution=None*)

Method to calculate *T* from a specified *P* and *V* for the PR EOS. Uses *Tc*, *a*, *b*, and *kappa* as well, obtained from the class's namespace.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

Notes

The exact solution can be derived as follows, and is excluded for breviety.

```
>>> from sympy import *
>>> P, T, V = symbols('P, T, V')
>>> Tc, Pc, omega = symbols('Tc, Pc, omega')
>>> R, a, b, kappa = symbols('R, a, b, kappa')
>>> a_alpha = a*(1 + kappa*(1-sqrt(T/Tc)))**2
>>> PR_formula = R*T/(V-b) - a_alpha/(V*(V+b)+b*(V-b)) - P
>>> #solve(PR_formula, T)
```

After careful evaluation of the results of the analytical formula, it was discovered, that numerical precision issues required several NR refinement iterations; at at times, when the analytical value is extremely erroneous, a call to a full numerical solver not using the analytical solution at all is required.

Examples

```
>>> eos = PR(Tc=658.0, Pc=1820000.0, omega=0.562, T=500., P=1e5)
>>> eos.solve_T(P=eos.P, V=eos.V_g)
500.0000000
```

Peng Robinson (1978)

class thermo.eos.**PR78**(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.PR*

Class for solving the Peng-Robinson cubic equation of state for a pure compound according to the 1978 variant [1] [2]. Subclasses *PR*, which provides everything except the variable *kappa*. Solves the EOS on initialization. See *PR* for further documentation.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha(T) = [1+\kappa(1-\sqrt{T_r})]^2$$

$$\kappa_i = 0.37464 + 1.54226\omega_i - 0.26992\omega_i^2 \text{ if } \omega_i \le 0.491$$

$$\kappa_i = 0.379642 + 1.48503\omega_i - 0.164423\omega_i^2 + 0.016666\omega_i^3 \text{ if } \omega_i > 0.491$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This variant is recommended over the original.

References

[1], [2]

Examples

P-T initialization (furfuryl alcohol), liquid phase:

```
>>> eos = PR78(Tc=632, Pc=5350000, omega=0.734, T=299., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 8.3519628969e-05, -63764.671093, -130.737153225)
```

high_omega_constants = (0.379642, 1.48503, -0.164423, 0.016666)
Constants for the *kappa* formula for the high-omega region.

low_omega_constants = (0.37464, 1.54226, -0.26992)
Constants for the *kappa* formula for the low-omega region.

Peng Robinson Stryjek-Vera

class thermo.eos.**PRSV**(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*, *kappa1=None*)
Bases: *thermo.eos.PR*

Class for solving the Peng-Robinson-Stryjek-Vera equations of state for a pure compound as given in [1]. The same as the Peng-Robinson EOS, except with a different *kappa* formula and with an optional fit parameter. Subclasses *PR*, which provides only several constants. See *PR* for further documentation and examples.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha(T) = [1+\kappa(1-\sqrt{T_r})]^2$$

$$\kappa = \kappa_0 + \kappa_1(1+T_r^{0.5})(0.7-T_r)$$

$$\kappa_0 = 0.378893 + 1.4897153\omega - 0.17131848\omega^2 + 0.0196554\omega^3$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

kappa1 [float, optional] Fit parameter; available in [1] for over 90 compounds, [-]

Notes

[1] recommends that *kappa1* be set to 0 for Tr > 0.7. This is not done by default; the class boolean *kappa1_Tr_limit* may be set to True and the problem re-solved with that specified if desired. *kappa1_Tr_limit* is not supported for P-V inputs.

Solutions for P-V solve for T with SciPy's *newton* solver, as there is no analytical solution for T

[2] and [3] are two more resources documenting the PRSV EOS. [4] lists *kappa* values for 69 additional compounds. See also *PRSV2*. Note that tabulated *kappa* values should be used with the critical parameters used in their fits. Both [1] and [4] only considered vapor pressure in fitting the parameter.

References

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

Examples

P-T initialization (hexane, with fit parameter in [1]), liquid phase:

```
>>> eos = PRSV(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6, kappa1=0.05104)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000130126913554, -31698.926746, -74.16751538)
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for this EOS.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the PRSV EOS.

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Uses the set values of *Tc*, *kappa0*, *kappa1*, and *a*.

The *a_alpha* function is shown below; the first and second derivatives are not shown for brevity.

$$a\alpha = a\left(\left(\kappa_0 + \kappa_1\left(\sqrt{\frac{T}{T_c}}+1\right)\left(-\frac{T}{T_c}+\frac{7}{10}\right)\right)\left(-\sqrt{\frac{T}{T_c}}+1\right)+1\right)^2$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

The expressions can be derived as follows:

```
>>> from sympy import *
>>> P, T, V = symbols('P, T, V')
>>> Tc, Pc, omega = symbols('Tc, Pc, omega')
>>> R, a, b, kappa0, kappa1 = symbols('R, a, b, kappa0, kappa1')
>>> kappa = kappa0 + kappa1*(1 + sqrt(T/Tc))*(Rational(7, 10)-T/Tc)
>>> a_alpha = a*(1 + kappa*(1-sqrt(T/Tc)))**2
>>> # diff(a_alpha, T)
>>> # diff(a_alpha, T, 2)
```

Examples

```
>>> eos = PRSV(Tc=507.6, Pc=3025000, omega=0.2975, T=406.08, P=1E6, kappa1=0.
↪05104)
>>> eos.a_alpha_and_derivatives_pure(185.0)
(4.76865472591, -0.0101408587212, 3.9138298092e-05)
```

a_alpha_pure(T)

Method to calculate $a\alpha$ for this EOS. Uses the set values of *Tc*, *kappa0*, *kappa1*, and *a*.

$$a\alpha = a\left(\left(\kappa_0 + \kappa_1\left(\sqrt{\frac{T}{T_c}} + 1\right)\left(-\frac{T}{T_c} + \frac{7}{10}\right)\right)\left(-\sqrt{\frac{T}{T_c}} + 1\right) + 1\right)^2$$

Parameters

T [float] Temperature at which to calculate the value, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

Examples

```
>>> eos = PRSV(Tc=507.6, Pc=3025000, omega=0.2975, T=406.08, P=1E6, kappa1=0.
↪05104)
>>> eos.a_alpha_pure(185.0)
4.7686547259
```

solve_T(P, V, *solution=None*)

Method to calculate T from a specified P and V for the PRSV EOS. Uses *Tc*, *a*, *b*, *kappa0* and *kappa* as well, obtained from the class's namespace.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

Notes

Not guaranteed to produce a solution. There are actually two solution, one much higher than normally desired; it is possible the solver could converge on this.

Peng Robinson Stryjek-Vera 2

class `thermo.eos.PRSV2`(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*, *kappa1=0*, *kappa2=0*, *kappa3=0*)

Bases: `thermo.eos.PR`

Class for solving the Peng-Robinson-Stryjek-Vera 2 equations of state for a pure compound as given in [1]. The same as the Peng-Robinson EOS, except with a different *kappa* formula and with three fit parameters. Subclasses `PR`, which provides only several constants. See `PR` for further documentation and examples.

$$P = \frac{RT}{v-b}-\frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a=0.45724\frac{R^2T_c^2}{P_c}$$

$$b=0.07780\frac{RT_c}{P_c}$$

$$\alpha(T)=[1+\kappa(1-\sqrt{T_r})]^2$$

$$\kappa = \kappa_0 + [\kappa_1 + \kappa_2(\kappa_3 - T_r)(1-T_r^{0.5})](1 + T_r^{0.5})(0.7 - T_r)$$

$$\kappa_0 = 0.378893 + 1.4897153\omega - 0.17131848\omega^2 + 0.0196554\omega^3$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

kappa1 [float, optional] Fit parameter; available in [1] for over 90 compounds, [-]

kappa2 [float, optional] Fit parameter; available in [1] for over 90 compounds, [-]

kappa [float, optional] Fit parameter; available in [1] for over 90 compounds, [-]

Notes

Note that tabulated *kappa* values should be used with the critical parameters used in their fits. [1] considered only vapor pressure in fitting the parameter.

References

[1]

Examples

P-T initialization (hexane, with fit parameter in [1]), liquid phase:

```
>>> eos = PRSV2(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6, kappa1=0.05104,␣
↪kappa2=0.8634, kappa3=0.460)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000130188257591, -31496.1841687, -73.615282963)
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for this EOS.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the PRSV2 EOS.

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Uses the set values of *Tc*, *kappa0*, *kappa1*, *kappa2*, *kappa3*, and *a*.

$$\alpha(T) = [1 + \kappa(1 - \sqrt{T_r})]^2$$

$$\kappa = \kappa_0 + [\kappa_1 + \kappa_2(\kappa_3 - T_r)(1 - T_r^{0.5})](1 + T_r^{0.5})(0.7 - T_r)$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

The first and second derivatives of *a_alpha* are available through the following SymPy expression.

```
>>> from sympy import *
>>> P, T, V = symbols('P, T, V')
>>> Tc, Pc, omega = symbols('Tc, Pc, omega')
>>> R, a, b, kappa0, kappa1, kappa2, kappa3 = symbols('R, a, b, kappa0, kappa1,␣
↪kappa2, kappa3')
>>> Tr = T/Tc
>>> kappa = kappa0 + (kappa1 + kappa2*(kappa3-Tr)*(1-
↪sqrt(Tr)))*(1+sqrt(Tr))*(Rational('0.7')-Tr)
>>> a_alpha = a*(1 + kappa*(1-sqrt(T/Tc)))**2
>>> diff(a_alpha, T)
>>> diff(a_alpha, T, 2)
```

Examples

```
>>> eos = PRSV2(Tc=507.6, Pc=3025000, omega=0.2975, T=400., P=1E6, kappa1=0.
↪05104, kappa2=0.8634, kappa3=0.460)
>>> eos.a_alpha_and_derivatives_pure(311.0)
(3.7245418495, -0.0066115440470, 2.05871011677e-05)
```

a_alpha_pure(*T*)

Method to calculate $a\alpha$ for this EOS. Uses the set values of *Tc*, *kappa0*, *kappa1*, *kappa2*, *kappa3*, and *a*.

$$\alpha(T) = [1 + \kappa(1 - \sqrt{T_r})]^2$$

$$\kappa = \kappa_0 + [\kappa_1 + \kappa_2(\kappa_3 - T_r)(1 - T_r^{0.5})](1 + T_r^{0.5})(0.7 - T_r)$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Examples

```
>>> eos = PRSV2(Tc=507.6, Pc=3025000, omega=0.2975, T=400., P=1E6, kappa1=0.
↪05104, kappa2=0.8634, kappa3=0.460)
>>> eos.a_alpha_pure(1276.0)
33.321674050
```

solve_T(*P*, *V*, *solution=None*)

Method to calculate *T* from a specified *P* and *V* for the PRSV2 EOS. Uses *Tc*, *a*, *b*, *kappa0*, *kappa1*, *kappa2*, and *kappa3* as well, obtained from the class's namespace.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

Notes

Not guaranteed to produce a solution. There are actually 8 solutions, six with an imaginary component at a tested point. The two temperature solutions are quite far apart, with one much higher than the other; it is possible the solver could converge on the higher solution, so use T inputs with care. This extra solution is a perfectly valid one however. The secant method is implemented at present.

Examples

```
>>> eos = PRSV2(Tc=507.6, Pc=3025000, omega=0.2975, T=400., P=1E6, kappa1=0.
↪05104, kappa2=0.8634, kappa3=0.460)
>>> eos.solve_T(P=eos.P, V=eos.V_g)
400.0
```

Peng Robinson Twu (1995)

class thermo.eos.**TWUPR**(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos_alpha_functions.TwuPR95_a_alpha*, *thermo.eos.PR*

Class for solving the Twu (1995) [1] variant of the Peng-Robinson cubic equation of state for a pure compound. Subclasses *PR*, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

The main implemented method here is *a_alpha_and_derivatives_pure*, which sets $a\alpha$ and its first and second derivatives.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)}\exp[L(1-T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.125283, 0.911807, 1.948150;

L1, M1, N1 = 0.511614, 0.784054, 2.812520

For supercritical conditions:

L0, M0, N0 = 0.401219, 4.963070, -0.2;

L1, M1, N1 = 0.024955, 1.248089, -8.

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

Claimed to be more accurate than the PR, PR78 and PRSV equations.

There is no analytical solution for T. There are multiple possible solutions for T under certain conditions; no guaranteed are provided regarding which solution is obtained.

References

[1]

Examples

```
>>> eos = TWUPR(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6)
>>> eos.V_l, eos.H_dep_l, eos.S_dep_l
(0.00013017554170, -31652.73712, -74.112850429)
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function.
a_alpha_pure(T)	Method to calculate $a\alpha$ for the Twu alpha function.

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)}\exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.125283, 0.911807, 1.948150;

L1, M1, N1 = 0.511614, 0.784054, 2.812520

For supercritical conditions:

L0, M0, N0 = 0.401219, 4.963070, -0.2;

L1, M1, N1 = 0.024955, 1.248089, -8.

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

The derivatives are somewhat long and are not described here for brevity; they are obtainable from the following SymPy expression.

```
>>> from sympy import *
>>> T, Tc, omega, N1, N0, M1, M0, L1, L0 = symbols('T, Tc, omega, N1, N0, M1,␣
↪M0, L1, L0')
>>> Tr = T/Tc
>>> alpha0 = Tr**(N0*(M0-1))*exp(L0*(1-Tr**(N0*M0)))
>>> alpha1 = Tr**(N1*(M1-1))*exp(L1*(1-Tr**(N1*M1)))
>>> alpha = alpha0 + omega*(alpha1-alpha0)
>>> diff(alpha, T)
>>> diff(alpha, T, T)
```

a_alpha_pure(T)

Method to calculate $a\alpha$ for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.125283, 0.911807, 1.948150;

L1, M1, N1 = 0.511614, 0.784054, 2.812520

For supercritical conditions:

L0, M0, N0 = 0.401219, 4.963070, -0.2;

L1, M1, N1 = 0.024955, 1.248089, -8.

Parameters

T [float] Temperature at which to calculate the value, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

Peng Robinson Polynomial alpha Function

class thermo.eos.**PRTranslatedPoly**(*Tc*, *Pc*, *omega*, *alpha_coeffs=None*, *c=0.0*, *T=None*, *P=None*, *V=None*)
Bases: *thermo.eos_alpha_functions.Poly_a_alpha*, *thermo.eos.PRTranslated*

Class for solving the volume translated Peng-Robinson equation of state with a polynomial alpha function. With the right coefficients, this model can reproduce any property incredibly well. Subclasses *PRTranslated*. Solves the EOS on initialization. This is intended as a base class for all translated variants of the Peng-Robinson EOS.

$$P = \frac{RT}{v+c-b} - \frac{a\alpha(T)}{(v+c)(v+c+b)+b(v+c-b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha(T) = f(T)$$

$$\kappa = 0.37464 + 1.54226\omega - 0.26992\omega^2$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

alpha_coeffs [tuple or None] Coefficients which may be specified by subclasses; set to None to use the original Peng-Robinson alpha function, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Examples

Methanol, with alpha functions reproducing CoolProp's implementation of its vapor pressure (up to 13 coefficients)

```
>>> alpha_coeffs_exact = [9.645280470011588e-32, -4.362226651748652e-28, 9.
↪034194757823037e-25, -1.1343330204981244e-21, 9.632898335494218e-19, -5.
↪841502902171077e-16, 2.601801729901228e-13, -8.615431349241052e-11, 2.
↪1202999753932622e-08, -3.829144045293198e-06, 0.0004930777289075716, -0.
↪04285337965522619, 2.2473964123842705, -51.13852710672087]
>>> kwargs = dict(Tc=512.5, Pc=8084000.0, omega=0.559, alpha_coeffs=alpha_coeffs_
↪exact, c=1.557458e-05)
>>> eos = PRTranslatedPoly(T=300, P=1e5, **kwargs)
```

(continues on next page)

(continued from previous page)

```
>>> eos.Psat(500)/PropsSI("P", 'T', 500.0, 'Q', 0, 'methanol')
1.0000112765
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives given that there is a polynomial equation for α.
a_alpha_pure(T)	Method to calculate *a_alpha* given that there is a polynomial equation for α.

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives given that there is a polynomial equation for α.

$$a\alpha = a \cdot \text{poly}(T)$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alpha_pure(*T*)

Method to calculate *a_alpha* given that there is a polynomial equation for α.

$$a\alpha = a \cdot \text{poly}(T)$$

Parameters

T [float] Temperature, [K]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

7.7.3 Volume Translated Peng-Robinson Family EOSs

Peng Robinson Translated

class thermo.eos.**PRTranslated**(*Tc*, *Pc*, *omega*, *alpha_coeffs=None*, *c=0.0*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.PR*

Class for solving the volume translated Peng-Robinson equation of state. Subclasses *PR*. Solves the EOS on initialization. This is intended as a base class for all translated variants of the Peng-Robinson EOS.

$$P = \frac{RT}{v + c - b} - \frac{a\alpha(T)}{(v + c)(v + c + b) + b(v + c - b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha(T) = [1 + \kappa(1 - \sqrt{T_r})]^2$$

$$\kappa = 0.37464 + 1.54226\omega - 0.26992\omega^2$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

alpha_coeffs [tuple or None] Coefficients which may be specified by subclasses; set to None to use the original Peng-Robinson alpha function, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

References

[1]

Examples

P-T initialization:

```
>>> eos = PRTranslated(T=305, P=1.1e5, Tc=512.5, Pc=8084000.0, omega=0.559, c=-1e-6)
>>> eos.phase, eos.V_l, eos.V_g
('l/g', 4.90798083711e-05, 0.0224350982488)
```

Peng Robinson Translated Twu (1991)

class thermo.eos.**PRTranslatedTwu**(*Tc*, *Pc*, *omega*, *alpha_coeffs=None*, *c=0.0*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos_alpha_functions.Twu91_a_alpha*, *thermo.eos.PRTranslated*

Class for solving the volume translated Peng-Robinson equation of state with the Twu (1991) [1] alpha function. Subclasses *thermo.eos_alpha_functions.Twu91_a_alpha* and *PRTranslated*. Solves the EOS on initialization.

$$P = \frac{RT}{v + c - b} - \frac{a\alpha(T)}{(v + c)(v + c + b) + b(v + c - b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha = \left(\frac{T}{T_c}\right)^{c_3(c_2-1)} e^{c_1\left(-\left(\frac{T}{T_c}\right)^{c_2c_3}+1\right)}$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

alpha_coeffs [tuple(float[3])] Coefficients L, M, N (also called C1, C2, C3) of TWU 1991 form, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This variant offers substantial improvements to the PR-type EOSs - likely getting about as accurate as this form of cubic equation can get.

References

[1]

Examples

P-T initialization:

```
>>> alpha_coeffs = (0.694911381318495, 0.919907783415812, 1.70412689631515)
>>> kwargs = dict(Tc=512.5, Pc=8084000.0, omega=0.559, alpha_coeffs=alpha_coeffs,␣
↪c=-1e-6)
>>> eos = PRTranslatedTwu(T=300, P=1e5, **kwargs)
>>> eos.phase, eos.V_l, eos.V_g
('l/g', 4.8918748906e-05, 0.024314406330)
```

Peng Robinson Translated-Consistent

class thermo.eos.**PRTranslatedConsistent**(*Tc*, *Pc*, *omega*, *alpha_coeffs=None*, *c=None*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.PRTranslatedTwu*

Class for solving the volume translated Le Guennec, Privat, and Jaubert revision of the Peng-Robinson equation of state for a pure compound according to [1]. Subclasses *PRTranslatedTwu*, which provides everything except the estimation of *c* and the alpha coefficients. This model's *alpha* is based on the TWU 1991 model; when estimating, *N* is set to 2. Solves the EOS on initialization. See *PRTranslated* for further documentation.

$$P = \frac{RT}{v + c - b} - \frac{a\alpha(T)}{(v + c)(v + c + b) + b(v + c - b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha = \left(\frac{T}{T_c}\right)^{c_3(c_2-1)} e^{c_1\left(-\left(\frac{T}{T_c}\right)^{c_2 c_3}+1\right)}$$

If c is not provided, it is estimated as:

$$c = \frac{RT_c}{P_c}(0.0198\omega - 0.0065)$$

If *alpha_coeffs* is not provided, the parameters L and M are estimated from the acentric factor as follows:

$$L = 0.1290\omega^2 + 0.6039\omega + 0.0877$$

$$M = 0.1760\omega^2 - 0.2600\omega + 0.8884$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

alpha_coeffs [tuple(float[3]), optional] Coefficients L, M, N (also called C1, C2, C3) of TWU 1991 form, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This variant offers substantial improvements to the PR-type EOSs - likely getting about as accurate as this form of cubic equation can get.

References

[1]

Examples

P-T initialization (methanol), liquid phase:

```
>>> eos = PRTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=250., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000124374813374486, -34155.16119794619, -83.34913258614345)
```

Peng Robinson Translated (Pina-Martinez, Privat, and Jaubert Variant)

class thermo.eos.**PRTranslatedPPJP**(*Tc*, *Pc*, *omega*, *c=0.0*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.PRTranslated*

Class for solving the volume translated Pina-Martinez, Privat, Jaubert, and Peng revision of the Peng-Robinson equation of state for a pure compound according to [1]. Subclasses *PRTranslated*, which provides everything except the variable *kappa*. Solves the EOS on initialization. See *PRTranslated* for further documentation.

$$P = \frac{RT}{v + c - b} - \frac{a\alpha(T)}{(v + c)(v + c + b) + b(v + c - b)}$$

$$a = 0.45724\frac{R^2T_c^2}{P_c}$$

$$b = 0.07780\frac{RT_c}{P_c}$$

$$\alpha(T) = [1 + \kappa(1 - \sqrt{T_r})]^2$$

$$\kappa = 0.3919 + 1.4996\omega - 0.2721\omega^2 + 0.1063\omega^3$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This variant offers incremental improvements in accuracy only, but those can be fairly substantial for some substances.

References

[1]

Examples

P-T initialization (methanol), liquid phase:

```
>>> eos = PRTranslatedPPJP(Tc=507.6, Pc=3025000, omega=0.2975, c=0.6390E-6, T=250.,
↪P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.0001229231238092, -33466.2428296, -80.75610242427)
```

7.7.4 Soave-Redlich-Kwong Family EOSs

Standard SRK

class thermo.eos.**SRK**(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.GCEOS*

Class for solving the Soave-Redlich-Kwong [1] [2] [3] cubic equation of state for a pure compound. Subclasses *GCEOS*, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2}-1)P_c}\right) = \frac{0.42748\cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664\cdot RT_c}{P_c}$$

$$\alpha(T) = \left[1 + m\left(1 - \sqrt{\frac{T}{T_c}}\right)\right]^2$$

$$m = 0.480 + 1.574\omega - 0.176\omega^2$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

References

[1], [2], [3]

Examples

```
>>> eos = SRK(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000146821077354, -31754.663859, -74.373272044)
```

Methods

P_max_at_V(V)	Method to calculate the maximum pressure the EOS can create at a constant volume, if one exists; returns None otherwise.
a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for this EOS.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the SRK EOS.

P_max_at_V(*V*)

Method to calculate the maximum pressure the EOS can create at a constant volume, if one exists; returns None otherwise.

Parameters

V [float] Constant molar volume, [m^3/mol]

Returns

P [float] Maximum possible isochoric pressure, [Pa]

Notes

The analytical determination of this formula involved some part of the discriminant, and much black magic.

Examples

```
>>> e = SRK(P=1e5, V=0.0001437, Tc=512.5, Pc=8084000.0, omega=0.559)
>>> e.P_max_at_V(e.V)
490523786.2
```

Zc = 0.3333333333333333

Mechanical compressibility of *SRK* EOS

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Uses the set values of *Tc*, *m*, and *a*.

$$a\alpha = a\left(m\left(-\sqrt{\frac{T}{T_c}}+1\right)+1\right)^2$$

$$\frac{da\alpha}{dT} = \frac{am}{T}\sqrt{\frac{T}{T_c}}\left(m\left(\sqrt{\frac{T}{T_c}}-1\right)-1\right)$$

$$\frac{d^2a\alpha}{dT^2} = \frac{am\sqrt{\frac{T}{T_c}}}{2T^2}\left(m+1\right)$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

a_alpha_pure(*T*)

Method to calculate $a\alpha$ for this EOS. Uses the set values of *Tc*, *m*, and *a*.

$$a\alpha = a\left(m\left(-\sqrt{\frac{T}{T_c}}+1\right)+1\right)^2$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

c1 = 0.4274802335403414

Full value of the constant in the *a* parameter

c2 = 0.08664034996495772

Full value of the constant in the *b* parameter

epsilon = 0.0

epsilon is always zero for the *SRK* EOS

solve_T(*P*, *V*, *solution=None*)

Method to calculate T from a specified P and V for the SRK EOS. Uses *a*, *b*, and *Tc* obtained from the class's namespace.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

Notes

The exact solution can be derived as follows; it is excluded for brevity.

```
>>> from sympy import *
>>> P, T, V, R, a, b, m = symbols('P, T, V, R, a, b, m')
>>> Tc, Pc, omega = symbols('Tc, Pc, omega')
>>> a_alpha = a*(1 + m*(1-sqrt(T/Tc)))**2
>>> SRK = R*T/(V-b) - a_alpha/(V*(V+b)) - P
>>> solve(SRK, T)
```

Twu SRK (1995)

class thermo.eos.**TWUSRK**(*Tc*, *Pc*, *omega*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos_alpha_functions.TwuSRK95_a_alpha*, *thermo.eos.SRK*

Class for solving the Soave-Redlich-Kwong cubic equation of state for a pure compound. Subclasses *GCEOS*, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

The main implemented method here is *a_alpha_and_derivatives_pure*, which sets $a\alpha$ and its first and second derivatives.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2}-1)P_c}\right) = \frac{0.42748\cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664\cdot RT_c}{P_c}$$

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)}\exp[L(1-T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.141599, 0.919422, 2.496441

L1, M1, N1 = 0.500315, 0.799457, 3.291790

For supercritical conditions:

L0, M0, N0 = 0.441411, 6.500018, -0.20

L1, M1, N1 = 0.032580, 1.289098, -8.0

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

There is no analytical solution for T. There are multiple possible solutions for T under certain conditions; no guaranteed are provided regarding which solution is obtained.

References

[1]

Examples

```
>>> eos = TWUSRK(Tc=507.6, Pc=3025000, omega=0.2975, T=299., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000146892222966, -31612.6025870, -74.022966093)
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function.
a_alpha_pure(T)	Method to calculate $a\alpha$ for the Twu alpha function.

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.141599, 0.919422, 2.496441

L1, M1, N1 = 0.500315, 0.799457, 3.291790

For supercritical conditions:

L0, M0, N0 = 0.441411, 6.500018, -0.20

L1, M1, N1 = 0.032580, 1.289098, -8.0

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

The derivatives are somewhat long and are not described here for brevity; they are obtainable from the following SymPy expression.

```
>>> from sympy import *
>>> T, Tc, omega, N1, N0, M1, M0, L1, L0 = symbols('T, Tc, omega, N1, N0, M1,␣
↪M0, L1, L0')
>>> Tr = T/Tc
>>> alpha0 = Tr**(N0*(M0-1))*exp(L0*(1-Tr**(N0*M0)))
>>> alpha1 = Tr**(N1*(M1-1))*exp(L1*(1-Tr**(N1*M1)))
>>> alpha = alpha0 + omega*(alpha1-alpha0)
>>> diff(alpha, T)
>>> diff(alpha, T, T)
```

a_alpha_pure(T)

Method to calculate $a\alpha$ for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.141599, 0.919422, 2.496441

L1, M1, N1 = 0.500315, 0.799457, 3.291790

For supercritical conditions:

L0, M0, N0 = 0.441411, 6.500018, -0.20

L1, M1, N1 = 0.032580, 1.289098, -8.0

Parameters

T [float] Temperature at which to calculate the value, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

API SRK

class thermo.eos.**APISRK**(*Tc*, *Pc*, *omega=None*, *T=None*, *P=None*, *V=None*, *S1=None*, *S2=0*)

Bases: *thermo.eos.SRK*

Class for solving the Refinery Soave-Redlich-Kwong cubic equation of state for a pure compound shown in the API Databook [1]. Subclasses *GCEOS*, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

Implemented methods here are *a_alpha_and_derivatives*, which sets $a\alpha$ and its first and second derivatives, and *solve_T*, which from a specified *P* and *V* obtains *T*. Two fit constants are used in this expresion, with an estimation scheme for the first if unavailable and the second may be set to zero.

Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2}-1)P_c}\right) = \frac{0.42748\cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664\cdot RT_c}{P_c}$$

$$\alpha(T) = \left[1 + S_1\left(1-\sqrt{T_r}\right) + S_2\frac{1-\sqrt{T_r}}{\sqrt{T_r}}\right]^2$$

$$S_1 = 0.48508 + 1.55171\omega - 0.15613\omega^2 \text{ if S1 is not tabulated}$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float, optional] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

S1 [float, optional] Fit constant or estimated from acentric factor if not provided [-]

S2 [float, optional] Fit constant or 0 if not provided [-]

References

[1]

Examples

```
>>> eos = APISRK(Tc=514.0, Pc=6137000.0, S1=1.678665, S2=-0.216396, P=1E6, T=299)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 7.0456950702e-05, -42826.286146, -103.626979037)
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for this EOS.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the API SRK EOS.

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Uses the set values of *Tc*, *a*, *S1*, and *S2*.

$$a\alpha(T) = a\left[1 + S_1\left(1-\sqrt{T_r}\right) + S_2\frac{1-\sqrt{T_r}}{\sqrt{T_r}}\right]^2$$

$$\frac{da\alpha}{dT} = a\frac{T_c}{T^2}\left(-S_2\left(\sqrt{\frac{T}{T_c}}-1\right)+\sqrt{\frac{T}{T_c}}\left(S_1\sqrt{\frac{T}{T_c}}+S_2\right)\right)\left(S_2\left(\sqrt{\frac{T}{T_c}}-1\right)+\sqrt{\frac{T}{T_c}}\left(S_1\left(\sqrt{\frac{T}{T_c}}-1\right)-1\right)\right)$$

$$\frac{d^2a\alpha}{dT^2} = a\frac{1}{2T^3}\left(S_1^2T\sqrt{\frac{T}{T_c}} - S_1S_2T\sqrt{\frac{T}{T_c}} + 3S_1S_2Tc\sqrt{\frac{T}{T_c}} + S_1T\sqrt{\frac{T}{T_c}} - 3S_2^2Tc\sqrt{\frac{T}{T_c}} + 4S_2^2Tc + 3S_2Tc\sqrt{\frac{T}{T_c}}\right)$$

a_alpha_pure(*T*)

Method to calculate $a\alpha$ for this EOS. Uses the set values of *Tc*, *m*, and *a*.

$$a\alpha = a\left(m\left(-\sqrt{\frac{T}{T_c}}+1\right)+1\right)^2$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

solve_T(*P*, *V*, *solution=None*)

Method to calculate T from a specified P and V for the API SRK EOS. Uses *a*, *b*, and *Tc* obtained from the class's namespace.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

Notes

If S2 is set to 0, the solution is the same as in the SRK EOS, and that is used. Otherwise, newton's method must be used to solve for T. There are 8 roots of T in that case, six of them real. No guarantee can be made regarding which root will be obtained.

SRK Translated

class thermo.eos.**SRKTranslated**(*Tc*, *Pc*, *omega*, *alpha_coeffs=None*, *c=0.0*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.SRK*

Class for solving the volume translated Peng-Robinson equation of state. Subclasses *SRK*. Solves the EOS on initialization. This is intended as a base class for all translated variants of the SRK EOS.

$$P = \frac{RT}{V+c-b} - \frac{a\alpha(T)}{(V+c)(V+c+b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2}-1)P_c}\right) = \frac{0.42748\cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664\cdot RT_c}{P_c}$$

$$\alpha(T) = \left[1 + m\left(1 - \sqrt{\frac{T}{T_c}}\right)\right]^2$$

$$m = 0.480 + 1.574\omega - 0.176\omega^2$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

alpha_coeffs [tuple or None] Coefficients which may be specified by subclasses; set to None to use the original Peng-Robinson alpha function, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

References

[1]

Examples

P-T initialization:

```
>>> eos = SRKTranslated(T=305, P=1.1e5, Tc=512.5, Pc=8084000.0, omega=0.559, c=-1e-
↪6)
>>> eos.phase, eos.V_l, eos.V_g
('l/g', 5.5131657318e-05, 0.022447661363)
```

SRK Translated-Consistent

class thermo.eos.**SRKTranslatedConsistent**(*Tc*, *Pc*, *omega*, *alpha_coeffs=None*, *c=None*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos_alpha_functions.Twu91_a_alpha*, *thermo.eos.SRKTranslated*

Class for solving the volume translated Le Guennec, Privat, and Jaubert revision of the SRK equation of state for a pure compound according to [1].

This model's *alpha* is based on the TWU 1991 model; when estimating, *N* is set to 2. Solves the EOS on initialization. See *SRK* for further documentation.

$$P = \frac{RT}{V + c - b} - \frac{a\alpha(T)}{(V + c)(V + c + b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2} - 1)P_c}\right) = \frac{0.42748 \cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2} - 1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664 \cdot RT_c}{P_c}$$

$$\alpha = \left(\frac{T}{T_c}\right)^{c_3(c_2-1)} e^{c_1\left(-\left(\frac{T}{T_c}\right)^{c_2 c_3} + 1\right)}$$

If c is not provided, it is estimated as:

$$c = \frac{RT_c}{P_c}(0.0172\omega - 0.0096)$$

If *alpha_coeffs* is not provided, the parameters L and M are estimated from the acentric factor as follows:

$$L = 0.0947\omega^2 + 0.6871\omega + 0.1508$$

$$M = 0.1615\omega^2 - 0.2349\omega + 0.8876$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

alpha_coeffs [tuple(float[3]), optional] Coefficients L, M, N (also called C1, C2, C3) of TWU 1991 form, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This variant offers substantial improvements to the SRK-type EOSs - likely getting about as accurate as this form of cubic equation can get.

References

[1]

Examples

P-T initialization (methanol), liquid phase:

```
>>> eos = SRKTranslatedConsistent(Tc=507.6, Pc=3025000, omega=0.2975, T=250., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.00011846802568940222, -34324.05211005662, -83.83861726864234)
```

SRK Translated (Pina-Martinez, Privat, and Jaubert Variant)

class thermo.eos.**SRKTranslatedPPJP**(*Tc*, *Pc*, *omega*, *c=0.0*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.SRK*

Class for solving the volume translated Pina-Martinez, Privat, Jaubert, and Peng revision of the Soave-Redlich-Kwong equation of state for a pure compound according to [1]. Subclasses *SRK*, which provides everything except the variable *kappa*. Solves the EOS on initialization. See *SRK* for further documentation.

$$P = \frac{RT}{V + c - b} - \frac{a\alpha(T)}{(V + c)(V + c + b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2} - 1)P_c}\right) = \frac{0.42748 \cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2} - 1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664 \cdot RT_c}{P_c}$$

$$\alpha(T) = \left[1 + m\left(1 - \sqrt{\frac{T}{T_c}}\right)\right]^2$$

$$m = 0.4810 + 1.5963\omega - 0.2963\omega^2 + 0.1223\omega^3$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This variant offers incremental improvements in accuracy only, but those can be fairly substantial for some substances.

References

[1]

Examples

P-T initialization (hexane), liquid phase:

```
>>> eos = SRKTranslatedPPJP(Tc=507.6, Pc=3025000, omega=0.2975, c=22.3098E-6, T=250.
↪, P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.00011666322408111662, -34158.934132722185, -83.06507748137201)
```

MSRK Translated

class thermo.eos.**MSRKTranslated**(*Tc, Pc, omega, M=None, N=None, alpha_coeffs=None, c=0.0, T=None, P=None, V=None*)

Bases: *thermo.eos_alpha_functions.Soave_1979_a_alpha*, *thermo.eos.SRKTranslated*

Class for solving the volume translated Soave (1980) alpha function, revision of the Soave-Redlich-Kwong equation of state for a pure compound according to [1]. Uses two fitting parameters *N* and *M* to more accurately fit the vapor pressure of pure species. Subclasses *SRKTranslated*. Solves the EOS on initialization. See *SRKTranslated* for further documentation.

$$P = \frac{RT}{V + c - b} - \frac{a\alpha(T)}{(V + c)(V + c + b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2} - 1)P_c}\right) = \frac{0.42748 \cdot R^2(T_c)^2}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2} - 1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664 \cdot RT_c}{P_c}$$

$$\alpha(T) = 1 + (1 - T_r)(M + \frac{N}{T_r})$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

alpha_coeffs [tuple(float[3]), optional] Coefficients M, N of this EOS's alpha function, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

This is an older correlation that offers lower accuracy on many properties which were sacrificed to obtain the vapor pressure accuracy. The alpha function of this EOS does not meet any of the consistency requriements for alpha functions.

Coefficients can be found in [2], or estimated with the method in [3]. The estimation method in [3] works as follows, using the acentric factor and true critical compressibility:

$$M = 0.4745 + 2.7349(\omega Z_c) + 6.0984(\omega Z_c)^2$$

$$N = 0.0674 + 2.1031(\omega Z_c) + 3.9512(\omega Z_c)^2$$

An alternate estimation scheme is provided in [1], which provides analytical solutions to calculate the parameters *M* and *N* from two points on the vapor pressure curve, suggested as 10 mmHg and 1 atm. This is used as an estimation method here if the parameters are not provided, and the two vapor pressure points are obtained from the original SRK equation of state.

References

[1], [2], [3]

Examples

P-T initialization (hexane), liquid phase:

```
>>> eos = MSRKTranslated(Tc=507.6, Pc=3025000, omega=0.2975, c=22.0561E-6, M=0.7446,
↪ N=0.2476, T=250., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.0001169276461322, -34571.6862673, -84.757900348)
```

Methods

estimate_MN(Tc, Pc, omega[, c])	Calculate the alpha values for the MSRK equation to match two pressure points, and solve analytically for the M, N required to match exactly that.

static estimate_MN(*Tc*, *Pc*, *omega*, *c=0.0*)

Calculate the alpha values for the MSRK equation to match two pressure points, and solve analytically for the M, N required to match exactly that. Since no experimental data is available, make it up with the original SRK EOS.

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

omega [float] Acentric factor, [-]

c [float, optional] Volume translation parameter, [m^3/mol]

Returns

M [float] M parameter, [-]

N [float] N parameter, [-]

Examples

```
>>> from sympy import *
>>> Tc, m, n = symbols('Tc, m, n')
>>> T0, T1 = symbols('T_10, T_760')
>>> alpha0, alpha1 = symbols('alpha_10, alpha_760')
>>> Eqs = [Eq(alpha0, 1 + (1 - T0/Tc)*(m + n/(T0/Tc))), Eq(alpha1, 1 + (1 - T1/
↪Tc)*(m + n/(T1/Tc)))]
>>> solve(Eqs, [n, m])
```

7.7.5 Van der Waals Equations of State

class thermo.eos.**VDW**(*Tc*, *Pc*, *T=None*, *P=None*, *V=None*, *omega=None*)

Bases: *thermo.eos.GCEOS*

Class for solving the Van der Waals [1] [2] cubic equation of state for a pure compound. Subclasses *GCEOS*, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a}{V^2}$$

$$a = \frac{27}{64}\frac{(RT_c)^2}{P_c}$$

$$b = \frac{RT_c}{8P_c}$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

omega [float, optional] Acentric factor - not used in equation of state!, [-]

Notes

omega is allowed as an input for compatibility with the other EOS forms, but is not used.

References

[1], [2]

Examples

```
>>> eos = VDW(Tc=507.6, Pc=3025000, T=299., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000223329856081, -13385.7273746, -32.65923125)
```

Attributes

omega

Methods

P_discriminant_zeros_analytical(T, b, delta, ...)	Method to calculate the pressures which zero the discriminant function of the *VDW* eos.
T_discriminant_zeros_analytical([valid])	Method to calculate the temperatures which zero the discriminant function of the *VDW* eos.
a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the *VDW* EOS.

static P_discriminant_zeros_analytical(*T*, *b*, *delta*, *epsilon*, *a_alpha*, *valid=False*)

Method to calculate the pressures which zero the discriminant function of the *VDW* eos. This is an cubic function solved analytically.

Parameters

T [float] Temperature, [K]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

valid [bool] Whether to filter the calculated pressures so that they are all real, and positive only, [-]

Returns

P_discriminant_zeros [tuple[float]] Pressures which make the discriminant zero, [Pa]

Notes

Calculated analytically. Derived as follows. Has multiple solutions.

```
>>> from sympy import *
>>> P, T, V, R, b, a = symbols('P, T, V, R, b, a')
>>> P_vdw = R*T/(V-b) - a/(V*V)
>>> delta, epsilon = 0, 0
>>> eta = b
>>> B = b*P/(R*T)
>>> deltas = delta*P/(R*T)
>>> thetas = a*P/(R*T)**2
>>> epsilons = epsilon*(P/(R*T))**2
>>> etas = eta*P/(R*T)
>>> a_coeff = 1
>>> b_coeff = (deltas - B - 1)
>>> c = (thetas + epsilons - deltas*(B+1))
>>> d = -(epsilons*(B+1) + thetas*etas)
>>> disc = b_coeff*b_coeff*c*c - 4*a_coeff*c*c*c - 4*b_coeff*b_coeff*b_coeff*d -
↪ 27*a_coeff*a_coeff*d*d + 18*a_coeff*b_coeff*c*d
>>> base = -(expand(disc/P**2*R**3*T**3/a))
>>> collect(base, P).args
```

T_discriminant_zeros_analytical(*valid=False*)

Method to calculate the temperatures which zero the discriminant function of the *VDW* eos. This is an analytical cubic function solved analytically.

Parameters

valid [bool] Whether to filter the calculated temperatures so that they are all real, and positive only, [-]

Returns

T_discriminant_zeros [list[float]] Temperatures which make the discriminant zero, [K]

Notes

Calculated analytically. Derived as follows. Has multiple solutions.

```
>>> from sympy import *
>>> P, T, V, R, b, a = symbols('P, T, V, R, b, a')
>>> delta, epsilon = 0, 0
>>> eta = b
>>> B = b*P/(R*T)
>>> deltas = delta*P/(R*T)
>>> thetas = a*P/(R*T)**2
>>> epsilons = epsilon*(P/(R*T))**2
>>> etas = eta*P/(R*T)
>>> a_coeff = 1
>>> b_coeff = (deltas - B - 1)
>>> c = (thetas + epsilons - deltas*(B+1))
>>> d = -(epsilons*(B+1) + thetas*etas)
>>> disc = b_coeff*b_coeff*c*c - 4*a_coeff*c*c*c - 4*b_coeff*b_coeff*b_coeff*d -
↪ 27*a_coeff*a_coeff*d*d + 18*a_coeff*b_coeff*c*d
```

(continues on next page)

(continued from previous page)

```
>>> base = -(expand(disc/P**2*R**3*T**3/a))
>>> base_T = simplify(base*T**3)
>>> sln = collect(expand(base_T), T).args
```

Zc = 0.375
Mechanical compressibility of *VDW* EOS

a_alpha_and_derivatives_pure(*T*)
Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Uses the set values of a.

$$a\alpha = a$$

$$\frac{da\alpha}{dT} = 0$$

$$\frac{d^2a\alpha}{dT^2} = 0$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

a_alpha_pure(*T*)
Method to calculate $a\alpha$. Uses the set values of a.

$$a\alpha = a$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

delta = 0.0
delta is always zero for the *VDW* EOS

epsilon = 0.0
epsilon is always zero for the *VDW* EOS

omega = None
omega has no impact on the *VDW* EOS

solve_T(*P*, *V*, *solution=None*)
Method to calculate T from a specified P and V for the *VDW* EOS. Uses a, and b, obtained from the class's namespace.

$$T = \frac{1}{RV^2}\left(PV^2\left(V - b\right) + Va - ab\right)$$

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

7.7.6 Redlich-Kwong Equations of State

class thermo.eos.RK(*Tc*, *Pc*, *T=None*, *P=None*, *V=None*, *omega=None*)

Bases: `thermo.eos.GCEOS`

Class for solving the Redlich-Kwong [1] [2] [3] cubic equation of state for a pure compound. Subclasses `GCEOS`, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a}{V\sqrt{\frac{T}{T_c}}(V+b)}$$

$$a = \left(\frac{R^2(T_c)^2}{9(\sqrt[3]{2}-1)P_c}\right) = \frac{0.42748\cdot R^2(T_c)^{2.5}}{P_c}$$

$$b = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_c}{P_c} = \frac{0.08664\cdot RT_c}{P_c}$$

Parameters

Tc [float] Critical temperature, [K]

Pc [float] Critical pressure, [Pa]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Notes

omega is allowed as an input for compatibility with the other EOS forms, but is not used.

References

[1], [2], [3]

Examples

```
>>> eos = RK(Tc=507.6, Pc=3025000, T=299., P=1E6)
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l', 0.000151893468781, -26160.8424877, -63.013137852)
```

Attributes

omega

Methods

T_discriminant_zeros_analytical([valid])	Method to calculate the temperatures which zero the discriminant function of the *RK* eos.
a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for this EOS.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the RK EOS.

T_discriminant_zeros_analytical(*valid=False*)

Method to calculate the temperatures which zero the discriminant function of the *RK* eos. This is an analytical function with an 11-coefficient polynomial which is solved with *numpy*.

Parameters

valid [bool] Whether to filter the calculated temperatures so that they are all real, and positive only, [-]

Returns

T_discriminant_zeros [float] Temperatures which make the discriminant zero, [K]

Notes

Calculated analytically. Derived as follows. Has multiple solutions.

```
>>> from sympy import *
>>> P, T, V, R, b, a, Troot = symbols('P, T, V, R, b, a, Troot')
>>> a_alpha = a/sqrt(T)
>>> delta, epsilon = b, 0
>>> eta = b
>>> B = b*P/(R*T)
>>> deltas = delta*P/(R*T)
>>> thetas = a_alpha*P/(R*T)**2
>>> epsilons = epsilon*(P/(R*T))**2
>>> etas = eta*P/(R*T)
>>> a_coeff = 1
>>> b_coeff = (deltas - B - 1)
>>> c = (thetas + epsilons - deltas*(B+1))
>>> d = -(epsilons*(B+1) + thetas*etas)
>>> disc = b_coeff*b_coeff*c*c - 4*a_coeff*c*c*c - 4*b_coeff*b_coeff*b_coeff*d -
→ 27*a_coeff*a_coeff*d*d + 18*a_coeff*b_coeff*c*d
```

(continues on next page)

(continued from previous page)

```
>>> new_disc = disc.subs(sqrt(T), Troot)
>>> new_T_base = expand(expand(new_disc)*Troot**15)
>>> ans = collect(new_T_base, Troot).args
```

Zc = 0.3333333333333333
Mechanical compressibility of *RK* EOS

a_alpha_and_derivatives_pure(*T*)
Method to calculate $a\alpha$ and its first and second derivatives for this EOS. Uses the set values of *a*.

$$a\alpha = \frac{a}{\sqrt{\frac{T}{T_c}}}$$

$$\frac{da\alpha}{dT} = -\frac{a}{2T\sqrt{\frac{T}{T_c}}}$$

$$\frac{d^2a\alpha}{dT^2} = \frac{3a}{4T^2\sqrt{\frac{T}{T_c}}}$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

a_alpha_pure(*T*)
Method to calculate $a\alpha$ for this EOS. Uses the set values of *a*.

$$a\alpha = \frac{a}{\sqrt{\frac{T}{T_c}}}$$

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

c1 = 0.4274802335403414
Full value of the constant in the *a* parameter

c2 = 0.08664034996495772
Full value of the constant in the *b* parameter

epsilon = 0.0
epsilon is always zero for the *RK* EOS

omega = None
omega has no impact on the *RK* EOS

solve_T(*P*, *V*, *solution=None*)

Method to calculate T from a specified P and V for the RK EOS. Uses a, and b, obtained from the class's namespace.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

Notes

The exact solution can be derived as follows; it is excluded for brevity.

```
>>> from sympy import *
>>> P, T, V, R = symbols('P, T, V, R')
>>> Tc, Pc = symbols('Tc, Pc')
>>> a, b = symbols('a, b')
>>> RK = Eq(P, R*T/(V-b) - a/sqrt(T)/(V*V + b*V))
>>> solve(RK, T)
```

7.7.7 Ideal Gas Equation of State

class thermo.eos.**IG**(*Tc=None*, *Pc=None*, *omega=None*, *T=None*, *P=None*, *V=None*)

Bases: *thermo.eos.GCEOS*

Class for solving the ideal gas equation in the *GCEOS* framework. This provides access to a number of derivatives and properties easily. It also keeps a common interface for all gas models. However, it is somewhat slow.

Subclasses *GCEOS*, which provides the methods for solving the EOS and calculating its assorted relevant thermodynamic properties. Solves the EOS on initialization.

Two of *T*, *P*, and *V* are needed to solve the EOS; values for *Tc* and *Pc* and *omega*, which are not used in the calculates, are set to those of methane by default to allow use without specifying them.

$$P = \frac{RT}{V}$$

Parameters

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

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

omega [float, optional] Acentric factor, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

References

[1]

Examples

T-P initialization, and exploring each phase's properties:

```
>>> eos = IG(T=400., P=1E6)
>>> eos.V_g, eos.phase
(0.0033257850472612 96, 'g')
>>> eos.H_dep_g, eos.S_dep_g, eos.U_dep_g, eos.G_dep_g, eos.A_dep_g
(0.0, 0.0, 0.0, 0.0, 0.0)
>>> eos.beta_g, eos.kappa_g, eos.Cp_dep_g, eos.Cv_dep_g
(0.0025, 1e-06, 0.0, 0.0)
>>> eos.fugacity_g, eos.PIP_g, eos.Z_g, eos.dP_dT_g
(1000000.0, 0.9999999999999999, 1.0, 2500.0)
```

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for this EOS.
a_alpha_pure(T)	Method to calculate $a\alpha$ for the ideal gas law, which is zero.
solve_T(P, V[, solution])	Method to calculate T from a specified P and V for the ideal gas equation of state.
volume_solutions(T, P[, b, delta, epsilon, ...])	Calculate the ideal-gas molar volume in a format compatible with the other cubic EOS solvers.

Zc = 1.0
float: Critical compressibility for an ideal gas is 1

a = 0.0
float: a parameter for an ideal gas is 0

a_alpha_and_derivatives_pure(*T*)
Method to calculate $a\alpha$ and its first and second derivatives for this EOS. All values are zero.

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

a_alpha_pure(*T*)
Method to calculate $a\alpha$ for the ideal gas law, which is zero.

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

b = 0.0

float: *b* parameter for an ideal gas is 0

delta = 0.0

float: *delta* parameter for an ideal gas is 0

epsilon = 0.0

float: *epsilon* parameter for an ideal gas is 0

solve_T(*P*, *V*, *solution=None*)

Method to calculate T from a specified P and V for the ideal gas equation of state.

$$T = \frac{PV}{R}$$

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

solution [str or None, optional] Not used, [-]

Returns

T [float] Temperature, [K]

static volume_solutions(*T*, *P*, *b=0.0*, *delta=0.0*, *epsilon=0.0*, *a_alpha=0.0*)

Calculate the ideal-gas molar volume in a format compatible with the other cubic EOS solvers. The ideal gas volume is the first element; and the secodn and third elements are zero. This is implemented to allow the ideal-gas model to be compatible with the cubic models, whose equations do not work with parameters of zero.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float, optional] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float, optional] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float, optional] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float, optional] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [list[float]] Three possible molar volumes, [m^3/mol]

Examples

```
>>> volume_solutions_ideal(T=300, P=1e7)
(0.0002494338785445972, 0.0, 0.0)
```

7.7.8 Lists of Equations of State

thermo.eos.**eos_list = [<class 'thermo.eos.IG'>, <class 'thermo.eos.PR'>, <class 'thermo.eos.PR78'>, <class 'thermo.eos.PRSV'>, <class 'thermo.eos.PRSV2'>, <class 'thermo.eos.VDW'>, <class 'thermo.eos.RK'>, <class 'thermo.eos.SRK'>, <class 'thermo.eos.APISRK'>, <class 'thermo.eos.TWUPR'>, <class 'thermo.eos.TWUSRK'>, <class 'thermo.eos.PRTranslatedPPJP'>, <class 'thermo.eos.SRKTranslatedPPJP'>, <class 'thermo.eos.MSRKTranslated'>, <class 'thermo.eos.PRTranslatedConsistent'>, <class 'thermo.eos.SRKTranslatedConsistent'>]**

list : List of all cubic equation of state classes.

thermo.eos.**eos_2P_list = [<class 'thermo.eos.PR'>, <class 'thermo.eos.PR78'>, <class 'thermo.eos.PRSV'>, <class 'thermo.eos.PRSV2'>, <class 'thermo.eos.VDW'>, <class 'thermo.eos.RK'>, <class 'thermo.eos.SRK'>, <class 'thermo.eos.APISRK'>, <class 'thermo.eos.TWUPR'>, <class 'thermo.eos.TWUSRK'>, <class 'thermo.eos.PRTranslatedPPJP'>, <class 'thermo.eos.SRKTranslatedPPJP'>, <class 'thermo.eos.MSRKTranslated'>, <class 'thermo.eos.PRTranslatedConsistent'>, <class 'thermo.eos.SRKTranslatedConsistent'>]**

list : List of all cubic equation of state classes that can represent multiple phases.

7.7.9 Demonstrations of Concepts

Maximum Pressure at Constant Volume

Some equations of state show this behavior. At a liquid volume, if the temperature is increased, the pressure should increase as well to create that same volume. However in some cases this is not the case as can be demonstrated for this hypothetical dodecane-like fluid:

Through experience, it is observed that this behavior is only shown for some sets of critical constants. It was found that if the expression for ${\frac{\partial P}{\partial T}}_V$ is set to zero, an analytical expression can be determined for exactly what that maximum pressure is. Some EOSs implement this function as *P_max_at_V*; those that don't, and fluids where there is no maximum pressure, will have that method but it will return None.

Debug Plots to Understand EOSs

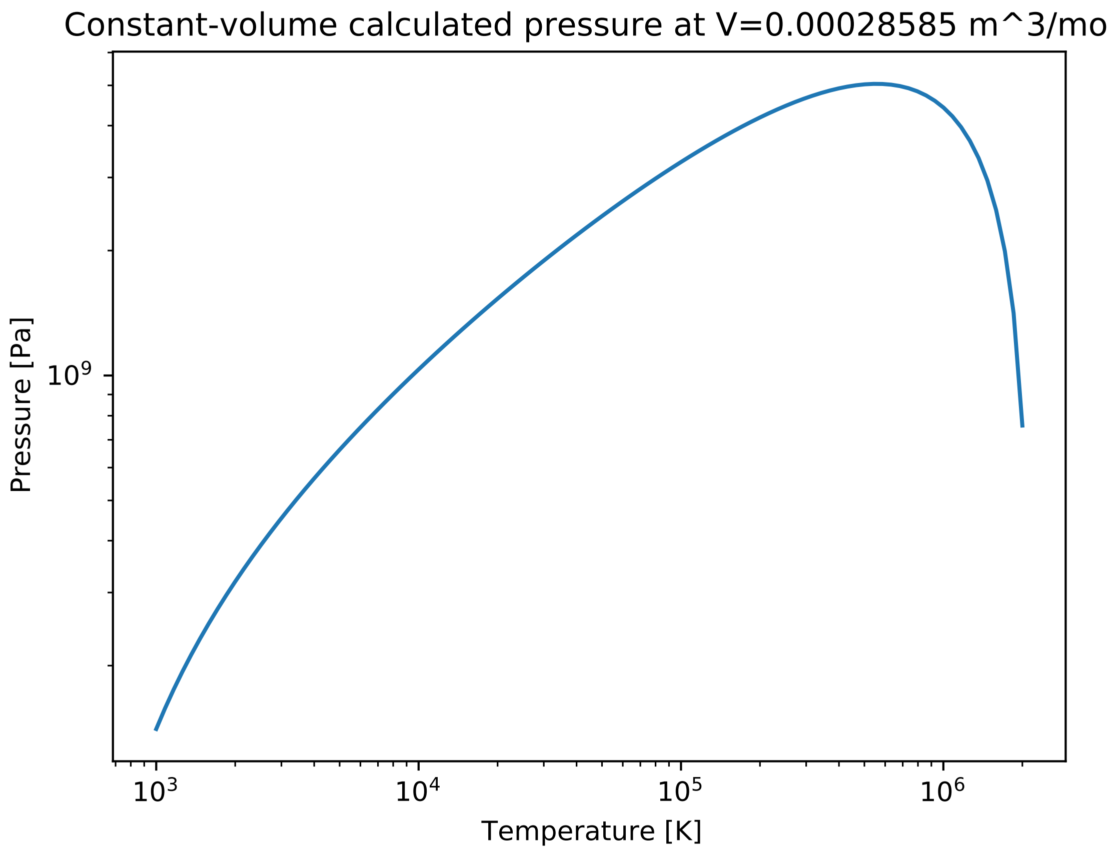
The *GCEOS.volume_errors* method shows the relative error in the volume solution. *mpmath* is requried for this functionality. It is not likely there is an error here but many problems have been found in the past.

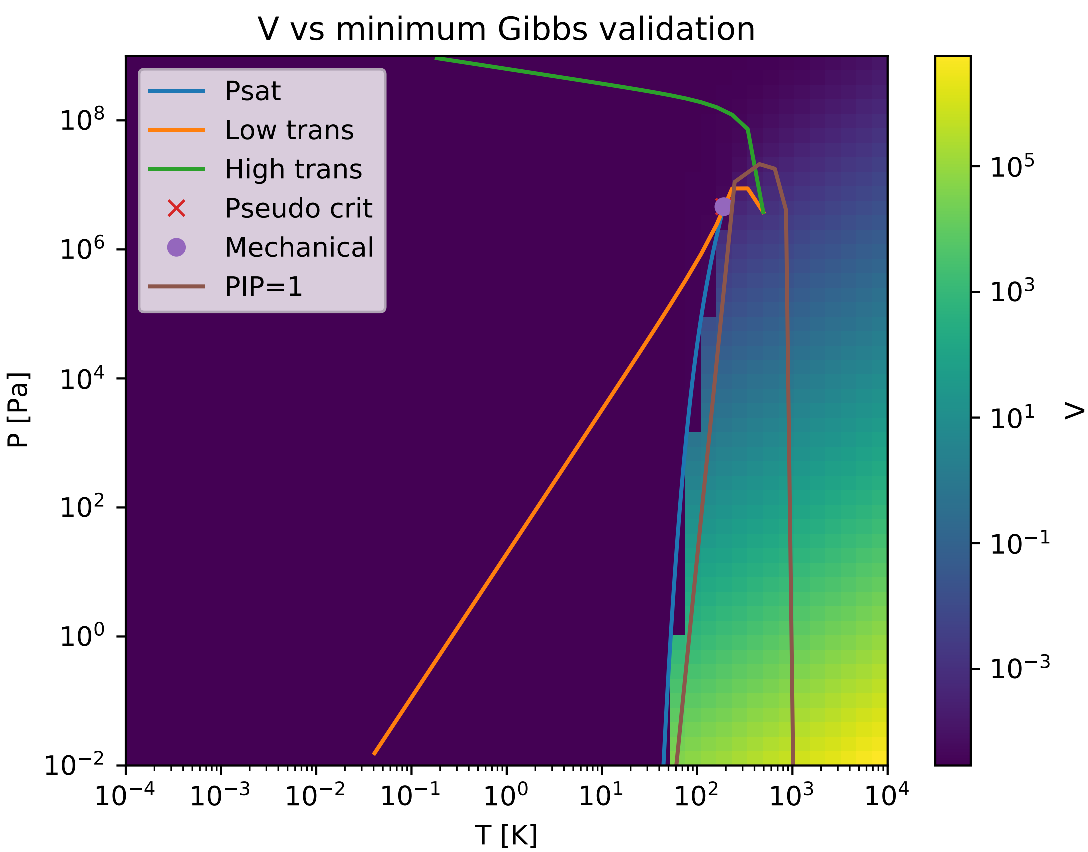
The *GCEOS.PT_surface_special* method shows some of the special curves of the EOS.

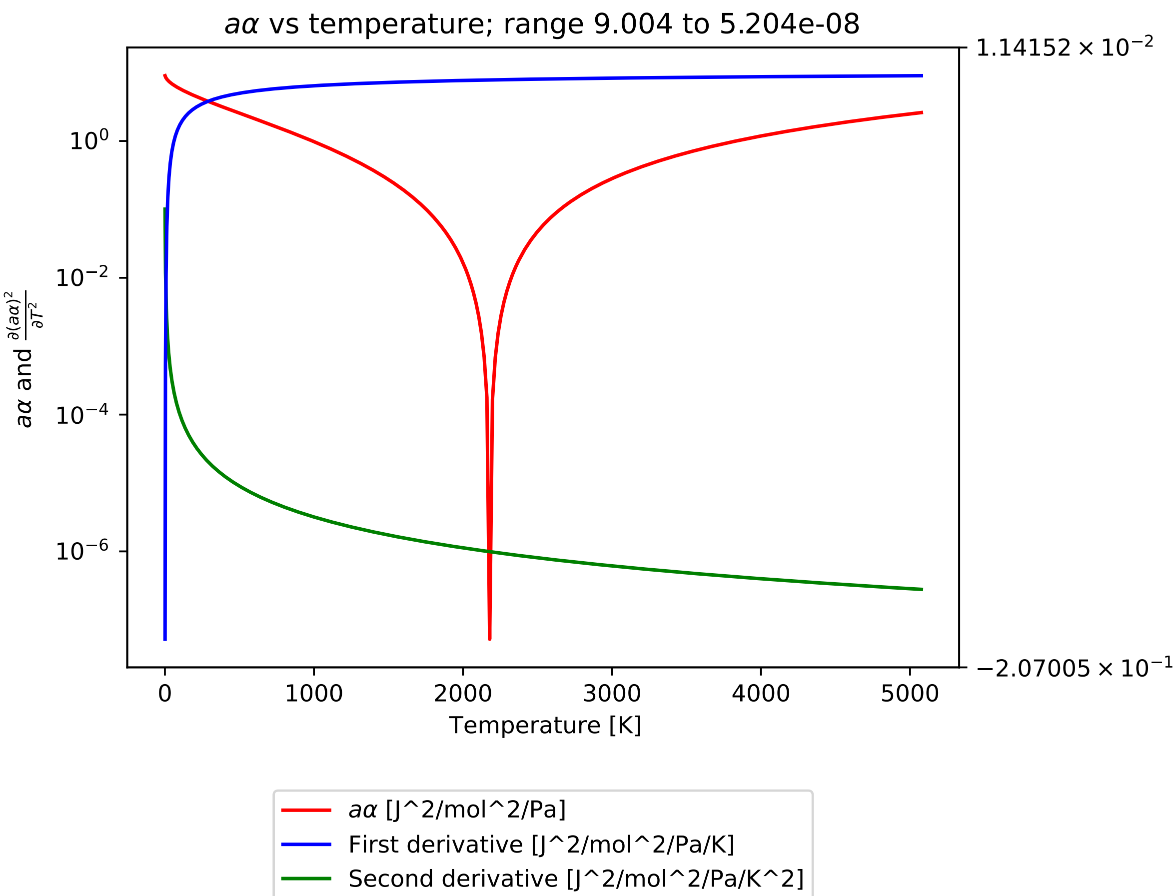
The *GCEOS.a_alpha_plot* method shows the alpha function curve. The following sample shows the SRK's default alpha function for methane.

If this doesn't look healthy, that is because it is not. There are strict thermodynamic consistency requirements that we know of today:

- The alpha function must be positive and continuous
- The first derivative must be negative and continuous
- The second derivative must be positive and continuous


Constant-volume calculated pressure at V=0.00028585 m^3/mol
Pressure [Pa]
10^9
10^3
10^4
10^5
10^6
Temperature [K]


V vs minimum Gibbs validation
Psat
Low trans
High trans
Pseudo crit
Mechanical
PIP=1
P [Pa]
T [K]
V
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{3}
10^{1}
10^{-1}
10^{-3}

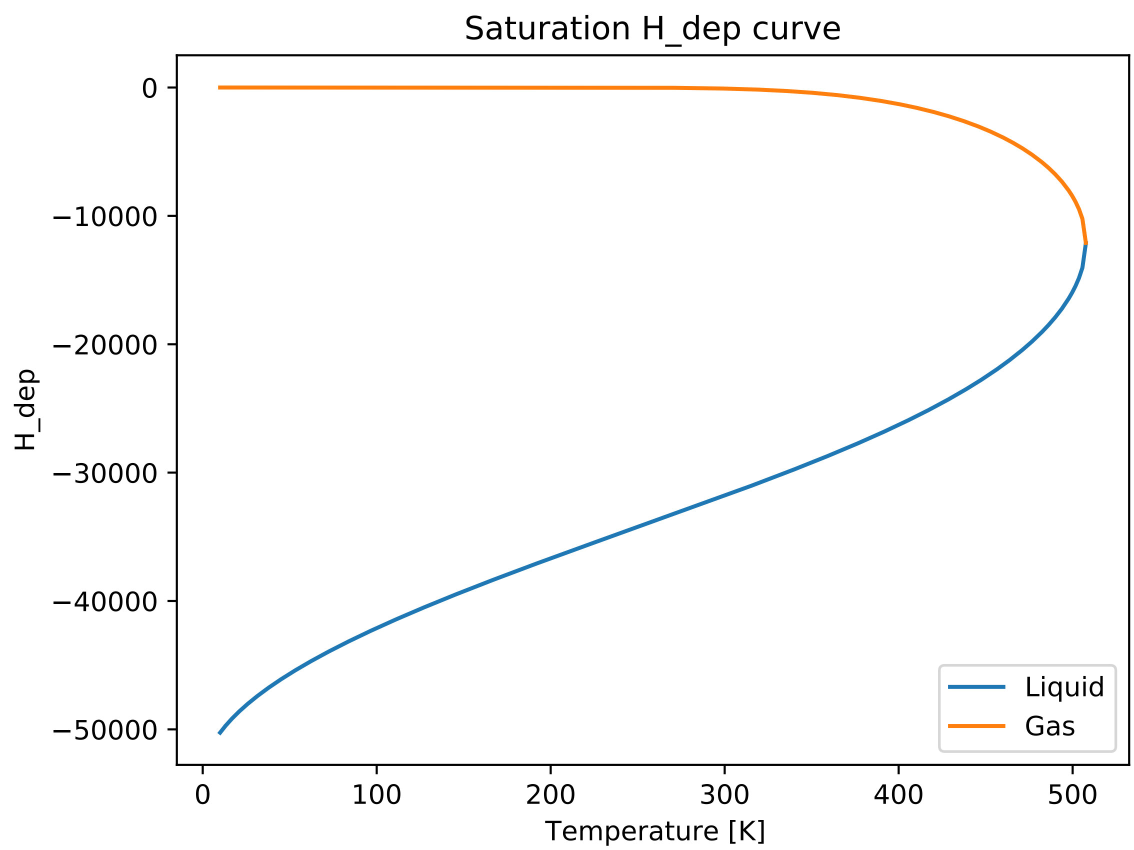


$a\alpha$ vs temperature; range 9.004 to 5.204e-08
$a\alpha$ and $\frac{\partial(a\alpha)^2}{\partial T^2}$
$\frac{\partial a\alpha}{\partial T}$
1.14152×10^{-2}
-2.07005×10^{-1}
10^{0}
10^{-2}
10^{-4}
10^{-6}
0
1000
2000
3000
4000
5000
Temperature [K]
$a\alpha$ [J^2/mol^2/Pa]
First derivative [J^2/mol^2/Pa/K]
Second derivative [J^2/mol^2/Pa/K^2]

- The third derivative must be negative

The first criterial and second criteria fail here.

There are two methods to review the saturation properties solution. The more general way is to review saturation properties as a plot:

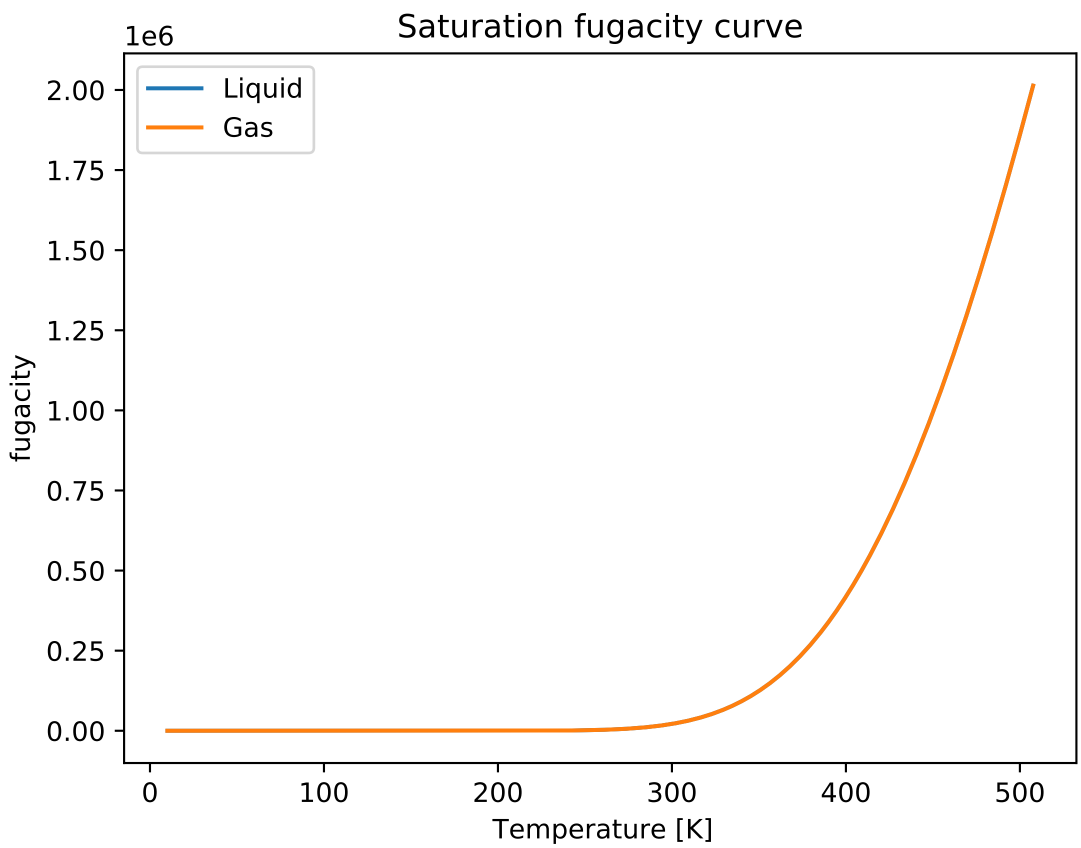


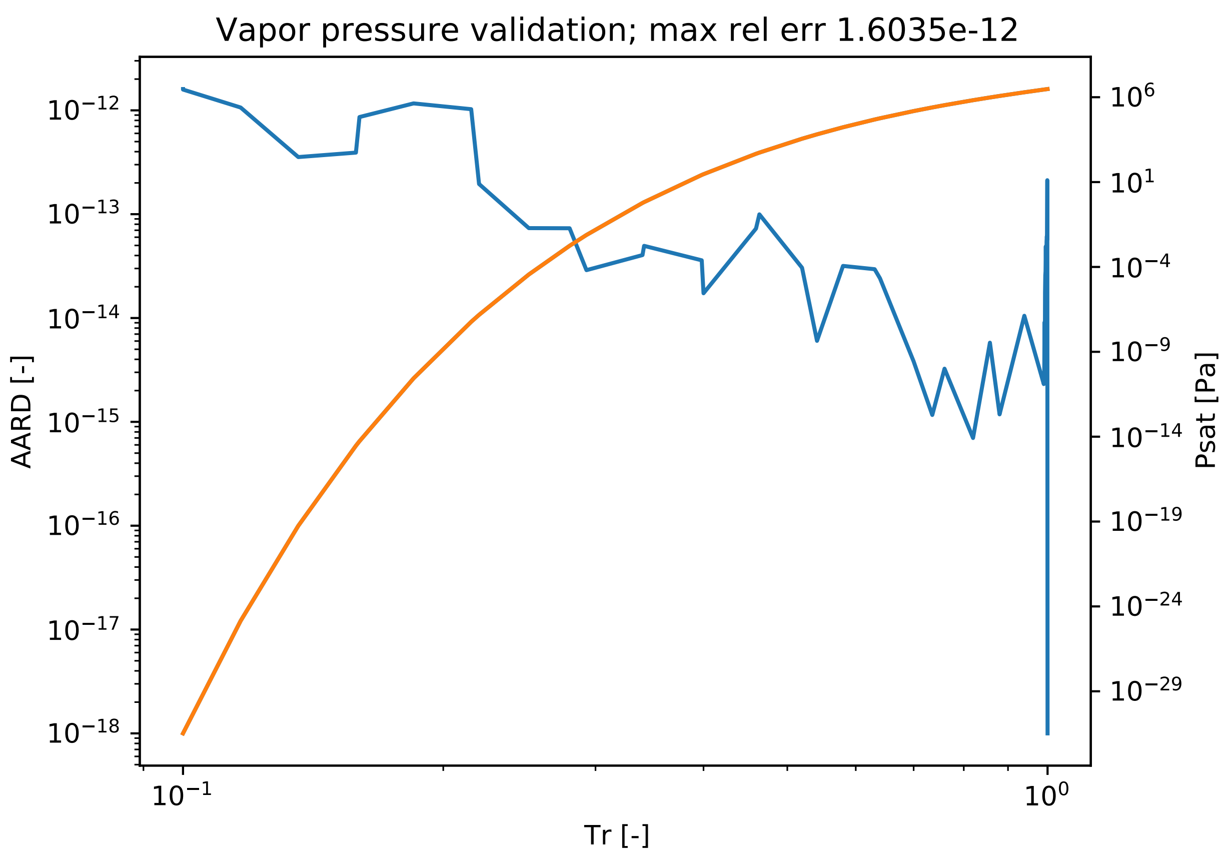
The second plot is more detailed, and is focused on the direct calculation of vapor pressure without using an iterative solution. It shows the relative error of the fit, which normally way below where it would present any issue - only 10-100x more error than it is possible to get with floating point numbers at all.

7.8 Cubic Equations of State for Mixtures (thermo.eos_mix)

This module contains implementations of most cubic equations of state for mixtures. This includes Peng-Robinson, SRK, Van der Waals, PRSV, TWU and many other variants.

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


Saturation fugacity curve
1e6
Liquid
Gas
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
fugacity
0
100
200
300
400
500
Temperature [K]



Vapor pressure validation; max rel err 1.6035e-12
AARD [-]
Psat [Pa]
Tr [-]
10^{-12}
10^{-13}
10^{-14}
10^{-15}
10^{-16}
10^{-17}
10^{-18}
10^{6}
10^{1}
10^{-4}
10^{-9}
10^{-14}
10^{-19}
10^{-24}
10^{-29}
10^{-1}
10^{0}

7.8.1 Base Class

class `thermo.eos_mix.GCEOSMIX`

Bases: *thermo.eos.GCEOS*

Class for solving a generic pressure-explicit three-parameter cubic equation of state for a mixture. Does not implement any parameters itself; must be subclassed by a mixture equation of state class which subclasses it.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V^2 + \delta V + \epsilon}$$

Attributes

A_dep_g Departure molar Helmholtz energy from ideal gas behavior for the gas phase, [J/mol].

A_dep_l Departure molar Helmholtz energy from ideal gas behavior for the liquid phase, [J/mol].

Cp_minus_Cv_g Cp - Cv for the gas phase, [J/mol/K].

Cp_minus_Cv_l Cp - Cv for the liquid phase, [J/mol/K].

U_dep_g Departure molar internal energy from ideal gas behavior for the gas phase, [J/mol].

U_dep_l Departure molar internal energy from ideal gas behavior for the liquid phase, [J/mol].

V_dep_g Departure molar volume from ideal gas behavior for the gas phase, [m^3/mol].

V_dep_l Departure molar volume from ideal gas behavior for the liquid phase, [m^3/mol].

V_g_mpmath The molar volume of the gas phase calculated with *mpmath* to a higher precision, [m^3/mol].

V_l_mpmath The molar volume of the liquid phase calculated with *mpmath* to a higher precision, [m^3/mol].

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

a_alpha_ijs Calculate and return the matrix $(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$.

beta_g Isobaric (constant-pressure) expansion coefficient for the gas phase, [1/K].

beta_l Isobaric (constant-pressure) expansion coefficient for the liquid phase, [1/K].

c1

c2

d2H_dep_dT2_g Second temperature derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K^2].

d2H_dep_dT2_g_P Second temperature derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K^2].

d2H_dep_dT2_g_V Second temperature derivative of departure enthalpy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^2].

d2H_dep_dT2_l Second temperature derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K^2].

d2H_dep_dT2_l_P Second temperature derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K^2].

d2H_dep_dT2_l_V Second temperature derivative of departure enthalpy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^2].

d2H_dep_dTdP_g Temperature and pressure derivative of departure enthalpy at constant pressure then temperature for the gas phase, [(J/mol)/K/Pa].

d2H_dep_dTdP_l Temperature and pressure derivative of departure enthalpy at constant pressure then temperature for the liquid phase, [(J/mol)/K/Pa].

d2P_dT2_PV_g Second derivative of pressure with respect to temperature twice, but with pressure held constant the first time and volume held constant the second time for the gas phase, [Pa/K^2].

d2P_dT2_PV_l Second derivative of pressure with respect to temperature twice, but with pressure held constant the first time and volume held constant the second time for the liquid phase, [Pa/K^2].

d2P_dTdP_g Second derivative of pressure with respect to temperature and, then pressure; and with volume held constant at first, then temperature, for the gas phase, [1/K].

d2P_dTdP_l Second derivative of pressure with respect to temperature and, then pressure; and with volume held constant at first, then temperature, for the liquid phase, [1/K].

d2P_dTdrho_g Derivative of pressure with respect to molar density, and temperature for the gas phase, [Pa/(K*mol/m^3)].

d2P_dTdrho_l Derivative of pressure with respect to molar density, and temperature for the liquid phase, [Pa/(K*mol/m^3)].

d2P_dVdP_g Second derivative of pressure with respect to molar volume and then pressure for the gas phase, [mol/m^3].

d2P_dVdP_l Second derivative of pressure with respect to molar volume and then pressure for the liquid phase, [mol/m^3].

d2P_dVdT_TP_g Second derivative of pressure with respect to molar volume and then temperature at constant temperature then pressure for the gas phase, [Pa*mol/m^3/K].

d2P_dVdT_TP_l Second derivative of pressure with respect to molar volume and then temperature at constant temperature then pressure for the liquid phase, [Pa*mol/m^3/K].

d2P_dVdT_g Alias of `GCEOS.d2P_dTdV_g`

d2P_dVdT_l Alias of `GCEOS.d2P_dTdV_l`

d2P_drho2_g Second derivative of pressure with respect to molar density for the gas phase, [Pa/(mol/m^3)^2].

d2P_drho2_l Second derivative of pressure with respect to molar density for the liquid phase, [Pa/(mol/m^3)^2].

d2S_dep_dT2_g Second temperature derivative of departure entropy with respect to temperature for the gas phase, [(J/mol)/K^3].

d2S_dep_dT2_g_V Second temperature derivative of departure entropy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^3].

d2S_dep_dT2_l Second temperature derivative of departure entropy with respect to temperature for the liquid phase, [(J/mol)/K^3].

d2S_dep_dT2_l_V Second temperature derivative of departure entropy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^3].

d2S_dep_dTdP_g Temperature and pressure derivative of departure entropy at constant pressure then temperature for the gas phase, [(J/mol)/K^2/Pa].

d2S_dep_dTdP_l Temperature and pressure derivative of departure entropy at constant pressure then temperature for the liquid phase, [(J/mol)/K^2/Pa].

d2T_dP2_g Second partial derivative of temperature with respect to pressure (constant volume) for the gas phase, [K/Pa^2].

d2T_dP2_l Second partial derivative of temperature with respect to pressure (constant temperature) for the liquid phase, [K/Pa^2].

d2T_dPdV_g Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the gas phase, [K*mol/(Pa*m^3)].

d2T_dPdV_l Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the liquid phase, [K*mol/(Pa*m^3)].

d2T_dPdrho_g Derivative of temperature with respect to molar density, and pressure for the gas phase, [K/(Pa*mol/m^3)].

d2T_dPdrho_l Derivative of temperature with respect to molar density, and pressure for the liquid phase, [K/(Pa*mol/m^3)].

d2T_dV2_g Second partial derivative of temperature with respect to volume (constant pressure) for the gas phase, [K*mol^2/m^6].

d2T_dV2_l Second partial derivative of temperature with respect to volume (constant pressure) for the liquid phase, [K*mol^2/m^6].

d2T_dVdP_g Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the gas phase, [K*mol/(Pa*m^3)].

d2T_dVdP_l Second partial derivative of temperature with respect to pressure (constant volume) and then volume (constant pressure) for the liquid phase, [K*mol/(Pa*m^3)].

d2T_drho2_g Second derivative of temperature with respect to molar density for the gas phase, [K/(mol/m^3)^2].

d2T_drho2_l Second derivative of temperature with respect to molar density for the liquid phase, [K/(mol/m^3)^2].

d2V_dP2_g Second partial derivative of volume with respect to pressure (constant temperature) for the gas phase, [m^3/(Pa^2*mol)].

d2V_dP2_l Second partial derivative of volume with respect to pressure (constant temperature) for the liquid phase, [m^3/(Pa^2*mol)].

d2V_dPdT_g Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the gas phase, [m^3/(K*Pa*mol)].

d2V_dPdT_l Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the liquid phase, [m^3/(K*Pa*mol)].

d2V_dT2_g Second partial derivative of volume with respect to temperature (constant pressure) for the gas phase, [m^3/(mol*K^2)].

d2V_dT2_l Second partial derivative of volume with respect to temperature (constant pressure) for the liquid phase, [m^3/(mol*K^2)].

d2V_dTdP_g Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the gas phase, [m^3/(K*Pa*mol)].

d2V_dTdP_l Second partial derivative of volume with respect to pressure (constant temperature) and then presssure (constant temperature) for the liquid phase, [m^3/(K*Pa*mol)].

d2a_alpha_dT2_dns Helper method for calculating the mole number derivatives of *d2a_alpha_dT2*.

d2a_alpha_dT2_dzs Helper method for calculating the mole number derivatives of *d2a_alpha_dT2*.

d2a_alpha_dT2_ijs Calculate and return the matrix of the second temperature derivatives of the alpha terms.

d2a_alpha_dTdP_g_V Derivative of the temperature derivative of *a_alpha* with respect to pressure at constant volume (varying T) for the gas phase, [J^2/mol^2/Pa^2/K].

d2a_alpha_dTdP_l_V Derivative of the temperature derivative of *a_alpha* with respect to pressure at constant volume (varying T) for the liquid phase, [J^2/mol^2/Pa^2/K].

d2a_alpha_dninjs Helper method for calculating the second partial molar derivatives of *a_alpha* (hessian).

d2a_alpha_dzizjs Helper method for calculating the second composition derivatives of *a_alpha* (hessian).

d2b_dninjs Helper method for calculating the second partial mole number derivatives of *b*.

d2b_dzizjs Helper method for calculating the second partial mole fraction derivatives of *b*.

d2rho_dP2_g Second derivative of molar density with respect to pressure for the gas phase, [(mol/m^3)/Pa^2].

d2rho_dP2_l Second derivative of molar density with respect to pressure for the liquid phase, [(mol/m^3)/Pa^2].

d2rho_dPdT_g Second derivative of molar density with respect to pressure and temperature for the gas phase, [(mol/m^3)/(K*Pa)].

d2rho_dPdT_l Second derivative of molar density with respect to pressure and temperature for the liquid phase, [(mol/m^3)/(K*Pa)].

d2rho_dT2_g Second derivative of molar density with respect to temperature for the gas phase, [(mol/m^3)/K^2].

d2rho_dT2_l Second derivative of molar density with respect to temperature for the liquid phase, [(mol/m^3)/K^2].

d3a_alpha_dT3 Method to calculate the third temperature derivative of $a\alpha$, [J^2/mol^2/Pa/K^3].

d3a_alpha_dninjnks Helper method for calculating the third mole number derivatives of *a_alpha*.

d3a_alpha_dzizjzks Helper method for calculating the third composition derivatives of *a_alpha*.

d3b_dninjnks Helper method for calculating the third partial mole number derivatives of *b*.

d3b_dzizjzks Helper method for calculating the third partial mole fraction derivatives of *b*.

d3delta_dzizjzks Helper method for calculating the third composition derivatives of *delta*.

d3epsilon_dzizjzks Helper method for calculating the third composition derivatives of *epsilon*.

dH_dep_dP_g Derivative of departure enthalpy with respect to pressure for the gas phase, [(J/mol)/Pa].

dH_dep_dP_g_V Derivative of departure enthalpy with respect to pressure at constant volume for the liquid phase, [(J/mol)/Pa].

dH_dep_dP_l Derivative of departure enthalpy with respect to pressure for the liquid phase, [(J/mol)/Pa].

dH_dep_dP_l_V Derivative of departure enthalpy with respect to pressure at constant volume for the gas phase, [(J/mol)/Pa].

dH_dep_dT_g Derivative of departure enthalpy with respect to temperature for the gas phase, [(J/mol)/K].

dH_dep_dT_g_V Derivative of departure enthalpy with respect to temperature at constant volume for the gas phase, [(J/mol)/K].

dH_dep_dT_l Derivative of departure enthalpy with respect to temperature for the liquid phase, [(J/mol)/K].

dH_dep_dT_l_V Derivative of departure enthalpy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K].

dH_dep_dV_g_P Derivative of departure enthalpy with respect to volume at constant pressure for the gas phase, [J/m^3].

dH_dep_dV_g_T Derivative of departure enthalpy with respect to volume at constant temperature for the gas phase, [J/m^3].

dH_dep_dV_l_P Derivative of departure enthalpy with respect to volume at constant pressure for the liquid phase, [J/m^3].

dH_dep_dV_l_T Derivative of departure enthalpy with respect to volume at constant temperature for the gas phase, [J/m^3].

dP_drho_g Derivative of pressure with respect to molar density for the gas phase, [Pa/(mol/m^3)].

dP_drho_l Derivative of pressure with respect to molar density for the liquid phase, [Pa/(mol/m^3)].

dS_dep_dP_g Derivative of departure entropy with respect to pressure for the gas phase, [(J/mol)/K/Pa].

dS_dep_dP_g_V Derivative of departure entropy with respect to pressure at constant volume for the gas phase, [(J/mol)/K/Pa].

dS_dep_dP_l Derivative of departure entropy with respect to pressure for the liquid phase, [(J/mol)/K/Pa].

dS_dep_dP_l_V Derivative of departure entropy with respect to pressure at constant volume for the liquid phase, [(J/mol)/K/Pa].

dS_dep_dT_g Derivative of departure entropy with respect to temperature for the gas phase, [(J/mol)/K^2].

dS_dep_dT_g_V Derivative of departure entropy with respect to temperature at constant volume for the gas phase, [(J/mol)/K^2].

dS_dep_dT_l Derivative of departure entropy with respect to temperature for the liquid phase, [(J/mol)/K^2].

dS_dep_dT_l_V Derivative of departure entropy with respect to temperature at constant volume for the liquid phase, [(J/mol)/K^2].

dS_dep_dV_g_P Derivative of departure entropy with respect to volume at constant pressure for the gas phase, [J/K/m^3].

dS_dep_dV_g_T Derivative of departure entropy with respect to volume at constant temperature for the gas phase, [J/K/m^3].

dS_dep_dV_l_P Derivative of departure entropy with respect to volume at constant pressure for the liquid phase, [J/K/m^3].

dS_dep_dV_l_T Derivative of departure entropy with respect to volume at constant temperature for the gas phase, [J/K/m^3].

dT_drho_g Derivative of temperature with respect to molar density for the gas phase, [K/(mol/m^3)].

dT_drho_l Derivative of temperature with respect to molar density for the liquid phase, [K/(mol/m^3)].

dZ_dP_g Derivative of compressibility factor with respect to pressure for the gas phase, [1/Pa].

dZ_dP_l Derivative of compressibility factor with respect to pressure for the liquid phase, [1/Pa].

dZ_dT_g Derivative of compressibility factor with respect to temperature for the gas phase, [1/K].

dZ_dT_l Derivative of compressibility factor with respect to temperature for the liquid phase, [1/K].

da_alpha_dP_g_V Derivative of the *a_alpha* with respect to pressure at constant volume (varying T) for the gas phase, [J^2/mol^2/Pa^2].

da_alpha_dP_l_V Derivative of the *a_alpha* with respect to pressure at constant volume (varying T) for the liquid phase, [J^2/mol^2/Pa^2].

da_alpha_dT_dns Helper method for calculating the mole number derivatives of *da_alpha_dT*.

da_alpha_dT_dzs Helper method for calculating the composition derivatives of *da_alpha_dT*.

da_alpha_dT_ijs Calculate and return the matrix for the temperature derivatives of the alpha terms.

da_alpha_dns Helper method for calculating the mole number derivatives of *a_alpha*.

da_alpha_dzs Helper method for calculating the composition derivatives of *a_alpha*.

db_dns Helper method for calculating the mole number derivatives of *b*.

db_dzs Helper method for calculating the composition derivatives of *b*.

dbeta_dP_g Derivative of isobaric expansion coefficient with respect to pressure for the gas phase, [1/(Pa*K)].

dbeta_dP_l Derivative of isobaric expansion coefficient with respect to pressure for the liquid phase, [1/(Pa*K)].

dbeta_dT_g Derivative of isobaric expansion coefficient with respect to temperature for the gas phase, [1/K^2].

dbeta_dT_l Derivative of isobaric expansion coefficient with respect to temperature for the liquid phase, [1/K^2].

dfugacity_dP_g Derivative of fugacity with respect to pressure for the gas phase, [-].

dfugacity_dP_l Derivative of fugacity with respect to pressure for the liquid phase, [-].

dfugacity_dT_g Derivative of fugacity with respect to temperature for the gas phase, [Pa/K].

dfugacity_dT_l Derivative of fugacity with respect to temperature for the liquid phase, [Pa/K].

dna_alpha_dT_dns Helper method for calculating the mole number derivatives of *da_alpha_dT*.

dna_alpha_dns Helper method for calculating the partial molar derivatives of *a_alpha*.

dnb_dns Helper method for calculating the partial molar derivative of *b*.

dphi_dP_g Derivative of fugacity coefficient with respect to pressure for the gas phase, [1/Pa].

dphi_dP_l Derivative of fugacity coefficient with respect to pressure for the liquid phase, [1/Pa].

dphi_dT_g Derivative of fugacity coefficient with respect to temperature for the gas phase, [1/K].

dphi_dT_l Derivative of fugacity coefficient with respect to temperature for the liquid phase, [1/K].

drho_dP_g Derivative of molar density with respect to pressure for the gas phase, [(mol/m^3)/Pa].

drho_dP_l Derivative of molar density with respect to pressure for the liquid phase, [(mol/m^3)/Pa].

drho_dT_g Derivative of molar density with respect to temperature for the gas phase, [(mol/m^3)/K].

drho_dT_l Derivative of molar density with respect to temperature for the liquid phase, [(mol/m^3)/K].

fugacity_g Fugacity for the gas phase, [Pa].

fugacity_l Fugacity for the liquid phase, [Pa].

kappa_g Isothermal (constant-temperature) expansion coefficient for the gas phase, [1/Pa].

kappa_l Isothermal (constant-temperature) expansion coefficient for the liquid phase, [1/Pa].

lnphi_g The natural logarithm of the fugacity coefficient for the gas phase, [-].

lnphi_l The natural logarithm of the fugacity coefficient for the liquid phase, [-].

more_stable_phase Checks the Gibbs energy of each possible phase, and returns 'l' if the liquid-like phase is more stable, and 'g' if the vapor-like phase is more stable.

mpmath_volume_ratios Method to compare, as ratios, the volumes of the implemented cubic solver versus those calculated using *mpmath*.

mpmath_volumes Method to calculate to a high precision the exact roots to the cubic equation, using *mpmath*.

mpmath_volumes_float Method to calculate real roots of a cubic equation, using *mpmath*, but returned as floats.

phi_g Fugacity coefficient for the gas phase, [Pa].

phi_l Fugacity coefficient for the liquid phase, [Pa].

pseudo_Pc Apply a linear mole-fraction mixing rule to compute the average critical pressure, [Pa].

pseudo_Tc Apply a linear mole-fraction mixing rule to compute the average critical temperature, [K].

pseudo_a Apply a linear mole-fraction mixing rule to compute the average *a* coefficient, [-].

pseudo_omega Apply a linear mole-fraction mixing rule to compute the average *omega*, [-].

rho_g Gas molar density, [mol/m^3].

rho_l Liquid molar density, [mol/m^3].

sorted_volumes List of lexicographically-sorted molar volumes available from the root finding algorithm used to solve the PT point.

state_specs Convenience method to return the two specified state specs (T, P, or V) as a dictionary.

Methods

Hvap(T)	Method to calculate enthalpy of vaporization for a pure fluid from an equation of state, without iteration.
PT_surface_special([Tmin, Tmax, Pmin, Pmax, ...])	Method to create a plot of the special curves of a fluid - vapor pressure, determinant zeros, pseudo critical point, and mechanical critical point.
P_PIP_transition(T[, low_P_limit])	Method to calculate the pressure which makes the phase identification parameter exactly 1.

continues on next page

Table 20 – continued from previous page

P_discriminant_zero_g()	Method to calculate the pressure which zero the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a vapor-like volume; and having a vapor-like volume.
P_discriminant_zero_l()	Method to calculate the pressure which zero the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a liquid-like volume; and having a liquid-like volume.
P_discriminant_zeros()	Method to calculate the pressures which zero the discriminant function of the general cubic eos, at the current temperature.
P_discriminant_zeros_analytical(T, b, delta, ...)	Method to calculate the pressures which zero the discriminant function of the general cubic eos.
P_max_at_V(V)	Dummy method.
Psat(T[, polish])	Generic method to calculate vapor pressure of a pure-component equation of state for a specified *T*.
Psat_errors([Tmin, Tmax, pts, plot, show, ...])	Method to create a plot of vapor pressure and the relative error of its calculation vs.
T_discriminant_zero_g([T_guess])	Method to calculate the temperature which zeros the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a vapor-like volume; and having a vapor-like volume.
T_discriminant_zero_l([T_guess])	Method to calculate the temperature which zeros the discriminant function of the general cubic eos, and is likely to sit on a boundary between not having a liquid-like volume; and having a liquid-like volume.
T_max_at_V(V[, Pmax])	Method to calculate the maximum temperature the EOS can create at a constant volume, if one exists; returns None otherwise.
T_min_at_V(V[, Pmin])	Returns the minimum temperature for the EOS to have the volume as specified.
Tsat(P[, polish])	Generic method to calculate the temperature for a specified vapor pressure of the pure fluid.
V_g_sat(T)	Method to calculate molar volume of the vapor phase along the saturation line.
V_l_sat(T)	Method to calculate molar volume of the liquid phase along the saturation line.
Vs_mpmath()	Method to calculate real roots of a cubic equation, using *mpmath*.
a_alpha_and_derivatives(T[, full, quick, ...])	Method to calculate *a_alpha* and its first and second derivatives for an EOS with the Van der Waals mixing rules.
a_alpha_and_derivatives_pure(T)	Dummy method to calculate $a\alpha$ and its first and second derivatives.
a_alpha_for_Psat(T, Psat[, a_alpha_guess])	Method to calculate which value of $a\alpha$ is required for a given *T*, *Psat* pair.
a_alpha_for_V(T, P, V)	Method to calculate which value of $a\alpha$ is required for a given *T*, *P* pair to match a specified *V*.
a_alpha_plot([Tmin, Tmax, pts, plot, show])	Method to create a plot of the $a\alpha$ parameter and its first two derivatives.

continues on next page

Table 20 – continued from previous page

as_json()	Method to create a JSON-friendly serialization of the eos which can be stored, and reloaded later.
check_sufficient_inputs()	Method to an exception if none of the pairs (T, P), (T, V), or (P, V) are given.
d2G_dep_dninjs(Z)	Calculates the molar departure Gibbs energy mole number derivatives (where the mole fractions sum to 1).
d2G_dep_dzizjs(Z)	Calculates the molar departure Gibbs energy second composition derivative (where the mole fractions do not sum to 1).
d2V_dninjs(Z)	Calculates the molar volume second mole number derivatives (where the mole fractions sum to 1).
d2V_dzizjs(Z)	Calculates the molar volume second composition derivative (where the mole fractions do not sum to 1).
d2lnphi_dninjs(Z)	Calculates the mixture log *fugacity coefficient* second mole number derivatives (where the mole fraction sum to 1).
d2lnphi_dzizjs(Z)	Calculates the mixture log *fugacity coefficient* second mole fraction derivatives (where the mole fractions do not sum to 1).
d2phi_sat_dT2(T[, polish])	Method to calculate the second temperature derivative of saturation fugacity coefficient of the compound.
dG_dep_dns(Z)	Calculates the molar departure Gibbs energy mole number derivatives (where the mole fractions sum to 1).
dG_dep_dzs(Z)	Calculates the molar departure Gibbs energy composition derivative (where the mole fractions do not sum to 1).
dH_dep_dT_sat_g(T[, polish])	Method to calculate and return the temperature derivative of saturation vapor excess enthalpy.
dH_dep_dT_sat_l(T[, polish])	Method to calculate and return the temperature derivative of saturation liquid excess enthalpy.
dH_dep_dns(Z)	Calculates the molar departure enthalpy mole number derivatives (where the mole fractions sum to 1).
dH_dep_dzs(Z)	Calculates the molar departure enthalpy composition derivative (where the mole fractions do not sum to 1).
dPsat_dT(T[, polish, also_Psat])	Generic method to calculate the temperature derivative of vapor pressure for a specified *T*.
dS_dep_dT_sat_g(T[, polish])	Method to calculate and return the temperature derivative of saturation vapor excess entropy.
dS_dep_dT_sat_l(T[, polish])	Method to calculate and return the temperature derivative of saturation liquid excess entropy.
dS_dep_dns(Z)	Calculates the molar departure entropy mole number derivatives (where the mole fractions sum to 1).
dS_dep_dzs(Z)	Calculates the molar departure entropy composition derivative (where the mole fractions do not sum to 1).
dV_dns(Z)	Calculates the molar volume mole number derivatives (where the mole fractions sum to 1).

continues on next page

Table 20 – continued from previous page

dV_dzs(Z)	Calculates the molar volume composition derivative (where the mole fractions do not sum to 1).
dZ_dns(Z)	Calculates the compressibility mole number derivatives (where the mole fractions sum to 1).
dZ_dzs(Z)	Calculates the compressibility composition derivatives (where the mole fractions do not sum to 1).
dfugacities_dns(phase)	Generic formula for calculating the mole number derivaitves of fugacities for each species in a mixture.
discriminant([T, P])	Method to compute the discriminant of the cubic volume solution with the current EOS parameters, optionally at the same (assumed) *T*, and *P* or at different ones, if values are specified.
dlnfugacities_dns(phase)	Generic formula for calculating the mole number derivaitves of log fugacities for each species in a mixture.
dlnphi_dns(Z)	Calculates the mixture log *fugacity coefficient* mole number derivatives (where the mole fractions sum to 1).
dlnphi_dzs(Z)	Calculates the mixture log *fugacity coefficient* mole fraction derivatives (where the mole fractions do not sum to 1).
dlnphis_dP(phase)	Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture.
dlnphis_dT(phase)	Generic formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture.
dlnphis_dns(Z)	Generic formula for calculating the mole number derivaitves of log fugacity coefficients for each species in a mixture.
dlnphis_dzs(Z)	Generic formula for calculating the mole fraction derivaitves of log fugacity coefficients for each species in a mixture.
dnG_dep_dns(Z)	Calculates the partial molar departure Gibbs energy.
dnH_dep_dns(Z)	Calculates the partial molar departure enthalpy.
dnV_dns(Z)	Calculates the partial molar volume of the specified phase No specific formula is implemented for this property - it is calculated from the molar volume mole fraction derivative.
dnZ_dns(Z)	Calculates the partial compressibility of the specified phase No specific formula is implemented for this property - it is calculated from the compressibility mole fraction derivative.
dphi_sat_dT(T[, polish])	Method to calculate the temperature derivative of saturation fugacity coefficient of the compound.
from_json(json_repr)	Method to create a mixture cubic equation of state from a JSON friendly serialization of another mixture cubic equation of state.

continues on next page

Table 20 – continued from previous page

fugacities([only_l, only_g])	Helper method for calculating fugacity coefficients for any phases present, using either the overall mole fractions for both phases or using specified mole fractions for each phase.
fugacity_coefficients(Z)	Generic formula for calculating log fugacity coefficients for each species in a mixture.
mechanical_critical_point()	Method to calculate the mechanical critical point of a mixture of defined composition.
model_hash()	Basic method to calculate a hash of the non-state parts of the model This is useful for comparing to models to determine if they are the same, i.e. in a VLL flash it is important to know if both liquids have the same model.
phi_sat(T[, polish])	Method to calculate the saturation fugacity coefficient of the compound.
pures()	Helper method which returns a list of pure *EOSs* at the same *T* and *P* and base EOS as the mixture.
resolve_full_alphas()	Generic method to resolve the eos with fully calculated alpha derviatives.
saturation_prop_plot(prop[, Tmin, Tmax, ...])	Method to create a plot of a specified property of the EOS along the (pure component) saturation line.
set_dnzs_derivatives_and_departures([n, x, ...])	Sets a number of mole number and/or composition partial derivatives of thermodynamic partial derivatives.
set_from_PT(Vs[, only_l, only_g])	Counts the number of real volumes in *Vs*, and determines what to do.
set_properties_from_solution(T, P, V, b, ...)	Sets all interesting properties which can be calculated from an EOS alone.
solve([pure_a_alphas, only_l, only_g, ...])	First EOS-generic method; should be called by all specific EOSs.
solve_T(P, V[, quick, solution])	Generic method to calculate *T* from a specified *P* and *V*.
solve_missing_volumes()	Generic method to ensure both volumes, if solutions are physical, have calculated properties.
state_hash()	Basic method to calculate a hash of the state of the model and its model parameters.
subset(idxs, **state_specs)	Method to construct a new GCEOSMIX that removes all components not specified in the *idxs* argument.
to([zs, T, P, V, fugacities])	Method to construct a new GCEOSMIX object at two of *T*, *P* or *V* with the specified composition.
to_PV(P, V)	Method to construct a new GCEOSMIX object at the spcified *P* and *V* with the current composition.
to_PV_zs(P, V, zs[, fugacities, only_l, only_g])	Method to construct a new GCEOSMIX instance at *P*, *V*, and *zs* with the same parameters as the existing object.
to_TP(T, P)	Method to construct a new GCEOSMIX object at the spcified *T* and *P* with the current composition.
to_TPV_pure(i[, T, P, V])	Helper method which returns a pure *EOSs* at the specs (two of *T*, *P* and *V*) and base EOS as the mixture for a particular index.

continues on next page

Table 20 – continued from previous page

to_TP_zs(T, P, zs[, fugacities, only_l, only_g])	Method to construct a new *GCEOSMIX* instance at *T*, *P*, and *zs* with the same parameters as the existing object.
to_TP_zs_fast(T, P, zs[, only_l, only_g, ...])	Method to construct a new *GCEOSMIX* instance with the same parameters as the existing object.
to_TV(T, V)	Method to construct a new *GCEOSMIX* object at the spcified *T* and *V* with the current composition.
to_mechanical_critical_point()	Method to construct a new *GCEOSMIX* object at the current object's properties and composition, but which is at the mechanical critical point.
volume_error()	Method to calculate the relative absolute error in the calculated molar volumes.
volume_errors([Tmin, Tmax, Pmin, Pmax, pts, ...])	Method to create a plot of the relative absolute error in the cubic volume solution as compared to a higher-precision calculation.
volume_solutions(T, P, b, delta, epsilon, ...)	Halley's method based solver for cubic EOS volumes based on the idea of initializing from a single liquid-like guess which is solved precisely, deflating the cubic analytically, solving the quadratic equation for the next two volumes, and then performing two halley steps on each of them to obtain the final solutions.
volume_solutions_full(T, P, b, delta, ...[, ...])	Newton-Raphson based solver for cubic EOS volumes based on the idea of initializing from an analytical solver.
volume_solutions_mp(T, P, b, delta, epsilon, ...)	Solution of this form of the cubic EOS in terms of volumes, using the *mpmath* arbitrary precision library.

stabiliy_iteration_Michelsen	

Psat(*T*, *polish=False*)

Generic method to calculate vapor pressure of a pure-component equation of state for a specified *T*. An explicit solution is used unless *polish* is True.

The result of this function has no physical meaning for multicomponent mixtures, and does not represent either a dew point or a bubble point!

Parameters

T [float] Temperature, [K]

polish [bool, optional] Whether to attempt to use a numerical solver to make the solution more precise or not

Returns

Psat [float] Vapor pressure using the pure-component approach, [Pa]

Notes

For multicomponent mixtures this may serve as a useful guess for the dew and the bubble pressure.

a_alpha_and_derivatives(*T*, *full=True*, *quick=True*, *pure_a_alphas=True*)

Method to calculate *a_alpha* and its first and second derivatives for an EOS with the Van der Waals mixing rules. Uses the parent class's interface to compute pure component values. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*.

For use in `solve_T` this returns only *a_alpha* if *full* is False.

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

Parameters

T [float] Temperature, [K]

full [bool, optional] If False, calculates and returns only *a_alpha*

quick [bool, optional] Only the quick variant is implemented; it is little faster anyhow

pure_a_alphas [bool, optional] Whether or not to recalculate the a_alpha terms of pure components (for the case of mixtures only) which stay the same as the composition changes (i.e in a PT flash), [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

Notes

The exact expressions can be obtained with the following SymPy expression below, commented out for brevity.

```
>>> from sympy import *
>>> kij, T = symbols('kij, T ')
>>> a_alpha_i, a_alpha_j = symbols('a_alpha_i, a_alpha_j', cls=Function)
>>> a_alpha_ij = (1-kij)*sqrt(a_alpha_i(T)*a_alpha_j(T))
>>> diff(a_alpha_ij, T)
>>> diff(a_alpha_ij, T, T)
```

property a_alpha_ijs

Calculate and return the matrix $(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$.

Returns

a_alpha_ijs [list[list[float]]] *a_alpha* terms for each component with every other component, [J^2/mol^2/Pa]

Notes

In an earlier implementation this matrix was stored each EOS solve; however, allocating that much memory becomes quite expensive for large number of component cases and this is now calculated on-demand only.

d2G_dep_dninjs(Z)

Calculates the molar departure Gibbs energy mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial^2 G_{dep}}{\partial n_j \partial n_i}\right)_{T,P,n_{i,j\ne k}} = f\left(\left(\frac{\partial^2 G_{dep}}{\partial x_j \partial x_i}\right)_{T,P,x_{i,j\ne k}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

d2G_dep_dninjs [float] Departure Gibbs energy second mole number derivatives, [J/mol^3]

d2G_dep_dzizjs(Z)

Calculates the molar departure Gibbs energy second composition derivative (where the mole fractions do not sum to 1). Verified numerically. Useful in solving for gibbs minimization calculations or for solving for the true critical point. Also forms the basis for the molar departure Gibbs energy mole second number derivative.

$$\left(\frac{\partial^2 G_{dep}}{\partial x_j \partial x_i}\right)_{T,P,x_{i,j\ne k}} = \text{run SymPy code to obtain - very long!}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

d2G_dep_dzizjs [float] Departure Gibbs free energy second composition derivatives, [J/mol]

Notes

The derivation for the derivative is performed as follows using SymPy. The function source code is an optimized variant created with the *cse* SymPy function, and hand optimized further.

```
>>> from sympy import *
>>> P, T, R, x1, x2 = symbols('P, T, R, x1, x2')
>>> a_alpha, delta, epsilon, V, b = symbols('a\ \\alpha, delta, epsilon, V, b',␣
↪cls=Function)
>>> da_alpha_dT, d2a_alpha_dT2 = symbols('da_alpha_dT, d2a_alpha_dT2',␣
↪cls=Function)
>>> S_dep = R*log(P*V(x1, x2)/(R*T)) + R*log(V(x1, x2)-b(x1, x2))+2*da_alpha_
↪dT(x1, x2)*atanh((2*V(x1, x2)+delta(x1, x2))/sqrt(delta(x1, x2)**2-
↪4*epsilon(x1, x2)))/sqrt(delta(x1, x2)**2-4*epsilon(x1, x2))-R*log(V(x1, x2))
>>> H_dep = P*V(x1, x2) - R*T + 2*atanh((2*V(x1, x2)+delta(x1, x2))/
↪sqrt(delta(x1, x2)**2-4*epsilon(x1, x2)))*(da_alpha_dT(x1, x2)*T-a_alpha(x1,␣
↪x2))/sqrt(delta(x1, x2)**2-4*epsilon(x1, x2))
>>> G_dep = simplify(H_dep - T*S_dep)
>>> diff(G_dep, x1, x2)
```

d2V_dninjs(*Z*)

Calculates the molar volume second mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the second mole fraction derivatives.

$$\left(\frac{\partial^2 V}{\partial n_i \partial n_j}\right)_{T,P,n_{k\neq i,j}} = f\left(\left(\frac{\partial^2 V}{\partial x_i \partial x_j}\right)_{T,P,x_{k\neq i,j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

d2V_dninjs [float] Molar volume second mole number derivatives, [m^3/mol^3]

d2V_dzizjs(*Z*)

Calculates the molar volume second composition derivative (where the mole fractions do not sum to 1). Verified numerically. Used in many other derivatives, and for the molar volume second mole number derivative.

$$\left(\frac{\partial^2 V}{\partial x_i \partial x_j}\right)_{T,P,x_{k\neq i,j}} = \text{run SymPy code to obtain - very long!}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

d2V_dzizjs [float] Molar volume second composition derivatives, [m^3/mol]

Notes

The derivation for the derivative is performed as follows using SymPy. The function source code is an optimized variant created with the *cse* SymPy function, and hand optimized further.

```
>>> from sympy import *
>>> P, T, R, x1, x2 = symbols('P, T, R, x1, x2')
>>> V, delta, epsilon, a_alpha, b = symbols('V, delta, epsilon, a\ \\alpha, b',␣
↪cls=Function)
>>> CUBIC = R*T/(V(x1, x2) - b(x1, x2)) - a_alpha(x1, x2)/(V(x1, x2)*V(x1, x2)␣
↪+ delta(x1, x2)*V(x1, x2) + epsilon(x1, x2)) - P
>>> solve(diff(CUBIC, x1, x2), Derivative(V(x1, x2), x1, x2))
```

property d2a_alpha_dT2_dns

Helper method for calculating the mole number derivatives of *d2a_alpha_dT2*. Note this is independent of the phase.

$$\left(\frac{\partial^3 a\alpha}{\partial n_i \partial T^2}\right)_{P,n_{i\neq j}} = f\left(\left(\frac{\partial^3 a\alpha}{\partial z_i \partial T^2}\right)_{P,z_{i\neq j}}\right)$$

Returns

d2a_alpha_dT2_dns [list[float]] Mole number derivative of *d2a_alpha_dT2* of each component, [kg*m^5/(mol^3*s^2*K^2)]

Notes

This derivative is checked numerically.

property d2a_alpha_dT2_dzs

Helper method for calculating the mole number derivatives of *d2a_alpha_dT2*. Note this is independent of the phase.

$$\left(\frac{\partial^3 a\alpha}{\partial z_i \partial T^2}\right)_{P, z_{i\ne j}} = \text{large expression}$$

Returns

d2a_alpha_dT2_dzs [list[float]] Composition derivative of *d2a_alpha_dT2* of each component, [kg*m^5/(mol^2*s^2*K^2)]

Notes

This derivative is checked numerically.

property d2a_alpha_dT2_ijs

Calculate and return the matrix of the second temperature derivatives of the alpha terms.

$$\frac{\partial^2 (a\alpha)_{ij}}{\partial T^2} = -\frac{\sqrt{\mathrm{a\alpha_i}(T)\,\mathrm{a\alpha_j}(T)}\,(k_{ij}-1)\left(\frac{\left(\mathrm{a\alpha_i}(T)\frac{d}{dT}\mathrm{a\alpha_j}(T) + \mathrm{a\alpha_j}(T)\frac{d}{dT}\mathrm{a\alpha_i}(T)\right)^2}{4\,\mathrm{a\alpha_i}(T)\,\mathrm{a\alpha_j}(T)} - \frac{\left(\mathrm{a\alpha_i}(T)\frac{d}{dT}\mathrm{a\alpha_j}(T) + \mathrm{a\alpha_j}(T)\frac{d}{dT}\mathrm{a\alpha_i}(T)\right)\frac{d}{dT}}{2\,\mathrm{a\alpha_j}(T)}\right.}{a}$$

Returns

d2a_alpha_dT2_ijs [list[list[float]]] Second temperature derivative of *a_alpha* terms for each component with every other component, [J^2/mol^2/Pa/K^2]

Notes

In an earlier implementation this matrix was stored each EOS solve; however, allocating that much memory becomes quite expensive for large number of component cases and this is now calculated on-demand only.

property d2a_alpha_dninjs

Helper method for calculating the second partial molar derivatives of *a_alpha* (hessian). Note this is independent of the phase.

$$\left(\frac{\partial^2 a\alpha}{\partial n_i \partial n_j}\right)_{T, P, n_{k\ne i,j}} = 2\left[3(a\alpha) + (a\alpha)_{ij} - 2(\text{term}_{i,j})\right]$$

$$\text{term}_{i,j} = \sum_k z_k\left((a\alpha)_{ik} + (a\alpha)_{jk}\right)$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

Returns

d2a_alpha_dninjs [list[float]] Second partial molar derivative of *alpha* of each component, [kg*m^5/(mol^4*s^2)]

Notes

This derivative is checked numerically.

property d2a_alpha_dzizjs

Helper method for calculating the second composition derivatives of *a_alpha* (hessian). Note this is independent of the phase.

$$\left(\frac{\partial^2 a\alpha}{\partial x_i \partial x_j}\right)_{T,P,x_{k\ne i,j}} = 2(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

Returns

d2a_alpha_dzizjs [list[float]] Second composition derivative of *alpha* of each component, [kg*m^5/(mol^2*s^2)]

Notes

This derivative is checked numerically.

property d2b_dninjs

Helper method for calculating the second partial mole number derivatives of *b*. Note this is independent of the phase.

$$\left(\frac{\partial^2 b}{\partial n_i \partial n_j}\right)_{T,P,n_{k\ne i,k}} = 2b - b_i - b_j$$

Returns

d2b_dninjs [list[list[float]]] Second Composition derivative of *b* of each component, [m^3/mol^3]

Notes

This derivative is checked numerically.

property d2b_dzizjs

Helper method for calculating the second partial mole fraction derivatives of *b*. Note this is independent of the phase.

$$\left(\frac{\partial^2 b}{\partial x_i \partial x_j}\right)_{T,P,n_{k\ne i,j}} = 0$$

Returns

d2b_dzizjs [list[list[float]]] Second mole fraction derivatives of *b* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

`d2lnphi_dninjs(Z)`

Calculates the mixture log *fugacity coefficient* second mole number derivatives (where the mole fraction sum to 1). No specific formula is implemented for this property - it is calculated from the second mole fraction derivative of Gibbs free energy.

$$\left(\frac{\partial^2 \ln \phi}{\partial n_i \partial n_j}\right)_{T,P,n_{i,j\ne k}} f\left(\left(\frac{\partial^2 G_{dep}}{\partial x_j \partial x_i}\right)_{T,P,x_{i,j\ne k}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

d2lnphi_dninjs [float] Mixture log fugacity coefficient second mole number derivatives, [-]

`d2lnphi_dzizjs(Z)`

Calculates the mixture log *fugacity coefficient* second mole fraction derivatives (where the mole fractions do not sum to 1). No specific formula is implemented for this property - it is calculated from the second mole fraction derivative of Gibbs free energy.

$$\left(\frac{\partial^2 \ln \phi}{\partial x_i \partial x_j}\right)_{T,P,x_{i,j\ne k}} = \frac{1}{RT}\left(\left(\frac{\partial^2 G_{dep}}{\partial x_j \partial x_i}\right)_{T,P,x_{i,j\ne k}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

d2lnphi_dzizjs [float] Mixture log fugacity coefficient second mole fraction derivatives, [-]

`property d3a_alpha_dninjnks`

Helper method for calculating the third mole number derivatives of *a_alpha*. Note this is independent of the phase.

$$\left(\frac{\partial^3 a\alpha}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\ne i,j,k}} = 4\left(-6(a\alpha) - [(a\alpha)_{i,j} + (a\alpha)_{i,k} + (a\alpha)_{j,k}] + 3\sum_m z_m[(a\alpha)_{i,m} + (a\alpha)_{j,m} + (a\alpha)_{k,m}]\right)$$

Returns

d3a_alpha_dninjnks [list[float]] Third mole number derivative of *alpha* of each component, [kg*m^5/(mol^5*s^2)]

Notes

This derivative is checked numerically.

`property d3a_alpha_dzizjzks`

Helper method for calculating the third composition derivatives of *a_alpha*. Note this is independent of the phase.

$$\left(\frac{\partial^3 a\alpha}{\partial x_i \partial x_j \partial x_k}\right)_{T,P,x_{m\ne i,j,k}} = 0$$

Returns

d3a_alpha_dzizjzks [list[float]] Third composition derivative of *alpha* of each component, [kg*m^5/(mol^2*s^2)]

Notes

This derivative is checked numerically.

`property d3b_dninjnks`

Helper method for calculating the third partial mole number derivatives of *b*. Note this is independent of the phase.

$$\left(\frac{\partial^3 b}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\ne i,j,k}} = 2(-3b + b_i + b_j + b_k)$$

Returns

d3b_dninjnks [list[list[list[float]]]] Third mole number derivative of *b* of each component, [m^3/mol^4]

Notes

This derivative is checked numerically.

`property d3b_dzizjzks`

Helper method for calculating the third partial mole fraction derivatives of *b*. Note this is independent of the phase.

$$\left(\frac{\partial^3 b}{\partial x_i \partial x_j \partial x_k}\right)_{T,P,n_{k\ne i,j,k}} = 0$$

Returns

d3b_dzizjzks [list[list[list[float]]]] Third mole fraction derivatives of *b* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

`property d3delta_dzizjzks`

Helper method for calculating the third composition derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \delta}{\partial x_i \partial x_j \partial x_k}\right)_{T,P,x_{m\ne i,j,k}} = 0$$

Returns

d3delta_dzizjzks [list[list[list[float]]]] Third composition derivative of *epsilon* of each component, [m^6/mol^5]

Notes

This derivative is checked numerically.

property d3epsilon_dzizjzks

Helper method for calculating the third composition derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \epsilon}{\partial x_i \partial x_j \partial x_k}\right)_{T, P, x_{m \ne i,j,k}} = 0$$

Returns

d2epsilon_dzizjzks [list[list[list[float]]]] Composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

dG_dep_dns(Z)

Calculates the molar departure Gibbs energy mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial G_{dep}}{\partial n_i}\right)_{T, P, n_{i\ne j}} = f\left(\left(\frac{\partial G_{dep}}{\partial x_i}\right)_{T, P, x_{i\ne j}} \right)$$

Apart from the ideal term, this is the formulation for chemical potential.

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dG_dep_dns [float] Departure Gibbs energy mole number derivatives, [J/mol^2]

dG_dep_dzs(Z)

Calculates the molar departure Gibbs energy composition derivative (where the mole fractions do not sum to 1). Verified numerically. Useful in solving for gibbs minimization calculations or for solving for the true critical point. Also forms the basis for the molar departure Gibbs energy mole number derivative and molar partial departure Gibbs energy.

$$\left(\frac{\partial G_{dep}}{\partial x_i}\right)_{T, P, x_{i\ne j}} = P \frac{d}{dx} V{\left(x \right)} - \frac{R T \left(\frac{d}{d x} V{\left(x \right)} - \frac{d}{d x} b{\left(x \right)}\right)}{V{\left(x \right)} - b{\left(x \right)}} - \frac{2 \left(- \delta{\left(x \right)} \frac{d}{d x} \delta{\left(x \right)} + 2 \frac{d}{d x} \epsilon{\left(x \right)}\right) \operatorname{a\alpha}{\left(x \right)} \operatorname{atanh}{\left(\frac{2 V{\left(x \right)}}{\sqrt{\delta^{2}{\left(x \right)} - 4 \epsilon{\left(x \right)}}}\right.}}{\left(\delta^{2}{\left(x \right)} - 4 \epsilon{\left(x \right)}\right)^{\frac{3}{2}}}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dG_dep_dzs [float] Departure Gibbs free energy composition derivatives, [J/mol]

Notes

The derivation for the derivative is performed as follows using SymPy. The function source code is an optimized variant created with the *cse* SymPy function, and hand optimized further.

```
>>> from sympy import *
>>> P, T, R, x = symbols('P, T, R, x')
>>> a_alpha, a, delta, epsilon, V, b, da_alpha_dT = symbols('a\ \\alpha, a,␣
↪delta, epsilon, V, b, da_alpha_dT', cls=Function)
>>> S_dep = R*log(P*V(x)/(R*T)) + R*log(V(x)-b(x))+2*da_alpha_
↪dT(x)*atanh((2*V(x)+delta(x))/sqrt(delta(x)**2-4*epsilon(x)))/
↪sqrt(delta(x)**2-4*epsilon(x))-R*log(V(x))
>>> H_dep = P*V(x) - R*T + 2*atanh((2*V(x)+delta(x))/sqrt(delta(x)**2-
↪4*epsilon(x)))*(da_alpha_dT(x)*T-a_alpha(x))/sqrt(delta(x)**2-4*epsilon(x))
>>> G_dep = simplify(H_dep - T*S_dep)
>>> diff(G_dep, x)
P*Derivative(V(x), x) - R*T*(Derivative(V(x), x) - Derivative(b(x), x))/(V(x) -␣
↪b(x)) - 2*(-delta(x)*Derivative(delta(x), x) + 2*Derivative(epsilon(x), x))*a␣
↪\alpha(x)*atanh(2*V(x)/sqrt(delta(x)**2 - 4*epsilon(x)) + delta(x)/
↪sqrt(delta(x)**2 - 4*epsilon(x)))/(delta(x)**2 - 4*epsilon(x))**(3/2) -␣
↪2*atanh(2*V(x)/sqrt(delta(x)**2 - 4*epsilon(x)) + delta(x)/sqrt(delta(x)**2 -␣
↪4*epsilon(x)))*Derivative(a \alpha(x), x)/sqrt(delta(x)**2 - 4*epsilon(x)) -␣
↪2*(2*(-delta(x)*Derivative(delta(x), x) + 2*Derivative(epsilon(x), x))*V(x)/
↪(delta(x)**2 - 4*epsilon(x))**(3/2) + (-delta(x)*Derivative(delta(x), x) +␣
↪2*Derivative(epsilon(x), x))*delta(x)/(delta(x)**2 - 4*epsilon(x))**(3/2) +␣
↪2*Derivative(V(x), x)/sqrt(delta(x)**2 - 4*epsilon(x)) + Derivative(delta(x),␣
↪x)/sqrt(delta(x)**2 - 4*epsilon(x)))*a \alpha(x)/((1 - (2*V(x)/
↪sqrt(delta(x)**2 - 4*epsilon(x)) + delta(x)/sqrt(delta(x)**2 -␣
↪4*epsilon(x)))**2)*sqrt(delta(x)**2 - 4*epsilon(x)))
```

dH_dep_dns(Z)

Calculates the molar departure enthalpy mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial H_{dep}}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial H_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dH_dep_dns [float] Departure enthalpy mole number derivatives, [J/mol^2]

dH_dep_dzs(Z)

Calculates the molar departure enthalpy composition derivative (where the mole fractions do not sum to 1). Verified numerically. Useful in solving for enthalpy specifications in newton-type methods, and forms the basis for the molar departure enthalpy mole number derivative and molar partial departure enthalpy.

$$\left(\frac{\partial H_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}} = P\frac{d}{dx}V(x) + \frac{2\left(T\frac{\partial}{\partial T}\operatorname{a\alpha}(T,x) - \operatorname{a\alpha}(x)\right)\left(-\delta(x)\frac{d}{dx}\delta(x) + 2\frac{d}{dx}\epsilon(x)\right)\operatorname{atanh}\left(\frac{2V(x)+\delta(x)}{\sqrt{\delta^2(x)-4\epsilon(x)}}\right)}{\left(\delta^2(x)-4\epsilon(x)\right)^{\frac{3}{2}}} + \frac{2\,(\cdots}{\cdots}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dH_dep_dzs [float] Departure enthalpy composition derivatives, [J/mol]

Notes

The derivation for the derivative is performed as follows using SymPy. The function source code is an optimized variant created with the *cse* SymPy function, and hand optimized further.

```
>>> from sympy import *
>>> P, T, V, R, b, a, delta, epsilon, x = symbols('P, T, V, R, b, a, delta,␣
↪epsilon, x')
>>> V, delta, epsilon, a_alpha, b = symbols('V, delta, epsilon, a_alpha, b',␣
↪cls=Function)
>>> H_dep = (P*V(x) - R*T + 2/sqrt(delta(x)**2 - 4*epsilon(x))*(T*Derivative(a_
↪alpha(T, x), T)
... - a_alpha(x))*atanh((2*V(x)+delta(x))/sqrt(delta(x)**2-4*epsilon(x))))
>>> diff(H_dep, x)
P*Derivative(V(x), x) + 2*(T*Derivative(a \alpha(T, x), T) - a \alpha(x))*(-
↪delta(x)*Derivative(delta(x), x) + 2*Derivative(epsilon(x), x))*atanh((2*V(x)␣
↪+ delta(x))/sqrt(delta(x)**2 - 4*epsilon(x)))/(delta(x)**2 -␣
↪4*epsilon(x))**(3/2) + 2*(T*Derivative(a \alpha(T, x), T) - a \alpha(x))*((-
↪delta(x)*Derivative(delta(x), x) + 2*Derivative(epsilon(x), x))*(2*V(x) +␣
↪delta(x))/(delta(x)**2 - 4*epsilon(x))**(3/2) + (2*Derivative(V(x), x) +␣
↪Derivative(delta(x), x))/sqrt(delta(x)**2 - 4*epsilon(x)))/((-(2*V(x) +␣
↪delta(x))**2/(delta(x)**2 - 4*epsilon(x)) + 1)*sqrt(delta(x)**2 -␣
↪4*epsilon(x))) + 2*(T*Derivative(a \alpha(T, x), T, x) - Derivative(a \
↪alpha(x), x))*atanh((2*V(x) + delta(x))/sqrt(delta(x)**2 - 4*epsilon(x)))/
↪sqrt(delta(x)**2 - 4*epsilon(x))
```

dS_dep_dns(Z)

Calculates the molar departure entropy mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial S_{dep}}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial S_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dS_dep_dns [float] Departure entropy mole number derivatives, [J/mol^2/K]

dS_dep_dzs(Z)

Calculates the molar departure entropy composition derivative (where the mole fractions do not sum to 1). Verified numerically. Useful in solving for entropy specifications in newton-type methods, and forms the basis for the molar departure entropy mole number derivative and molar partial departure entropy.

$$\left(\frac{\partial S_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}} = \frac{1}{T}\left(\left(\frac{\partial H_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}} - \left(\frac{\partial G_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dS_dep_dzs [float] Departure entropy composition derivatives, [J/mol/K]

dV_dns(*Z*)

Calculates the molar volume mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial V}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial V}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dV_dns [float] Molar volume mole number derivatives, [m^3/mol^2]

dV_dzs(*Z*)

Calculates the molar volume composition derivative (where the mole fractions do not sum to 1). Verified numerically. Used in many other derivatives, and for the molar volume mole number derivative and partial molar volume calculation.

$$\left(\frac{\partial V}{\partial x_i}\right)_{T,P,x_{i\ne j}} = \frac{-RT\left(V^2(x)+V(x)\delta(x)+\epsilon(x)\right)^3\frac{d}{dx}b(x) + (V(x)-b(x))^2\left(V^2(x)+V(x)\delta(x)+\epsilon(x)\right)^2\frac{d}{dx}\mathrm{a}\alpha(x)}{-RT\left(V^2(x)+V(x)\delta(x)+\epsilon(x)\right)^3}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dV_dzs [float] Molar volume composition derivatives, [m^3/mol]

Notes

The derivation for the derivative is performed as follows using SymPy. The function source code is an optimized variant created with the *cse* SymPy function, and hand optimized further.

```
>>> from sympy import *
>>> P, T, R, x = symbols('P, T, R, x')
>>> V, delta, epsilon, a_alpha, b = symbols('V, delta, epsilon, a\ \\alpha, b', cls=Function)
>>> CUBIC = R*T/(V(x) - b(x)) - a_alpha(x)/(V(x)*V(x) + delta(x)*V(x) + epsilon(x)) - P
>>> solve(diff(CUBIC, x), Derivative(V(x), x))
[(-R*T*(V(x)**2 + V(x)*delta(x) + epsilon(x))**3*Derivative(b(x), x) + (V(x) - b(x))**2*(V(x)**2 + V(x)*delta(x) + epsilon(x))**2*Derivative(a \alpha(x), x) - (V(x) - b(x))**2*V(x)**3*a \alpha(x)*Derivative(delta(x), x) - (V(x) - b(x))**2*V(x)**2*a \alpha(x)*delta(x)*Derivative(delta(x), x) - (V(x) - b(x))**2*V(x)**2*a \alpha(x)*Derivative(epsilon(x), x) - (V(x) - b(x))**2*V(x)*a \alpha(x)*delta(x)*Derivative(epsilon(x), x) - (V(x) - b(x))**2*V(x)*a \alpha(x)*epsilon(x)*Derivative(delta(x), x) - (V(x) - b(x))**2*a \alpha(x)*epsilon(x)*Derivative(epsilon(x), x))/(-R*T*(V(x)**2 + V(x)*delta(x) + epsilon(x))**3 + 2*(V(x) - b(x))**2*V(x)**3*a \alpha(x) + 3*(V(x) - b(x))**2*V(x)**2*a \alpha(x)*delta(x) + (V(x) - b(x))**2*V(x)*a \alpha(x)*delta(x)**2 + 2*(V(x) - b(x))**2*V(x)*a \alpha(x)*epsilon(x) + (V(x) - b(x))**2*a \alpha(x)*delta(x)*epsilon(x))]
```

dZ_dns(*Z*)

Calculates the compressibility mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial Z}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial Z}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dZ_dns [float] Compressibility number derivatives, [1/mol]

dZ_dzs(*Z*)

Calculates the compressibility composition derivatives (where the mole fractions do not sum to 1). No specific formula is implemented for this property - it is calculated from the composition derivative of molar volume, which does have its formula implemented.

$$\left(\frac{\partial Z}{\partial x_i}\right)_{T,P,x_{i\ne j}} = \frac{P}{RT}\left(\frac{\partial V}{\partial x_i}\right)_{T,P,x_{i\ne j}}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dZ_dzs [float] Compressibility composition derivative, [-]

property da_alpha_dT_dns

Helper method for calculating the mole number derivatives of *da_alpha_dT*. Note this is independent of the phase.

$$\left(\frac{\partial^2 a\alpha}{\partial n_i \partial T}\right)_{P,n_{i\ne j}} = 2\left[\sum_j -z_j(k_{ij}-1)(a\alpha)_i(a\alpha)_j \frac{\partial (a\alpha)_i}{\partial T}\frac{\partial (a\alpha)_j}{\partial T}\left((a\alpha)_i(a\alpha)_j\right)^{-0.5} - \frac{\partial a\alpha}{\partial T}\right]$$

Returns

da_alpha_dT_dns [list[float]] Composition derivative of *da_alpha_dT* of each component, [kg*m^5/(mol^3*s^2*K)]

Notes

This derivative is checked numerically.

property da_alpha_dT_dzs

Helper method for calculating the composition derivatives of *da_alpha_dT*. Note this is independent of the phase.

$$\left(\frac{\partial^2 a\alpha}{\partial x_i \partial T}\right)_{P,x_{i\ne j}} = 2\sum_j -z_j(k_{ij}-1)(a\alpha)_i(a\alpha)_j \frac{\partial (a\alpha)_i}{\partial T}\frac{\partial (a\alpha)_j}{\partial T}\left((a\alpha)_i(a\alpha)_j\right)^{-0.5}$$

Returns

da_alpha_dT_dzs [list[float]] Composition derivative of *da_alpha_dT* of each component, [kg*m^5/(mol^2*s^2*K)]

Notes

This derivative is checked numerically.

property da_alpha_dT_ijs

Calculate and return the matrix for the temperature derivatives of the alpha terms.

$$\frac{\partial (a\alpha)_{ij}}{\partial T} = \frac{\sqrt{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} \left(1 - k_{ij}\right) \left(\frac{\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)}}{2} + \frac{\operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}}{2}\right)}{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}}$$

Returns

da_alpha_dT_ijs [list[list[float]]] First temperature derivative of *a_alpha* terms for each component with every other component, [J^2/mol^2/Pa/K]

Notes

In an earlier implementation this matrix was stored each EOS solve; however, allocating that much memory becomes quite expensive for large number of component cases and this is now calculated on-demand only.

property da_alpha_dns

Helper method for calculating the mole number derivatives of *a_alpha*. Note this is independent of the phase.

$$\left(\frac{\partial a \alpha}{\partial n_i}\right)_{T, P, n_{i\ne j}} = 2(-a\alpha + \sum_j z_j(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j})$$

Returns

da_alpha_dns [list[float]] Mole number derivative of *alpha* of each component, [kg*m^5/(mol^3*s^2)]

Notes

This derivative is checked numerically.

property da_alpha_dzs

Helper method for calculating the composition derivatives of *a_alpha*. Note this is independent of the phase.

$$\left(\frac{\partial a \alpha}{\partial x_i}\right)_{T, P, x_{i\ne j}} = 2 \cdot \sum_j z_j(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

Returns

da_alpha_dzs [list[float]] Composition derivative of *alpha* of each component, [kg*m^5/(mol^2*s^2)]

Notes

This derivative is checked numerically.

property db_dns

Helper method for calculating the mole number derivatives of b. Note this is independent of the phase.

$$\left(\frac{\partial b}{\partial n_i}\right)_{T,P,n_{i\ne j}} = b_i - b$$

Returns

db_dns [list[float]] Composition derivative of b of each component, [m^3/mol^2]

Notes

This derivative is checked numerically.

property db_dzs

Helper method for calculating the composition derivatives of b. Note this is independent of the phase.

$$\left(\frac{\partial b}{\partial x_i}\right)_{T,P,x_{i\ne j}} = b_i$$

Returns

db_dzs [list[float]] Composition derivative of b of each component, [m^3/mol]

Notes

This derivative is checked numerically.

dfugacities_dns(*phase*)

Generic formula for calculating the mole number derivaitves of fugacities for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here.

$$\left(\frac{\partial f_i}{\partial n_i}\right)_{P,n_{j\ne i}}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dfugacities_dns [list[list[float]]] Mole number derivatives of fugacities for each species, [-]

dlnfugacities_dns(*phase*)

Generic formula for calculating the mole number derivaitves of log fugacities for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here.

$$\left(\frac{\partial \ln f_i}{\partial n_i}\right)_{P,n_{j\ne i}}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnfugacities_dns [list[list[float]]] Mole number derivatives of log fugacities for each species, [-]

dlnphi_dns(*Z*)

Calculates the mixture log *fugacity coefficient* mole number derivatives (where the mole fractions sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative of Gibbs free energy.

$$\left(\frac{\partial \ln \phi}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial G_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

This property can be converted into a partial molar property to obtain the individual fugacity coefficients.

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dlnphi_dns [float] Mixture log fugacity coefficient mole number derivatives, [1/mol]

dlnphi_dzs(*Z*)

Calculates the mixture log *fugacity coefficient* mole fraction derivatives (where the mole fractions do not sum to 1). No specific formula is implemented for this property - it is calculated from the mole fraction derivative of Gibbs free energy.

$$\left(\frac{\partial \ln \phi}{\partial x_i}\right)_{T,P,x_{i\ne j}} = \frac{1}{RT}\left(\left(\frac{\partial G_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dlnphi_dzs [float] Mixture log fugacity coefficient mole fraction derivatives, [-]

dlnphis_dP(*phase*)

Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here.

Normally this routine is slower than EOS-specific ones, as it does not make assumptions that certain parameters are zero or equal to other parameters.

$$\left(\frac{\partial \ln \phi_i}{\partial P}\right)_{T,nj\ne i} = \frac{G_{dep}}{\partial P}_{T,n} + \left(\frac{\partial^2 \ln \phi}{\partial P \partial n_i}\right)_{T,P,n_{j\ne i}}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dP [float] Pressure derivatives of log fugacity coefficient for each species, [1/Pa]

Notes

This expression for the partial derivative of the mixture *lnphi* with respect to pressure and mole number can be derived as follows; to convert to the partial molar *lnphi* pressure and temperature derivative, add ::math::*frac{G_{dep}/(RT)}{partial P}_{T, n}*.

```
>>> from sympy import *
>>> P, T, R, n = symbols('P, T, R, n')
>>> a_alpha, a, delta, epsilon, V, b, da_alpha_dT, d2a_alpha_dT2 = symbols('a_
→alpha, a, delta, epsilon, V, b, da_alpha_dT, d2a_alpha_dT2', cls=Function)
>>> S_dep = R*log(P*V(n, P)/(R*T)) + R*log(V(n, P)-b(n))+2*da_alpha_dT(n,␣
→T)*atanh((2*V(n, P)+delta(n))/sqrt(delta(n)**2-4*epsilon(n)))/
→sqrt(delta(n)**2-4*epsilon(n))-R*log(V(n, P))
>>> H_dep = P*V(n, P) - R*T + 2*atanh((2*V(n, P)+delta(n))/sqrt(delta(n)**2-
→4*epsilon(n)))*(da_alpha_dT(n, T)*T-a_alpha(n, T))/sqrt(delta(n)**2-
→4*epsilon(n))
>>> G_dep = H_dep - T*S_dep
>>> lnphi = simplify(G_dep/(R*T))
>>> diff(diff(lnphi, P), n)
P*Derivative(V(n, P), P, n)/(R*T) + Derivative(V(n, P), P, n)/V(n, P) -␣
→Derivative(V(n, P), P)*Derivative(V(n, P), n)/V(n, P)**2 - Derivative(V(n, P),
→ P, n)/(V(n, P) - b(n)) - (-Derivative(V(n, P), n) + Derivative(b(n),␣
→n))*Derivative(V(n, P), P)/(V(n, P) - b(n))**2 + Derivative(V(n, P), n)/(R*T)␣
→- 4*(-2*delta(n)*Derivative(delta(n), n) + 4*Derivative(epsilon(n), n))*a_
→alpha(n, T)*Derivative(V(n, P), P)/(R*T*(1 - (2*V(n, P)/sqrt(delta(n)**2 -␣
→4*epsilon(n)) + delta(n)/sqrt(delta(n)**2 - 4*epsilon(n)))**2)*(delta(n)**2 -␣
→4*epsilon(n))**2) - 4*a_alpha(n, T)*Derivative(V(n, P), P, n)/(R*T*(1 -␣
→(2*V(n, P)/sqrt(delta(n)**2 - 4*epsilon(n)) + delta(n)/sqrt(delta(n)**2 -␣
→4*epsilon(n)))**2)*(delta(n)**2 - 4*epsilon(n))) - 4*Derivative(V(n, P),␣
→P)*Derivative(a_alpha(n, T), n)/(R*T*(1 - (2*V(n, P)/sqrt(delta(n)**2 -␣
→4*epsilon(n)) + delta(n)/sqrt(delta(n)**2 - 4*epsilon(n)))**2)*(delta(n)**2 -␣
→4*epsilon(n))) - 4*(2*V(n, P)/sqrt(delta(n)**2 - 4*epsilon(n)) + delta(n)/
→sqrt(delta(n)**2 - 4*epsilon(n)))*(4*(-delta(n)*Derivative(delta(n), n) +␣
→2*Derivative(epsilon(n), n))*V(n, P)/(delta(n)**2 - 4*epsilon(n))**(3/2) +␣
→2*(-delta(n)*Derivative(delta(n), n) + 2*Derivative(epsilon(n), n))*delta(n)/
→(delta(n)**2 - 4*epsilon(n))**(3/2) + 4*Derivative(V(n, P), n)/
→sqrt(delta(n)**2 - 4*epsilon(n)) + 2*Derivative(delta(n), n)/sqrt(delta(n)**2␣
→- 4*epsilon(n)))*a_alpha(n, T)*Derivative(V(n, P), P)/(R*T*(1 - (2*V(n, P)/
→sqrt(delta(n)**2 - 4*epsilon(n)) + delta(n)/sqrt(delta(n)**2 -␣
→4*epsilon(n)))**2)**2*(delta(n)**2 - 4*epsilon(n))) + R*T*(P*Derivative(V(n,␣
→P), P)/(R*T) + V(n, P)/(R*T))*Derivative(V(n, P), n)/(P*V(n, P)**2) -␣
→R*T*(P*Derivative(V(n, P), P, n)/(R*T) + Derivative(V(n, P), n)/(R*T))/(P*V(n,
→ P))
```

dlnphis_dT(*phase*)

Generic formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here.

Normally this routine is slower than EOS-specific ones, as it does not make assumptions that certain parameters are zero or equal to other parameters.

$$\left(\frac{\partial \ln \phi_i}{\partial T}\right)_{P,nj\ne i} = \frac{\frac{G_{dep}}{RT}}{\partial T}_{P,n} + \left(\frac{\partial^2 \ln \phi}{\partial T \partial n_i}\right)_{P,n_{j\ne i}}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dT [float] Temperature derivatives of log fugacity coefficient for each species, [1/K]

Notes

This expression for the partial derivative of the mixture *lnphi* with respect to pressure and mole number can be derived as follows; to convert to the partial molar *lnphi* pressure and temperature derivative, add ::math::*frac{G_{dep}/(RT)}{partial T}_{P, n}*.

```
>>> from sympy import *
>>> P, T, R, n = symbols('P, T, R, n')
>>> a_alpha, a, delta, epsilon, V, b, da_alpha_dT, d2a_alpha_dT2 = symbols('a_
↪alpha, a, delta, epsilon, V, b, da_alpha_dT, d2a_alpha_dT2', cls=Function)
>>> S_dep = R*log(P*V(n, T)/(R*T)) + R*log(V(n, T)-b(n))+2*da_alpha_dT(n,␣
↪T)*atanh((2*V(n, T)+delta(n))/sqrt(delta(n)**2-4*epsilon(n)))/
↪sqrt(delta(n)**2-4*epsilon(n))-R*log(V(n, T))
>>> H_dep = P*V(n, T) - R*T + 2*atanh((2*V(n, T)+delta(n))/sqrt(delta(n)**2-
↪4*epsilon(n)))*(da_alpha_dT(n, T)*T-a_alpha(n, T))/sqrt(delta(n)**2-
↪4*epsilon(n))
>>> G_dep = H_dep - T*S_dep
>>> lnphi = simplify(G_dep/(R*T))
>>> diff(diff(lnphi, T), n)
```

dlnphis_dns(*Z*)

Generic formula for calculating the mole number derivaitves of log fugacity coefficients for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here.

$$\left(\frac{\partial \ln \phi_i}{\partial n_i}\right)_{P,n_{j\ne i}}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dlnphis_dns [list[list[float]]] Mole number derivatives of log fugacity coefficient for each species, [-]

dlnphis_dzs(*Z*)

Generic formula for calculating the mole fraction derivaitves of log fugacity coefficients for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here.

$$\left(\frac{\partial \ln \phi_i}{\partial z_i}\right)_{P,z_{j\ne i}}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dlnphis_dzs [list[list[float]]] Mole fraction derivatives of log fugacity coefficient for each species (such that the mole fractions do not sum to 1), [-]

dnG_dep_dns(Z)

Calculates the partial molar departure Gibbs energy. No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial nG_{dep}}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial G_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dnG_dep_dns [float] Partial molar departure Gibbs energy of the phase, [J/mol]

dnH_dep_dns(Z)

Calculates the partial molar departure enthalpy. No specific formula is implemented for this property - it is calculated from the mole fraction derivative.

$$\left(\frac{\partial nH_{dep}}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial H_{dep}}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dnH_dep_dns [float] Partial molar departure enthalpies of the phase, [J/mol]

dnV_dns(Z)

Calculates the partial molar volume of the specified phase No specific formula is implemented for this property - it is calculated from the molar volume mole fraction derivative.

$$\left(\frac{\partial nV}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial V}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dnV_dns [float] Partial molar volume of the mixture of the specified phase, [m^3/mol]

dnZ_dns(Z)

Calculates the partial compressibility of the specified phase No specific formula is implemented for this property - it is calculated from the compressibility mole fraction derivative.

$$\left(\frac{\partial nZ}{\partial n_i}\right)_{T,P,n_{i\ne j}} = f\left(\left(\frac{\partial Z}{\partial x_i}\right)_{T,P,x_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dnZ_dns [float] Partial compressibility of the mixture of the specified phase, [-]

property dna_alpha_dT_dns

Helper method for calculating the mole number derivatives of *da_alpha_dT*. Note this is independent of the phase.

$$\left(\frac{\partial^2 n a \alpha}{\partial n_i \partial T}\right)_{P,n_{i\ne j}} = 2\left[\sum_j -z_j(k_{ij}-1)(a\alpha)_i(a\alpha)_j \frac{\partial (a\alpha)_i}{\partial T}\frac{\partial (a\alpha)_j}{\partial T}\left((a\alpha)_i(a\alpha)_j\right)^{-0.5} - 0.5\frac{\partial a\alpha}{\partial T}\right]$$

Returns

dna_alpha_dT_dns [list[float]] Composition derivative of *da_alpha_dT* of each component, [kg*m^5/(mol^2*s^2*K)]

Notes

This derivative is checked numerically.

property dna_alpha_dns

Helper method for calculating the partial molar derivatives of *a_alpha*. Note this is independent of the phase.

$$\left(\frac{\partial a\alpha}{\partial n_i}\right)_{T,P,n_{i\ne j}} = 2(-0.5a\alpha + \sum_j z_j(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j})$$

Returns

dna_alpha_dns [list[float]] Partial molar derivative of *alpha* of each component, [kg*m^5/(mol^2*s^2)]

Notes

This derivative is checked numerically.

property dnb_dns

Helper method for calculating the partial molar derivative of *b*. Note this is independent of the phase.

$$\left(\frac{\partial n\cdot b}{\partial n_i}\right)_{T,P,n_{i\ne j}} = b_i$$

Returns

dnb_dns [list[float]] Partial molar derivative of *b* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

classmethod from_json(*json_repr*)

Method to create a mixture cubic equation of state from a JSON friendly serialization of another mixture cubic equation of state.

Parameters

json_repr [dict] Json representation, [-]

Returns

eos_mix [*GCEOSMIX*] Newly created object from the json serialization, [-]

Notes

It is important that the input string be in the same format as that created by `GCEOS.as_json`.

Examples

```
>>> import pickle
>>> eos = PRSV2MIX(Tcs=[507.6], Pcs=[3025000], omegas=[0.2975], zs=[1], T=299.,
↪P=1E6, kappa1s=[0.05104], kappa2s=[0.8634], kappa3s=[0.460])
>>> json_stuff = pickle.dumps(eos.as_json())
>>> new_eos = GCEOSMIX.from_json(pickle.loads(json_stuff))
>>> assert new_eos == eos
```

fugacities(*only_l=False*, *only_g=False*)

Helper method for calculating fugacity coefficients for any phases present, using either the overall mole fractions for both phases or using specified mole fractions for each phase.

Requires *fugacity_coefficients* to be implemented by each subclassing EOS.

In addition to setting *fugacities_l* and/or *fugacities_g*, this also sets the fugacity coefficients *phis_l* and/or *phis_g*.

$$\hat{\phi}_i^g = \frac{\hat{f}_i^g}{y_i P}$$

$$\hat{\phi}_i^l = \frac{\hat{f}_i^l}{x_i P}$$

Note that in a flash calculation, each phase requires their own EOS object.

Parameters

only_l [bool] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set.

only_g [bool] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set.

Notes

It is helpful to check that *fugacity_coefficients* has been implemented correctly using the following expression, from [1].

$$\ln \hat{\phi}_i = \left[\frac{\partial(n\ln\phi)}{\partial n_i}\right]_{T,P,n_j,V_t}$$

For reference, several expressions for fugacity of a component are as follows, shown in [1] and [2].

$$\ln \hat{\phi}_i = \int_{0}^P\left(\frac{\hat{V}_i}{RT} - \frac{1}{P}\right)dP$$

$$\ln \hat{\phi}_i = \int_V^\infty \left[\frac{1}{RT}\frac{\partial P}{\partial n_i} - \frac{1}{V}\right] dV - \ln Z$$

References

[1], [2]

fugacity_coefficients(Z)

Generic formula for calculating log fugacity coefficients for each species in a mixture. Verified numerically. Applicable to all cubic equations of state which can be cast in the form used here. Normally this routine is slower than EOS-specific ones, as it does not make assumptions that certain parameters are zero or equal to other parameters.

$$\left(\frac{\partial n\ln\phi}{\partial n_i}\right)_{n_{k\ne i}} = \ln\phi_i = \ln\phi + n\left(\frac{\partial\ln\phi}{\partial n_i}\right)_{n_{k\ne i}}$$

$$\left(\frac{\partial\ln\phi}{\partial n_i}\right)_{T,P,n_{i\ne j}} = \frac{1}{RT}\left(\left(\frac{\partial G_{dep}}{\partial n_i}\right)_{T,P,n_{i\ne j}}\right)$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

log_phis [float] Log fugacity coefficient for each species, [-]

kwargs_linear = ()

Tuple of 1D arguments used by the specific EOS in addition to the conventional ones.

kwargs_square = ('kijs',)

Tuple of 2D arguments used by the specific EOS.

mechanical_critical_point()

Method to calculate the mechanical critical point of a mixture of defined composition.

The mechanical critical point is where:

$$\frac{\partial P}{\partial\rho}|_T = \frac{\partial^2 P}{\partial\rho^2}|_T = 0$$

Returns

T [float] Mechanical critical temperature, [K]

P [float] Mechanical critical temperature, [Pa]

Notes

One useful application of the mechanical critical temperature is that the phase identification approach of Venkatarathnam is valid only up to it.

Note that the equation of state, when solved at these conditions, will have fairly large (1e-3 - 1e-6) results for the derivatives; but they are the minimum. This is just from floating point precision.

It can also be checked looking at the calculated molar volumes - all three (available with `sorted_volumes`) will be very close (1e-5 difference in practice), again differing because of floating point error.

The algorithm here is a custom implementation, using Newton-Raphson's method with the initial guesses described in [1] (mole-weighted critical pressure average, critical temperature average using a quadratic mixing rule). Normally ~4 iterations are needed to solve the system. It is relatively fast, as only one evaluation of *a_alpha* and *da_alpha_dT* are needed per call to function and its jacobian.

References

[1], [2]

mix_kwargs_to_pure = {}

multicomponent = True

All inherited classes of GCEOSMIX are multicomponent.

nonstate_constants = ('N', 'cmps', 'Tcs', 'Pcs', 'omegas', 'kijs', 'kwargs', 'ais', 'bs')

property pseudo_Pc

Apply a linear mole-fraction mixing rule to compute the average critical pressure, [Pa].

Examples

```
>>> base = RKMIX(T=150.0, P=4e6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.pseudo_Pc
3878000.0
```

property pseudo_Tc

Apply a linear mole-fraction mixing rule to compute the average critical temperature, [K].

Examples

```
>>> base = RKMIX(T=150.0, P=4e6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.pseudo_Tc
151.9
```

property pseudo_a

Apply a linear mole-fraction mixing rule to compute the average *a* coefficient, [-].

Examples

```
>>> base = RKMIX(T=150.0, P=4e6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.pseudo_a
0.17634464184
```

property pseudo_omega

Apply a linear mole-fraction mixing rule to compute the average *omega*, [-].

Examples

```
>>> base = RKMIX(T=150.0, P=4e6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.pseudo_omega
0.0284
```

pures()

Helper method which returns a list of pure *EOSs* at the same T and P and base EOS as the mixture.

Returns

eos_pures [list[eos]] A list of pure-species EOSs at the same T and P as the system, [-]

Notes

This is useful for i.e. comparing mixture fugacities with the Lewis-Randall rule or when using an activity coefficient model which require pure component fugacities.

scalar = True

Whether the model is implemented using pure-Python lists of floats, or numpy arrays of float64.

set_dnzs_derivatives_and_departures(*n=True*, *x=True*, *only_l=False*, *only_g=False*)

Sets a number of mole number and/or composition partial derivatives of thermodynamic partial derivatives.

The list of properties set is as follows, with all properties suffixed with '_l' or '_g'

if *n* is True: d2P_dTdns, d2P_dVdns, d2V_dTdns, d2V_dPdns, d2T_dVdns, d2T_dPdns, d3P_dT2dns, d3P_dV2dns, d3V_dT2dns, d3V_dP2dns, d3T_dV2dns, d3T_dP2dns, d3V_dPdTdns, d3P_dTdVdns, d3T_dPdVdns, dV_dep_dns, dG_dep_dns, dH_dep_dns, dU_dep_dns, dS_dep_dns, dA_dep_dns

if *x* is True: d2P_dTdzs, d2P_dVdzs, d2V_dTdzs, d2V_dPdzs, d2T_dVdzs, d2T_dPdzs, d3P_dT2dzs, d3P_dV2dzs, d3V_dT2dzs, d3V_dP2dzs, d3T_dV2dzs, d3T_dP2dzs, d3V_dPdTdzs, d3P_dTdVdzs, d3T_dPdVdzs, dV_dep_dzs, dG_dep_dzs, dH_dep_dzs, dU_dep_dzs, dS_dep_dzs, dA_dep_dzs

Parameters

n [bool, optional] Whether or not to set the mole number derivatives (sums up to one), [-]

x [bool, optional] Whether or not to set the composition derivatives (does not sum up to one), [-]

only_l [bool, optional] Whether or not to set only the liquid-like phase properties (if there are two phases), [-]

only_g [bool, optional] Whether or not to set only the gas-like phase properties (if there are two phases), [-]

solve_T(*P*, *V*, *quick=True*, *solution=None*)

Generic method to calculate T from a specified P and V. Provides SciPy's *newton* solver, and iterates to solve the general equation for P, recalculating *a_alpha* as a function of temperature using *a_alpha_and_derivatives* each iteration.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

quick [bool, optional] Unimplemented, although it may be possible to derive explicit expressions as done for many pure-component EOS

solution [str or None, optional] 'l' or 'g' to specify a liquid of vapor solution (if one exists); if None, will select a solution more likely to be real (closer to STP, attempting to avoid temperatures like 60000 K or 0.0001 K).

Returns

T [float] Temperature, [K]

stabiliy_iteration_Michelsen(*T*, *P*, *zs*, *Ks_initial=None*, *maxiter=20*, *xtol=1e-12*, *liq=True*)

subset(*idxs*, ***state_specs*)

Method to construct a new GCEOSMIX that removes all components not specified in the *idxs* argument.

Parameters

idxs [list[int] or Slice] Indexes of components that should be included, [-]

Returns

subset_eos [GCEOSMIX] Multicomponent GCEOSMIX at the same specified specs but with a composition normalized to 1 and with fewer components, [-]

state_specs [float] Keyword arguments which can be any of *T*, *P*, *V*, *zs*; *zs* is optional, as are (*T*, *P*, *V*), but if any of (*T*, *P*, *V*) are specified, a second one is required as well, [various]

Notes

Subclassing equations of state require their kwargs_linear and kwargs_square attributes to be correct for this to work. *Tcs*, *Pcs*, and *omegas* are always assumed to be used.

Examples

```
>>> kijs = [[0.0, 0.00076, 0.00171], [0.00076, 0.0, 0.00061], [0.00171, 0.00061,
↪ 0.0]]
>>> PR3 = PRMIX(Tcs=[469.7, 507.4, 540.3], zs=[0.8168, 0.1501, 0.0331],␣
↪omegas=[0.249, 0.305, 0.349], Pcs=[3.369E6, 3.012E6, 2.736E6], T=322.29,␣
↪P=101325.0, kijs=kijs)
>>> PR3.subset([1,2])
PRMIX(Tcs=[507.4, 540.3], Pcs=[3012000.0, 2736000.0], omegas=[0.305, 0.349],␣
↪kijs=[[0.0, 0.00061], [0.00061, 0.0]], zs=[0.8193231441048036, 0.
↪1806768558951965], T=322.29, P=101325.0)
>>> PR3.subset([1,2], T=500.0, P=1e5, zs=[.2, .8])
PRMIX(Tcs=[507.4, 540.3], Pcs=[3012000.0, 2736000.0], omegas=[0.305, 0.349],␣
↪kijs=[[0.0, 0.00061], [0.00061, 0.0]], zs=[0.2, 0.8], T=500.0, P=100000.0)
>>> PR3.subset([1,2], zs=[.2, .8])
PRMIX(Tcs=[507.4, 540.3], Pcs=[3012000.0, 2736000.0], omegas=[0.305, 0.349],␣
↪kijs=[[0.0, 0.00061], [0.00061, 0.0]], zs=[0.2, 0.8], T=322.29, P=101325.0)
```

to(*zs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*)

Method to construct a new GCEOSMIX object at two of *T*, *P* or *V* with the specified composition. In the event the specs match those of the current object, it will be returned unchanged.

Parameters

zs [list[float], optional] Mole fractions of EOS, [-]

T [float or None, optional] Temperature, [K]

P [float or None, optional] Pressure, [Pa]

V [float or None, optional] Molar volume, [m^3/mol]

fugacities [bool] Whether or not to calculate fugacities, [-]

Returns

obj [*GCEOSMIX*] Pure component *GCEOSMIX* at the two specified specs, [-]

Notes

Constructs the object with parameters *Tcs*, *Pcs*, *omegas*, and *kwargs*.

Examples

```
>>> base = PRMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.to(T=300.0, P=1e9).state_specs
{'T': 300.0, 'P': 1000000000.0}
>>> base.to(T=300.0, V=1.0).state_specs
{'T': 300.0, 'V': 1.0}
>>> base.to(P=1e5, V=1.0).state_specs
{'P': 100000.0, 'V': 1.0}
```

to_PV(*P*, *V*)

Method to construct a new *GCEOSMIX* object at the spcified *P* and *V* with the current composition. In the event the *P* and *V* match the current object's *P* and *V*, it will be returned unchanged.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

Returns

obj [*GCEOSMIX*] Pure component *GCEOSMIX* at specified *P* and *V*, [-]

Notes

Constructs the object with parameters *Tcs*, *Pcs*, *omegas*, and *kwargs*.

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> new = base.to_PV(P=1000000.0, V=1.0)
>>> base.state_specs, new.state_specs
({'T': 500.0, 'P': 1000000.0}, {'P': 1000000.0, 'V': 1.0})
```

to_PV_zs(*P*, *V*, *zs*, *fugacities=True*, *only_l=False*, *only_g=False*)

Method to construct a new *GCEOSMIX* instance at *P*, *V*, and *zs* with the same parameters as the existing object. Optionally, only one set of phase properties can be solved for, increasing speed. The fugacities calculation can be be skipped by by setting *fugacities* to False.

Parameters

P [float] Pressure, [Pa]

V [float] Molar volume, [m^3/mol]

zs [list[float]] Mole fractions of each component, [-]

fugacities [bool] Whether or not to calculate and set the fugacities of each component, [-]

only_l [bool] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set.

only_g [bool] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set.

Returns

eos [*GCEOSMIX*] Multicomponent *GCEOSMIX* at the specified conditions [-]

Notes

A check for whether or not *P*, *V*, and *zs* are the same as the existing instance is performed; if it is, the existing object is returned.

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.to_PV_zs(V=0.004162, P=1e5, zs=[.1, 0.9])
RKMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011],␣
↪kijs=[[0.0, 0.0], [0.0, 0.0]], zs=[0.1, 0.9], P=100000.0, V=0.004162)
```

to_TP(*T*, *P*)

Method to construct a new *GCEOSMIX* object at the spcified *T* and *P* with the current composition. In the event the *T* and *P* match the current object's *T* and *P*, it will be returned unchanged.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

Returns

obj [*GCEOSMIX*] Pure component *GCEOSMIX* at specified *T* and *P*, [-]

Notes

Constructs the object with parameters *Tcs*, *Pcs*, *omegas*, and *kwargs*.

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> new = base.to_TP(T=10.0, P=2000.0)
>>> base.state_specs, new.state_specs
({'T': 500.0, 'P': 1000000.0}, {'T': 10.0, 'P': 2000.0})
```

to_TPV_pure(*i*, *T=None*, *P=None*, *V=None*)

Helper method which returns a pure *EOSs* at the specs (two of *T*, *P* and *V*) and base EOS as the mixture for a particular index.

Parameters

i [int] Index of specified compound, [-]

T [float or None, optional] Specified temperature, [K]

P [float or None, optional] Specified pressure, [Pa]

V [float or None, optional] Specified volume, [m^3/mol]

Returns

eos_pure [eos] A pure-species EOSs at the two specified *T*, *P*, and *V* for component *i*, [-]

to_TP_zs(*T*, *P*, *zs*, *fugacities=True*, *only_l=False*, *only_g=False*)

Method to construct a new *GCEOSMIX* instance at *T*, *P*, and *zs* with the same parameters as the existing object. Optionally, only one set of phase properties can be solved for, increasing speed. The fugacities calculation can be be skipped by by setting *fugacities* to False.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

zs [list[float]] Mole fractions of each component, [-]

fugacities [bool] Whether or not to calculate and set the fugacities of each component, [-]

only_l [bool] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set.

only_g [bool] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set.

Returns

eos [*GCEOSMIX*] Multicomponent *GCEOSMIX* at the specified conditions [-]

Notes

A check for whether or not *T*, *P*, and *zs* are the same as the existing instance is performed; if it is, the existing object is returned.

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.to_TP_zs(T=300, P=1e5, zs=[.1, 0.9])
RKMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011],␣
↪kijs=[[0.0, 0.0], [0.0, 0.0]], zs=[0.1, 0.9], T=300, P=100000.0)
```

to_TP_zs_fast(*T*, *P*, *zs*, *only_l=False*, *only_g=False*, *full_alphas=True*)

Method to construct a new *GCEOSMIX* instance with the same parameters as the existing object. If both instances are at the same temperature, *a_alphas* and *da_alpha_dTs* and *d2a_alpha_dT2s* are shared between the instances. It is always assumed the new object has a differet composition. Optionally, only one set of phase properties can be solved for, increasing speed. Additionally, if *full_alphas* is set to False no temperature derivatives of *a_alpha* will be computed. Those derivatives are not needed in the context of a PT or PVF flash.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

zs [list[float]] Mole fractions of each component, [-]

only_l [bool] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set.

only_g [bool] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set.

Returns

eos [*GCEOSMIX*] Multicomponent *GCEOSMIX* at the specified conditions [-]

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.to_TP_zs_fast(T=300, P=1e5, zs=base.zs)
RKMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011],␣
↪kijs=[[0.0, 0.0], [0.0, 0.0]], zs=[0.6, 0.4], T=300, P=100000.0)
```

to_TV(*T*, *V*)

Method to construct a new *GCEOSMIX* object at the spcified *T* and *V* with the current composition. In the event the *T* and *V* match the current object's *T* and *V*, it will be returned unchanged.

Parameters

T [float] Temperature, [K]

V [float] Molar volume, [m^3/mol]

Returns

obj [*GCEOSMIX*] Pure component *GCEOSMIX* at specified *T* and *V*, [-]

Notes

Constructs the object with parameters *Tcs*, *Pcs*, *omegas*, and *kwargs*.

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> new = base.to_TV(T=1000000.0, V=1.0)
>>> base.state_specs, new.state_specs
({'T': 500.0, 'P': 1000000.0}, {'T': 1000000.0, 'V': 1.0})
```

to_mechanical_critical_point()

Method to construct a new *GCEOSMIX* object at the current object's properties and composition, but which is at the mechanical critical point.

Returns

obj [*GCEOSMIX*] Pure component *GCEOSMIX* at mechanical critical point [-]

Examples

```
>>> base = RKMIX(T=500.0, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.6, 0.4])
>>> base.to_mechanical_critical_point()
RKMIX(Tcs=[126.1, 190.6], Pcs=[3394000.0, 4604000.0], omegas=[0.04, 0.011],␣
↪kijs=[[0.0, 0.0], [0.0, 0.0]], zs=[0.6, 0.4], T=151.861, P=3908737.9)
```

translated = False

Whether or not the model implements volume translation.

7.8.2 Peng-Robinson Family EOSs

Standard Peng Robinson

class thermo.eos_mix.**PRMIX**(*Tcs, Pcs, omegas, zs, kijs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_mix.GCEOSMIX*, *thermo.eos.PR*

Class for solving the Peng-Robinson [1] [2] cubic equation of state for a mixture of any number of compounds. Subclasses *PR*. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = [1 + \kappa_i(1 - \sqrt{T_{r,i}})]^2$$

$$\kappa_i = 0.37464 + 1.54226\omega_i - 0.26992\omega_i^2$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1], [2]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = PRMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], omegas=[0.
↪04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(3.6257362939e-05, 0.00070066592313)
>>> eos.fugacities_l, eos.fugacities_g
([793860.83821, 73468.552253], [436530.92470, 358114.63827])
```

Attributes

d2delta_dninjs Helper method for calculating the second mole number derivatives (hessian) of *delta*.

d2delta_dzizjs Helper method for calculating the second composition derivatives (hessian) of *delta*.

d2epsilon_dninjs Helper method for calculating the second mole number derivatives (hessian) of *epsilon*.

d2epsilon_dzizjs Helper method for calculating the second composition derivatives (hessian) of *epsilon*.

d3a_alpha_dT3 Method to calculate approximately the third temperature derivative of *a_alpha* for the PR EOS.

d3delta_dninjnks Helper method for calculating the third partial mole number derivatives of *delta*.

d3epsilon_dninjnks Helper method for calculating the third partial mole number derivatives of *epsilon*.

ddelta_dns Helper method for calculating the mole number derivatives of *delta*.

ddelta_dzs Helper method for calculating the composition derivatives of *delta*.

depsilon_dns Helper method for calculating the mole number derivatives of *epsilon*.

depsilon_dzs Helper method for calculating the composition derivatives of *epsilon*.

Methods

a_alpha_and_derivatives_vectorized(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the PR EOS.
a_alphas_vectorized(T)	Method to calculate the pure-component *a_alphas* for the PR EOS.
d3a_alpha_dT3_vectorized(T)	Method to calculate the third temperature derivative of pure-component *a_alphas* for the PR EOS.
dlnphis_dP(phase)	Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture for the Peng-Robinson EOS.
dlnphis_dT(phase)	Formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture for the Peng-Robinson equation of state.
dlnphis_dzs(Z)	Calculate and return the mole fraction derivaitves of log fugacity coefficients for each species in a mixture.
eos_pure	alias of thermo.eos.PR
fugacity_coefficients(Z)	Literature formula for calculating fugacity coefficients for each species in a mixture.

a_alpha_and_derivatives_vectorized(*T*)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the PR EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a\left(\kappa\left(-\frac{T^{0.5}}{Tc^{0.5}}+1\right)+1\right)^2$$

$$\frac{da\alpha}{dT} = -\frac{1.0a\kappa}{T^{0.5}Tc^{0.5}}\left(\kappa\left(-\frac{T^{0.5}}{Tc^{0.5}}+1\right)+1\right)$$

$$\frac{d^2a\alpha}{dT^2} = 0.5a\kappa\left(-\frac{1}{T^{1.5}Tc^{0.5}}\left(\kappa\left(\frac{T^{0.5}}{Tc^{0.5}}-1\right)-1\right)+\frac{\kappa}{T^{1.0}Tc^{1.0}}\right)$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

`a_alphas_vectorized(T)`

Method to calculate the pure-component *a_alphas* for the PR EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a\left(\kappa\left(-\frac{T^{0.5}}{Tc^{0.5}}+1\right)+1\right)^2$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

`property d2delta_dninjs`

Helper method for calculating the second mole number derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2\delta}{\partial n_i \partial n_j}\right)_{T,P,n_{k\ne i,j}} = 4b - 2b_i - 2b_j$$

Returns

d2delta_dninjs [list[list[float]]] Second mole number derivative of *delta* of each component, [m^3/mol^3]

Notes

This derivative is checked numerically.

`property d2delta_dzizjs`

Helper method for calculating the second composition derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2\delta}{\partial x_i \partial x_j}\right)_{T,P,x_{k\ne i,j}} = 0$$

Returns

d2delta_dzizjs [list[float]] Second Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property d2epsilon_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \epsilon}{\partial n_i n_j}\right)_{T,P,n_{k\neq i,j}} = -2b(2b - b_i - b_j) - 2(b - b_i)(b - b_j)$$

Returns

d2epsilon_dninjs [list[list[float]]] Second mole number derivative of *epsilon* of each component, [m^6/mol^4]

Notes

This derivative is checked numerically.

property d2epsilon_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \epsilon}{\partial x_i \partial x_j}\right)_{T,P,x_{k\neq i,j}} = 2b_i b_j$$

Returns

d2epsilon_dzizjs [list[list[float]]] Second composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

property d3a_alpha_dT3

Method to calculate approximately the third temperature derivative of *a_alpha* for the PR EOS. A rigorous calculation has not been implemented.

Parameters

T [float] Temperature, [K]

Returns

d3a_alpha_dT3 [float] Third temperature derivative $a\alpha$, [J^2/mol^2/Pa/K^3]

d3a_alpha_dT3_vectorized(T)

Method to calculate the third temperature derivative of pure-component *a_alphas* for the PR EOS. This vectorized implementation is added for extra speed.

Parameters

T [float] Temperature, [K]

Returns

d3a_alpha_dT3s [list[float]] Third temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^3]

property d3delta_dninjnks

Helper method for calculating the third partial mole number derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \delta}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\neq i,j,k}} = 4(-3b + b_i + b_j + b_k)$$

Returns

d3delta_dninjnks [list[list[list[float]]]] Third mole number derivative of *delta* of each component, [m^3/mol^4]

Notes

This derivative is checked numerically.

property d3epsilon_dninjnks

Helper method for calculating the third partial mole number derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \epsilon}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\neq i,j,k}} = 24b^2 - 12b(b_i + b_j + b_k) + 4(b_i b_j + b_i b_k + b_j b_k)$$

Returns

d3epsilon_dninjnks [list[list[list[float]]]] Third mole number derivative of *epsilon* of each component, [m^6/mol^5]

Notes

This derivative is checked numerically.

property ddelta_dns

Helper method for calculating the mole number derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial \delta}{\partial n_i}\right)_{T,P,n_{i\neq j}} = 2(b_i - b)$$

Returns

ddelta_dns [list[float]] Mole number derivative of *delta* of each component, [m^3/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dzs

Helper method for calculating the composition derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial \delta}{\partial x_i}\right)_{T,P,x_{i\neq j}} = 2b_i$$

Returns

ddelta_dzs [list[float]] Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property depsilon_dns

Helper method for calculating the mole number derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial \epsilon}{\partial n_i}\right)_{T,P,n_{i\neq j}} = 2b(b - b_i)$$

Returns

depsilon_dns [list[float]] Composition derivative of *epsilon* of each component, [m^6/mol^3]

Notes

This derivative is checked numerically.

property depsilon_dzs

Helper method for calculating the composition derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial \epsilon}{\partial x_i}\right)_{T,P,x_{i\neq j}} = -2b_i \cdot b$$

Returns

depsilon_dzs [list[float]] Composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

dlnphis_dP(*phase*)

Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture for the Peng-Robinson EOS. Verified numerically.

$$\left(\frac{\partial \ln \phi_i}{\partial P}\right)_{T,nj\neq i}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dP [float] Pressure derivatives of log fugacity coefficient for each species, [1/Pa]

Notes

This expression was derived using SymPy and optimized with the *cse* technique.

dlnphis_dT(*phase*)

Formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture for the Peng-Robinson equation of state. Verified numerically.

$$\left(\frac{\partial \ln \phi_i}{\partial T}\right)_{P,nj \ne i}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dT [float] Temperature derivatives of log fugacity coefficient for each species, [1/K]

Notes

This expression was derived using SymPy and optimized with the *cse* technique.

dlnphis_dzs(*Z*)

Calculate and return the mole fraction derivaitves of log fugacity coefficients for each species in a mixture. This formula is specific to the Peng-Robinson equation of state.

$$\left(\frac{\partial \ln \phi_i}{\partial z_i}\right)_{P,z_{j \ne i}}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

dlnphis_dzs [list[list[float]]] Mole fraction derivatives of log fugacity coefficient for each species (such that the mole fractions do not sum to 1), [-]

Notes

This formula is from [1] but is validated to match the generic implementation.

References

[1]

Examples

```
>>> kijs = [[0, 0.00076, 0.00171], [0.00076, 0, 0.00061], [0.00171, 0.00061, 0]]
>>> eos = PRMIX(Tcs=[469.7, 507.4, 540.3], zs=[0.8168, 0.1501, 0.0331],
↪omegas=[0.249, 0.305, 0.349], Pcs=[3.369E6, 3.012E6, 2.736E6], T=322.29,
↪P=101325, kijs=kijs)
>>> eos.dlnphis_dzs(eos.Z_l)
[[0.009938069276, 0.0151503498382, 0.018297235797], [-0.038517738793, -0.
↪05958926042, -0.068438990795], [-0.07057106923, -0.10363920720, -0.
↪14116283024]]
```

(continues on next page)

(continued from previous page)

eos_pure
alias of *thermo.eos.PR*

fugacity_coefficients(Z)
Literature formula for calculating fugacity coefficients for each species in a mixture. Verified numerically. Applicable to most derivatives of the Peng-Robinson equation of state as well. Called by *fugacities* on initialization, or by a solver routine which is performing a flash calculation.

$$\ln \hat \phi_i = \frac{B_i}{B}(Z-1)-\ln(Z-B) + \frac{A}{2\sqrt{2}B}\left[\frac{B_i}{B} - \frac{2}{a\alpha}\sum_i y_i(a\alpha)_{ij}\right]\ln\left[\frac{Z+(1+\sqrt{2})B}{Z-(\sqrt{2}-1)B}\right]$$

$$A = \frac{(a\alpha)P}{R^2 T^2}$$

$$B = \frac{bP}{RT}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

log_phis [float] Log fugacity coefficient for each species, [-]

Peng Robinson (1978)

class thermo.eos_mix.**PR78MIX**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*, *only_l=False*, *only_g=False*)
Bases: *thermo.eos_mix.PRMIX*

Class for solving the Peng-Robinson cubic equation of state for a mixture of any number of compounds according to the 1978 variant. Subclasses *PR*. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{v-b}-\frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a \alpha = \sum_i \sum_j z_i z_j {(a\alpha)}_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_{i}(a\alpha)_{j}}$$

$$b = \sum_i z_i b_i$$

$$a_i=0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$$

$$b_i=0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i=[1+\kappa_i(1-\sqrt{T_{r,i}})]^2$$

$$\kappa_i = 0.37464+1.54226\omega_i-0.26992\omega^2_i \text{ if } \omega_i \le 0.491$$

$$\kappa_i = 0.379642 + 1.48503 \omega_i - 0.164423\omega_i^2 + 0.016666 \omega_i^3 \text{ if } \omega_i > 0.491$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

This variant is recommended over the original.

References

[1], [2]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa, with modified acentric factors to show the difference between *PRMIX*

```
>>> eos = PR78MIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.6, 0.7], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(3.2396438915e-05, 0.00050433802024)
>>> eos.fugacities_l, eos.fugacities_g
([833048.45119, 6160.9088153], [460717.27767, 279598.90103])
```

Methods

`eos_pure`	alias of `thermo.eos.PR78`

eos_pure
alias of `thermo.eos.PR78`

Peng Robinson Stryjek-Vera

class thermo.eos_mix.**PRSVMIX**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *T=None*, *P=None*, *V=None*, *kappa1s=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: `thermo.eos_mix.PRMIX`, `thermo.eos.PRSV`

Class for solving the Peng-Robinson-Stryjek-Vera equations of state for a mixture as given in [1]. Subclasses `PRMIX` and `PRSV`. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Inherits the method of calculating fugacity coefficients from `PRMIX`. Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2 T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = [1+\kappa_i(1-\sqrt{T_{r,i}})]^2$$

$$\kappa_i = \kappa_{0,i} + \kappa_{1,i}(1+T_{r,i}^{0.5})(0.7-T_{r,i})$$

$$\kappa_{0,i} = 0.378893 + 1.4897153\omega_i - 0.17131848\omega_i^2 + 0.0196554\omega_i^3$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

kappa1s [list[float], optional] Fit parameter; available in [1] for over 90 compounds, SRKMIXTranslated[-]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

[1] recommends that *kappa1* be set to 0 for Tr > 0.7. This is not done by default; the class boolean *kappa1_Tr_limit* may be set to True and the problem re-solved with that specified if desired. *kappa1_Tr_limit* is not supported for P-V inputs.

For P-V initializations, a numerical solver is used to find T.

[2] and [3] are two more resources documenting the PRSV EOS. [4] lists *kappa* values for 69 additional compounds. See also *PRSV2MIX*. Note that tabulated *kappa* values should be used with the critical parameters used in their fits. Both [1] and [4] only considered vapor pressure in fitting the parameter.

References

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

Examples

P-T initialization, two-phase, nitrogen and methane

```
>>> eos = PRSVMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.phase, eos.V_l, eos.H_dep_l, eos.S_dep_l
('l/g', 3.6235536165e-05, -6349.0055583, -49.1240502472)
```

Methods

a_alpha_and_derivatives_vectorized(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the PRSV EOS.
a_alphas_vectorized(T)	Method to calculate the pure-component *a_alphas* for the PRSV EOS.
eos_pure	alias of *thermo.eos.PRSV*

a_alpha_and_derivatives_vectorized(T)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the PRSV EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a\left(\left(\kappa_0 + \kappa_1\left(\sqrt{\frac{T}{Tc}} + 1\right)\left(-\frac{T}{Tc} + \frac{7}{10}\right)\right)\left(-\sqrt{\frac{T}{Tc}} + 1\right) + 1\right)^2$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alphas_vectorized(*T*)

Method to calculate the pure-component *a_alphas* for the PRSV EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a\left(\left(\kappa_0 + \kappa_1\left(\sqrt{\frac{T}{Tc}} + 1\right)\left(-\frac{T}{Tc} + \frac{7}{10}\right)\right)\left(-\sqrt{\frac{T}{Tc}} + 1\right) + 1\right)^2$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

eos_pure

alias of *thermo.eos.PRSV*

Peng Robinson Stryjek-Vera 2

class thermo.eos_mix.**PRSV2MIX**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *T=None*, *P=None*, *V=None*, *kappa1s=None*, *kappa2s=None*, *kappa3s=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: *thermo.eos_mix.PRMIX*, *thermo.eos.PRSV2*

Class for solving the Peng-Robinson-Stryjek-Vera 2 equations of state for a Mixture as given in [1]. Subclasses *PRMIX* and *PRSV2 <thermo.eos.PRSV2>*. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Inherits the method of calculating fugacity coefficients from *PRMIX*. Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b) + b(v-b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2 T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = [1 + \kappa_i(1 - \sqrt{T_{r,i}})]^2$$

$$\kappa_i = \kappa_{0,i} + [\kappa_{1,i} + \kappa_{2,i}(\kappa_{3,i} - T_{r,i})(1 - T_{r,i}^{0.5})](1 + T_{r,i}^{0.5})(0.7 - T_{r,i})$$

$$\kappa_{0,i} = 0.378893 + 1.4897153\omega_i - 0.17131848\omega_i^2 + 0.0196554\omega_i^3$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

kappa1s [list[float], optional] Fit parameter; available in [1] for over 90 compounds, [-]

kappa2s [list[float], optional] Fit parameter; available in [1] for over 90 compounds, [-]

kappa3s [list[float], optional] Fit parameter; available in [1] for over 90 compounds, [-]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

Note that tabulated *kappa* values should be used with the critical parameters used in their fits. [1] considered only vapor pressure in fitting the parameter.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = PRSV2MIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(3.6235536165e-05, 0.00070024238654)
>>> eos.fugacities_l, eos.fugacities_g
([794057.58318, 72851.22327], [436553.65618, 357878.11066])
```

Methods

a_alpha_and_derivatives_vectorized(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the PRSV2 EOS.
a_alphas_vectorized(T)	Method to calculate the pure-component *a_alphas* for the PRSV2 EOS.
eos_pure	alias of *thermo.eos.PRSV2*

a_alpha_and_derivatives_vectorized(*T*)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the PRSV2 EOS. This vectorized implementation is added for extra speed.

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alphas_vectorized(*T*)

Method to calculate the pure-component *a_alphas* for the PRSV2 EOS. This vectorized implementation is added for extra speed.

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Examples

```
>>> eos = PRSV2MIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.a_alphas_vectorized(300)
[0.0860568595, 0.20174345803]
```

eos_pure
alias of *thermo.eos.PRSV2*

Peng Robinson Twu (1995)

class thermo.eos_mix.**TWUPRMIX**(*Tcs, Pcs, omegas, zs, kijs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_alpha_functions.TwuPR95_a_alpha*, *thermo.eos_mix.PRMIX*

Class for solving the Twu [1] variant of the Peng-Robinson cubic equation of state for a mixture. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v-b} - \frac{a\alpha(T)}{v(v+b)+b(v-b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2 T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha_i = \alpha_i^{(0)} + \omega_i(\alpha_i^{(1)} - \alpha_i^{(0)})$$

$$\alpha^{(\text{0 or 1})} = T_{r,i}^{N(M-1)} \exp[L(1 - T_{r,i}^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.125283, 0.911807, 1.948150;

L1, M1, N1 = 0.511614, 0.784054, 2.812520

For supercritical conditions:

L0, M0, N0 = 0.401219, 4.963070, -0.2;

L1, M1, N1 = 0.024955, 1.248089, -8.

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T. Claimed to be more accurate than the PR, PR78 and PRSV equations.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = TWUPRMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(3.624571041e-05, 0.0007004401318)
>>> eos.fugacities_l, eos.fugacities_g
([792155.022163, 73305.88829], [436468.967764, 358049.2495573])
```

Methods

eos_pure	alias of *thermo.eos.TWUPR*

eos_pure
alias of *thermo.eos.TWUPR*

Peng Robinson Translated

class thermo.eos_mix.**PRMIXTranslated**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *cs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: *thermo.eos_mix.PRMIX*

Class for solving the Peng-Robinson [1] [2] translated cubic equation of state for a mixture of any number of compounds. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v+c-b} - \frac{a\alpha(T)}{(v+c)(v+c+b)+b(v+c-b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2 T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = [1+\kappa_i(1-\sqrt{T_{r,i}})]^2$$

$$\kappa_i = 0.37464 + 1.54226\omega_i - 0.26992\omega_i^2$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters; always zero in the original implementation, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1], [2]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = PRMIXTranslated(T=115, P=1E6, cs=[-4.4e-6, -4.35e-6], Tcs=[126.1, 190.6],␣
↪Pcs=[33.94E5, 46.04E5], omegas=[0.04, 0.011], zs=[0.2, 0.8], kijs=[[0,0.03],[0.03,
↪0]])
>>> eos.V_l, eos.V_g
(3.9079056337e-05, 0.00060231393016)
>>> eos.fugacities_l, eos.fugacities_g
([442838.8615, 108854.48589], [184396.972, 565531.7709])
```

Attributes

d2delta_dninjs Helper method for calculating the second mole number derivatives (hessian) of *delta*.

d2delta_dzizjs Helper method for calculating the second composition derivatives (hessian) of *delta*.

d2epsilon_dninjs Helper method for calculating the second mole number derivatives (hessian) of *epsilon*.

d2epsilon_dzizjs Helper method for calculating the second composition derivatives (hessian) of *epsilon*.

d3delta_dninjnks Helper method for calculating the third partial mole number derivatives of *delta*.

d3delta_dzizjzks Helper method for calculating the third composition derivatives of *delta*.

d3epsilon_dninjnks Helper method for calculating the third partial mole number derivatives of *epsilon*.

d3epsilon_dzizjzks Helper method for calculating the third composition derivatives of *epsilon*.

ddelta_dns Helper method for calculating the mole number derivatives of *delta*.

ddelta_dzs Helper method for calculating the composition derivatives of *delta*.

depsilon_dns Helper method for calculating the mole number derivatives of *epsilon*.

depsilon_dzs Helper method for calculating the composition derivatives of *epsilon*.

Methods

eos_pure	alias of *thermo.eos.PRTranslated*

property d2delta_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial^2 \delta}{\partial n_i \partial n_j}\right)_{T,P,n_{k\ne i,j}} = 2\left(\delta - b_i^0 - b_j^0 - c_i - c_j\right)$$

Returns

d2delta_dninjs [list[list[float]]] Second mole number derivative of *delta* of each component, [m^3/mol^3]

Notes

This derivative is checked numerically.

property d2delta_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \delta}{\partial x_i \partial x_j}\right)_{T,P,x_{k\ne i,j}} = 0$$

Returns

d2delta_dzizjs [list[float]] Second Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property d2epsilon_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \epsilon}{\partial n_i n_j}\right)_{T,P,n_{k\ne i,j}} = -2b^0(2b^0 - b_i^0 - b_j^0) + c(4b^0 - 2b_i^0 - 2b_j^0 + 2c - c_i - c_j) - 2(b^0 - b_i^0)(b^0 - b_j^0) + (c - c_i)(2b^0 -$$

Returns

d2epsilon_dninjs [list[list[float]]] Second mole number derivative of *epsilon* of each component, [m^6/mol^4]

Notes

This derivative is checked numerically.

property d2epsilon_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \epsilon}{\partial x_i \partial x_j}\right)_{T, P, x_{k\ne i,j}} = -2b^0_i b^0_j + 2b^0_i c_j + 2b^0_j c_i + 2c_i c_j$$

Returns

d2epsilon_dzizjs [list[list[float]]] Second composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

property d3delta_dninjnks

Helper method for calculating the third partial mole number derivatives of *delta*. Note this is independent of the phase. b^0 refers to the original *b* parameter not involving any translation.

$$\left(\frac{\partial^3 \delta}{\partial n_i \partial n_j \partial n_k}\right)_{T, P, n_{m\ne i,j,k}} = 4\left(b^0_i + b^0_j + b^0_k + c_i + c_j + c_k\right) - 6\delta$$

Returns

d3delta_dninjnks [list[list[list[float]]]] Third mole number derivative of *delta* of each component, [m^3/mol^4]

Notes

This derivative is checked numerically.

property d3delta_dzizjzks

Helper method for calculating the third composition derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \delta}{\partial x_i \partial x_j \partial x_k}\right)_{T, P, x_{m\ne i,j,k}} = 0$$

Returns

d3delta_dzizjzks [list[list[list[float]]]] Third composition derivative of *epsilon* of each component, [m^6/mol^5]

Notes

This derivative is checked numerically.

property d3epsilon_dninjnks

Helper method for calculating the third partial mole number derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \epsilon}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m \ne i,j,k}} = 4b^0(3b^0 - b_i^0 - b_j^0 - b_k^0) - 2c(6b^0 - 2(b_i^0 + b_j^0 + b_k^0) + 3c - (c_i + c_j + c_k)) + 2(b^0 - b_i^0)(2$$

Returns

d3epsilon_dninjnks [list[list[list[float]]]] Third mole number derivative of *epsilon* of each component, [m^6/mol^5]

Notes

This derivative is checked numerically.

property d3epsilon_dzizjzks

Helper method for calculating the third composition derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \epsilon}{\partial x_i \partial x_j \partial x_k}\right)_{T,P,x_{m \ne i,j,k}} = 0$$

Returns

d2epsilon_dzizjzks [list[list[list[float]]]] Composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dns

Helper method for calculating the mole number derivatives of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \delta}{\partial n_i}\right)_{T,P,n_{i \ne j}} = 2(c_i + b_i^0) - \delta$$

Returns

ddelta_dns [list[float]] Mole number derivative of *delta* of each component, [m^3/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dzs

Helper method for calculating the composition derivatives of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \delta}{\partial x_i}\right)_{T,P,x_{i \ne j}} = 2(c_i + b_i^0)$$

Returns

ddelta_dzs [list[float]] Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property depsilon_dns

Helper method for calculating the mole number derivatives of *epsilon*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \epsilon}{\partial n_i}\right)_{T,P,n_{i\neq j}} = 2b^0(b^0 - b_i^0) - c(2b^0 - 2b_i^0 + c - c_i) - (c - c_i)(2b^0 + c)$$

Returns

depsilon_dns [list[float]] Composition derivative of *epsilon* of each component, [m^6/mol^3]

Notes

This derivative is checked numerically.

property depsilon_dzs

Helper method for calculating the composition derivatives of *epsilon*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \epsilon}{\partial x_i}\right)_{T,P,x_{i\neq j}} = c_i(2b_i^0 + c) + c(2b_i^0 + c_i) - 2b^0 b_i^0$$

Returns

depsilon_dzs [list[float]] Composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

eos_pure

alias of `thermo.eos.PRTranslated`

Peng Robinson Translated-Consistent

class thermo.eos_mix.**PRMIXTranslatedConsistent**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *cs=None*, *alpha_coeffs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: `thermo.eos_alpha_functions.Twu91_a_alpha`, `thermo.eos_mix.PRMIXTranslated`

Class for solving the volume translated Le Guennec, Privat, and Jaubert revision of the Peng-Robinson equation of state according to [1].

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v + c - b} - \frac{a\alpha(T)}{(v + c)(v + c + b) + b(v + c - b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2 T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha_i = \left(\frac{T}{T_c}\right)^{c_3(c_2-1)} e^{c_1\left(-\left(\frac{T}{T_c}\right)^{c_2 c_3}+1\right)}$$

If *c* is not provided, they are estimated as:

$$c = \frac{RT_c}{P_c}(0.0198\omega - 0.0065)$$

If *alpha_coeffs* is not provided, the parameters *L* and *M* are estimated from the acentric factor as follows:

$$L = 0.1290\omega^2 + 0.6039\omega + 0.0877$$

$$M = 0.1760\omega^2 - 0.2600\omega + 0.8884$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters, [m^3/mol]

alpha_coeffs [list[tuple(float[3])], optional] Coefficients L, M, N (also called C1, C2, C3) of TWU 1991 form, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = PRMIXTranslatedConsistent(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5,␣
↪46.04E5], omegas=[0.04, 0.011], zs=[0.2, 0.8], kijs=[[0,0.03],[0.03,0]])
>>> eos.V_l, eos.V_g
(3.675235812e-05, 0.00059709319879)
>>> eos.fugacities_l, eos.fugacities_g
([443454.9336, 106184.004057], [184122.74082, 563037.785])
```

Methods

eos_pure	alias of *thermo.eos.PRTranslatedConsistent*

eos_pure
alias of *thermo.eos.PRTranslatedConsistent*

Peng Robinson Translated (Pina-Martinez, Privat, and Jaubert Variant)

class thermo.eos_mix.**PRMIXTranslatedPPJP**(*Tcs, Pcs, omegas, zs, kijs=None, cs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_mix.PRMIXTranslated*

Class for solving the Pina-Martinez, Privat, Jaubert, and Peng revision of the Peng-Robinson equation of state.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{v+c-b} - \frac{a\alpha(T)}{(v+c)(v+c+b)+b(v+c-b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$$

$$b_i = 0.07780\frac{RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = [1 + \kappa_i(1 - \sqrt{T_{r,i}})]^2$$

$$\kappa_i = 0.3919 + 1.4996\omega - 0.2721\omega^2 + 0.1063\omega^3$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = PRMIXTranslatedPPJP(T=115, P=1E6, cs=[-4.4e-6, -4.35e-6], Tcs=[126.1, 190.
↪6], Pcs=[33.94E5, 46.04E5], omegas=[0.04, 0.011], zs=[0.2, 0.8], kijs=[[0,0.03],
↪[0.03,0]])
>>> eos.V_l, eos.V_g
(3.8989032701e-05, 0.00059686183724)
>>> eos.fugacities_l, eos.fugacities_g
([444791.13707, 104520.280997], [184782.600238, 563352.147])
```

Methods

eos_pure	alias of *thermo.eos.PRTranslatedPPJP*

eos_pure
alias of *thermo.eos.PRTranslatedPPJP*

7.8.3 SRK Family EOSs

Standard SRK

class thermo.eos_mix.**SRKMIX**(*Tcs, Pcs, omegas, zs, kijs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_mix.EpsilonZeroMixingRules*, *thermo.eos_mix.GCEOSMIX*, *thermo.eos.SRK*

Class for solving the Soave-Redlich-Kwong cubic equation of state for a mixture of any number of compounds. Solves the EOS on initialization and calculates fugacities for all components in all phases.

The implemented method here is *fugacity_coefficients*, which implements the formula for fugacity coefficients in a mixture as given in [1]. Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748 \cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664 \cdot RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = \left[1 + m_i\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\right]^2$$

$$m_i = 0.480 + 1.574\omega_i - 0.176\omega_i^2$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1], [2], [3]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> SRK_mix = SRKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> SRK_mix.V_l, SRK_mix.V_g
(4.1047569614e-05, 0.0007110158049)
```

Methods

`a_alpha_and_derivatives_vectorized`(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the SRK EOS.
`a_alphas_vectorized`(T)	Method to calculate the pure-component *a_alphas* for the SRK EOS.
`dlnphis_dP`(phase)	Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture for the SRK EOS.
`dlnphis_dT`(phase)	Formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture for the SRK equation of state.
`eos_pure`	alias of `thermo.eos.SRK`
`fugacity_coefficients`(Z)	Literature formula for calculating fugacity coefficients for each species in a mixture.

a_alpha_and_derivatives_vectorized(*T*)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the SRK EOS.

This vectorized implementation is added for extra speed.

$$a\alpha = a\left(m\left(-\sqrt{\frac{T}{Tc}}+1\right)+1\right)^2$$

$$\frac{da\alpha}{dT} = \frac{am}{T}\sqrt{\frac{T}{Tc}}\left(m\left(\sqrt{\frac{T}{Tc}}-1\right)-1\right)$$

$$\frac{d^2a\alpha}{dT^2} = \frac{am\sqrt{\frac{T}{Tc}}}{2T^2}\left(m+1\right)$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

`a_alphas_vectorized`(*T*)

Method to calculate the pure-component *a_alphas* for the SRK EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a\left(m\left(-\sqrt{\frac{T}{Tc}}+1\right)+1\right)^2$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

`dlnphis_dP`(*phase*)

Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture for the SRK EOS. Verified numerically.

$$\left(\frac{\partial \ln \phi_i}{\partial P}\right)_{T,nj\neq i}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dP [float] Pressure derivatives of log fugacity coefficient for each species, [1/Pa]

Notes

This expression was derived using SymPy and optimized with the *cse* technique.

dlnphis_dT(*phase*)

Formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture for the SRK equation of state. Verified numerically.

$$\left(\frac{\partial \ln \phi_i}{\partial T}\right)_{P,nj \ne i}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dT [float] Temperature derivatives of log fugacity coefficient for each species, [1/K]

Notes

This expression was derived using SymPy and optimized with the *cse* technique.

eos_pure

alias of `thermo.eos.SRK`

fugacity_coefficients(*Z*)

Literature formula for calculating fugacity coefficients for each species in a mixture. Verified numerically. Applicable to most derivatives of the SRK equation of state as well. Called by `fugacities` on initialization, or by a solver routine which is performing a flash calculation.

$$\ln \hat \phi_i = \frac{B_i}{B}(Z-1) - \ln(Z-B) + \frac{A}{B}\left[\frac{B_i}{B} - \frac{2}{a \alpha}\sum_i y_i(a\alpha)_{ij}\right]\ln\left(1+\frac{B}{Z}\right)$$

$$A=\frac{a\alpha P}{R^2T^2}$$

$$B = \frac{bP}{RT}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

log_phis [float] Log fugacity coefficient for each species, [-]

Twu SRK (1995)

class thermo.eos_mix.**TWUSRKMIX**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: `thermo.eos_alpha_functions.TwuSRK95_a_alpha`, `thermo.eos_mix.SRKMIX`

Class for solving the Twu variant of the Soave-Redlich-Kwong cubic equation of state for a mixture. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664\cdot RT_{c,i}}{P_{c,i}}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$\alpha_i = \alpha^{(0,i)} + \omega_i(\alpha^{(1,i)} - \alpha^{(0,i)})$$

$$\alpha^{(\text{0 or 1, i})} = T_{r,i}^{N(M-1)}\exp[L(1-T_{r,i}^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.141599, 0.919422, 2.496441

L1, M1, N1 = 0.500315, 0.799457, 3.291790

For supercritical conditions:

L0, M0, N0 = 0.441411, 6.500018, -0.20

L1, M1, N1 = 0.032580, 1.289098, -8.0

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T. Claimed to be more accurate than the SRK equation.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = TWUSRKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(4.1087927542e-05, 0.00071170732525)
>>> eos.fugacities_l, eos.fugacities_g
([809692.830826, 74093.6388157], [441783.431489, 362470.3174107])
```

Methods

eos_pure	alias of *thermo.eos.TWUSRK*

eos_pure
alias of *thermo.eos.TWUSRK*

API SRK

class thermo.eos_mix.**APISRKMIX**(*Tcs, Pcs, zs, omegas=None, kijs=None, T=None, P=None, V=None, S1s=None, S2s=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_mix.SRKMIX*, *thermo.eos.APISRK*

Class for solving the Refinery Soave-Redlich-Kwong cubic equation of state for a mixture of any number of compounds, as shown in the API Databook [1]. Subclasses *APISRK*. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664\cdot RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = \left[1 + S_{1,i}\left(1-\sqrt{T_{r,i}}\right) + S_{2,i}\frac{1-\sqrt{T_{r,i}}}{\sqrt{T_{r,i}}}\right]^2$$

$$S_{1,i} = 0.48508 + 1.55171\omega_i - 0.15613\omega_i^2 \text{ if S1 is not tabulated}$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

S1s [float, optional] Fit constant or estimated from acentric factor if not provided [-]

S2s [float, optional] Fit constant or 0 if not provided [-]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = APISRKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(4.101592310e-05, 0.00071046883030)
>>> eos.fugacities_l, eos.fugacities_g
([817882.3033, 71620.4823812], [442158.29113, 361519.79877])
```

Methods

eos_pure	alias of thermo.eos.APISRK

eos_pure
alias of thermo.eos.APISRK

SRK Translated

class thermo.eos_mix.**SRKMIXTranslated**(*Tcs, Pcs, omegas, zs, kijs=None, cs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: thermo.eos_mix.SRKMIX

Class for solving the volume translated Soave-Redlich-Kwong cubic equation of state for a mixture of any number of compounds. Subclasses SRKMIX. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V+c-b} - \frac{a\alpha(T)}{(V+c)(V+c+b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

$$b = \sum_i z_i b_i$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664\cdot RT_{c,i}}{P_{c,i}}$$

$$\alpha(T)_i = \left[1 + m_i\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\right]^2$$

$$m_i = 0.480 + 1.574\omega_i - 0.176\omega_i^2$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters; always zero in the original implementation, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = SRKMIXTranslated(T=115, P=1E6, cs=[-4.4e-6, -4.35e-6], Tcs=[126.1, 190.6],
↪ Pcs=[33.94E5, 46.04E5], omegas=[0.04, 0.011], zs=[0.2, 0.8], kijs=[[0,0.03],[0.
↪03,0]])
>>> eos.V_l, eos.V_g
(4.35928920e-05, 0.00060927202)
```

Attributes

`d2delta_dninjs` Helper method for calculating the second mole number derivatives (hessian) of *delta*.

`d2delta_dzizjs` Helper method for calculating the second composition derivatives (hessian) of *delta*.

`d2epsilon_dninjs` Helper method for calculating the second mole number derivatives (hessian) of *epsilon*.

`d2epsilon_dzizjs` Helper method for calculating the second composition derivatives (hessian) of *epsilon*.

`d3delta_dninjnks` Helper method for calculating the third partial mole number derivatives of *delta*.

`d3delta_dzizjzks` Helper method for calculating the third composition derivatives of *delta*.

`d3epsilon_dninjnks` Helper method for calculating the third partial mole number derivatives of *epsilon*.

`d3epsilon_dzizjzks` Helper method for calculating the third composition derivatives of *epsilon*.

`ddelta_dns` Helper method for calculating the mole number derivatives of *delta*.

`ddelta_dzs` Helper method for calculating the composition derivatives of *delta*.

`depsilon_dns` Helper method for calculating the mole number derivatives of *epsilon*.

`depsilon_dzs` Helper method for calculating the composition derivatives of *epsilon*.

Methods

eos_pure	alias of thermo.eos.SRKTranslated

property d2delta_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial^2 \delta}{\partial n_i \partial n_j}\right)_{T,P,n_{k\ne i,j}} = \left((b^0 - c_i - c_j) + 4c - b^0_i - b^0_j\right)$$

Returns

d2delta_dninjs [list[list[float]]] Second mole number derivative of *delta* of each component, [m^3/mol^3]

Notes

This derivative is checked numerically.

property d2delta_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \delta}{\partial x_i \partial x_j}\right)_{T,P,x_{k\ne i,j}} = 0$$

Returns

d2delta_dzizjs [list[float]] Second Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property d2epsilon_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \epsilon}{\partial n_i n_j}\right)_{T,P,n_{k\ne i,j}} = b^0(2c - c_i - c_j) + c(2b^0 - b^0_i - b^0_j) + 2c(2c - c_i - c_j) + (b^0 - b^0_i)(c - c_j) + (b^0 - b^0_j)(c - c_i) +$$

Returns

d2epsilon_dninjs [list[list[float]]] Second mole number derivative of *epsilon* of each component, [m^6/mol^4]

Notes

This derivative is checked numerically.

property d2epsilon_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \epsilon}{\partial x_i \partial x_j}\right)_{T, P, x_{k\ne i,j}} = b^0_i c_j + b^0_j c_i + 2c_i c_j$$

Returns

d2epsilon_dzizjs [list[list[float]]] Second composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

property d3delta_dninjnks

Helper method for calculating the third partial mole number derivatives of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial^3 \delta}{\partial n_i \partial n_j \partial n_k}\right)_{T, P, n_{m\ne i,j,k}} = -6b^0 + 2(b^0_i + b^0_j + b^0_k) + -12c + 4(c_i + c_j + c_k)$$

Returns

d3delta_dninjnks [list[list[list[float]]]] Third mole number derivative of *delta* of each component, [m^3/mol^4]

Notes

This derivative is checked numerically.

property d3delta_dzizjzks

Helper method for calculating the third composition derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \delta}{\partial x_i \partial x_j \partial x_k}\right)_{T, P, x_{m\ne i,j,k}} = 0$$

Returns

d3delta_dzizjzks [list[list[list[float]]]] Third composition derivative of *epsilon* of each component, [m^6/mol^5]

Notes

This derivative is checked numerically.

property d3epsilon_dninjnks

Helper method for calculating the third partial mole number derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \epsilon}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\ne i,j,k}} = -2b^0(3c - c_i - c_j - c_k) - 2c(3b^0 - b_i^0 - b_j^0 - b_k^0) - 4c(3c - c_i - c_j - c_k) - (b^0 - b_i^0)(2c$$

Returns

d3epsilon_dninjnks [list[list[list[float]]]] Third mole number derivative of *epsilon* of each component, [m^6/mol^5]

Notes

This derivative is checked numerically.

property d3epsilon_dzizjzks

Helper method for calculating the third composition derivatives of *epsilon*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \epsilon}{\partial x_i \partial x_j \partial x_k}\right)_{T,P,x_{m\ne i,j,k}} = 0$$

Returns

d2epsilon_dzizjzks [list[list[list[float]]]] Composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dns

Helper method for calculating the mole number derivatives of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \delta}{\partial n_i}\right)_{T,P,n_{i\ne j}} = (2c_i + b_i^0) - \delta$$

Returns

ddelta_dns [list[float]] Mole number derivative of *delta* of each component, [m^3/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dzs

Helper method for calculating the composition derivatives of *delta*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \delta}{\partial x_i}\right)_{T,P,x_{i\ne j}} = 2(c_i + b_i^0)$$

Returns

ddelta_dzs [list[float]] Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property depsilon_dns

Helper method for calculating the mole number derivatives of *epsilon*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \epsilon}{\partial n_i}\right)_{T,P,n_{i\neq j}} = -b^0(c - c_i) - c(b^0 - b_i^0) - 2c(c - c_i)$$

Returns

depsilon_dns [list[float]] Composition derivative of *epsilon* of each component, [m^6/mol^3]

Notes

This derivative is checked numerically.

property depsilon_dzs

Helper method for calculating the composition derivatives of *epsilon*. Note this is independent of the phase. b^0 refers to the original b parameter not involving any translation.

$$\left(\frac{\partial \epsilon}{\partial x_i}\right)_{T,P,x_{i\neq j}} = c_i b^0 + 2cc_i + b_i c$$

Returns

depsilon_dzs [list[float]] Composition derivative of *epsilon* of each component, [m^6/mol^2]

Notes

This derivative is checked numerically.

eos_pure

alias of `thermo.eos.SRKTranslated`

SRK Translated-Consistent

class thermo.eos_mix.**SRKMIXTranslatedConsistent**(*Tcs*, *Pcs*, *omegas*, *zs*, *kijs=None*, *cs=None*, *alpha_coeffs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: `thermo.eos_alpha_functions.Twu91_a_alpha`, `thermo.eos_mix.SRKMIXTranslated`

Class for solving the volume translated Le Guennec, Privat, and Jaubert revision of the SRK equation of state according to [1].

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V + c - b} - \frac{a\alpha(T)}{(V + c)(V + c + b)}$$

$$a \alpha = \sum_i \sum_j z_i z_j {(a\alpha)}_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_{i}(a\alpha)_{j}}$$

$$\alpha_i = \left(\frac{T}{T_{c,i}}\right)^{c_{3} \left(c_{2} - 1\right)} e^{c_{1} \left(- \left(\frac{T}{T_{c,i}}\right)^{c_{2} c_{3}} + 1\right)}$$

$$b = \sum_i z_i b_i$$

$$a_i =\left(\frac{R^2(T_{c,i})^{2}}{9(\sqrt[3]{2}-1)P_{c,i}} \right) =\frac{0.42748\cdot R^2(T_{c,i})^{2}}{P_{c,i}}$$

$$b_i =\left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} =\frac{0.08664\cdot R T_{c,i}}{P_{c,i}}$$

If *cs* is not provided, they are estimated as:

$$c = \frac{R T_c}{P_c}(0.0172\omega - 0.0096)$$

If *alpha_coeffs* is not provided, the parameters *L* and *M* are estimated from each of the acentric factors as follows:

$$L = 0.0947\omega^2 + 0.6871\omega + 0.1508$$

$$M = 0.1615\omega^2 - 0.2349\omega + 0.8876$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters, [m^3/mol]

alpha_coeffs [list[list[float]]] Coefficients for `thermo.eos_alpha_functions.Twu91_a_alpha`, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = SRKMIXTranslatedConsistent(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5,
↪ 46.04E5], omegas=[0.04, 0.011], zs=[0.2, 0.8], kijs=[[0,0.03],[0.03,0]])
>>> eos.V_l, eos.V_g
(3.591044498e-05, 0.0006020501621)
```

Methods

eos_pure	alias of *thermo.eos.SRKTranslatedConsistent*

eos_pure
alias of *thermo.eos.SRKTranslatedConsistent*

MSRK Translated

class thermo.eos_mix.**MSRKMIXTranslated**(*Tcs, Pcs, omegas, zs, kijs=None, cs=None, alpha_coeffs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_alpha_functions.Soave_1979_a_alpha*, *thermo.eos_mix.SRKMIXTranslatedConsistent*

Class for solving the volume translated Soave (1980) alpha function, revision of the Soave-Redlich-Kwong equation of state for a pure compound according to [1]. Uses two fitting parameters N and M to more accurately fit the vapor pressure of pure species.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V+c-b} - \frac{a\alpha(T)}{(V+c)(V+c+b)}$$

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

$$\alpha(T)_i = 1 + (1-T_{r,i})(M + \frac{N}{T_{r,i}})$$

$$b = \sum_i z_i b_i$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664\cdot RT_{c,i}}{P_{c,i}}$$

This is an older correlation that offers lower accuracy on many properties which were sacrificed to obtain the vapor pressure accuracy. The alpha function of this EOS does not meet any of the consistency requriements for alpha functions.

Coefficients can be found in [2], or estimated with the method in [3]. The estimation method in [3] works as follows, using the acentric factor and true critical compressibility:

$$M = 0.4745 + 2.7349(\omega Z_c) + 6.0984(\omega Z_c)^2$$

$$N = 0.0674 + 2.1031(\omega Z_c) + 3.9512(\omega Z_c)^2$$

An alternate estimation scheme is provided in [1], which provides analytical solutions to calculate the parameters *M* and *N* from two points on the vapor pressure curve, suggested as 10 mmHg and 1 atm. This is used as an estimation method here if the parameters are not provided, and the two vapor pressure points are obtained from the original SRK equation of state.

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters, [m^3/mol]

alpha_coeffs [list[list[float]]] Coefficients for `thermo.eos_alpha_functions.Soave_1979_a_alpha`, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1], [2], [3]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = MSRKMIXTranslated(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.
↪04E5], omegas=[0.04, 0.011], zs=[0.2, 0.8], kijs=[[0,0.03],[0.03,0]])
>>> eos.V_l, eos.V_g
(3.9222990198e-05, 0.00060438075638)
```

Methods

eos_pure	alias of *thermo.eos.MSRKTranslated*

eos_pure
alias of *thermo.eos.MSRKTranslated*

7.8.4 Cubic Equation of State with Activity Coefficients

class thermo.eos_mix.**PSRK**(*Tcs, Pcs, omegas, zs, alpha_coeffs, ge_model, kijs=None, cs=None, T=None, P=None, V=None, fugacities=True, only_l=False, only_g=False*)

Bases: *thermo.eos_alpha_functions.Mathias_Copeman_poly_a_alpha*, *thermo.eos_mix.PSRKMixingRules*, *thermo.eos_mix.SRKMIXTranslated*

Class for solving the Predictive Soave-Redlich-Kwong [1] equation of state for a mixture of any number of compounds. Solves the EOS on initialization.

Two of *T*, *P*, and *V* are needed to solve the EOS.

> **Warning:** This class is not complete! Fugacities and their derivatives among others are not yet implemented.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V(V+b)}$$

$$b = \sum_i z_i b_i$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748 \cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664 \cdot RT_{c,i}}{P_{c,i}}$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

omegas [float] Acentric factors of all compounds, [-]

zs [float] Overall mole fractions of all species, [-]

alpha_coeffs [list[list[float]]] Coefficients for *thermo.eos_alpha_functions.Mathias_Copeman_poly_a_alpha*, [-]

ge_model [*thermo.activity.GibbsExcess* object] Excess Gibbs free energy model; to match the *PSRK* model, this is a *thermo.unifac.UNIFAC* object, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

cs [list[float], optional] Volume translation parameters; always zero in the original implementation, [m^3/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

References

[1]

Examples

T-P initialization, equimolar CO2, n-hexane:

```
>>> from thermo.unifac import UNIFAC, PSRKIP, PSRKSG
>>> Tcs = [304.2, 507.4]
>>> Pcs = [7.37646e6, 3.014419e6]
>>> omegas = [0.2252, 0.2975]
>>> zs = [0.5, 0.5]
>>> Mathias_Copeman_coeffs = [[-1.7039, 0.2515, 0.8252, 1.0], [2.9173, -1.4411, 1.
↪1061, 1.0]]
>>> T = 313.
>>> P = 1E6
>>> ge_model = UNIFAC.from_subgroups(T=T, xs=zs, chemgroups=[{117: 1}, {1:2, 2:4}],␣
↪subgroups=PSRKSG, interaction_data=PSRKIP, version=0)
>>> eos = PSRK(Tcs=Tcs, Pcs=Pcs, omegas=omegas, zs=zs, ge_model=ge_model, alpha_
↪coeffs=Mathias_Copeman_coeffs, T=T, P=P)
>>> eos
```

(continues on next page)

(continued from previous page)

```
PSRK(Tcs=[304.2, 507.4], Pcs=[7376460.0, 3014419.0], omegas=[0.2252, 0.2975],␣
↪kijs=[[0.0, 0.0], [0.0, 0.0]], alpha_coeffs=[[-1.7039, 0.2515, 0.8252, 1.0], [2.
↪9173, -1.4411, 1.1061, 1.0]], cs=[0.0, 0.0], ge_model=UNIFAC(T=313.0, xs=[0.5, 0.
↪5], rs=[1.3, 4.4998000000000005], qs=[0.982, 3.856], Qs=[0.848, 0.54, 0.982],␣
↪vs=[[0, 2], [0, 4], [1, 0]], psi_abc=([[0.0, 0.0, 919.8], [0.0, 0.0, 919.8], [-38.
↪672, -38.672, 0.0]], [[0.0, 0.0, -3.9132], [0.0, 0.0, -3.9132], [0.8615, 0.8615,␣
↪0.0]], [[0.0, 0.0, 0.0046309], [0.0, 0.0, 0.0046309], [-0.0017906, -0.0017906, 0.
↪0]]), version=0), zs=[0.5, 0.5], T=313.0, P=1000000.0)
>>> eos.phase, eos.V_l, eos.V_g
('l/g', 0.000110889753959, 0.00197520225546)
```

Methods

eos_pure	alias of *thermo.eos.SRKTranslated*

eos_pure
alias of *thermo.eos.SRKTranslated*

7.8.5 Van der Waals Equation of State

class thermo.eos_mix.**VDWMIX**(*Tcs*, *Pcs*, *zs*, *kijs=None*, *T=None*, *P=None*, *V=None*, *omegas=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: *thermo.eos_mix.EpsilonZeroMixingRules*, *thermo.eos_mix.GCEOSMIX*, *thermo.eos.VDW*

Class for solving the Van der Waals [1] [2] cubic equation of state for a mixture of any number of compounds. Solves the EOS on initialization and calculates fugacities for all components in all phases.

Two of T, P, and V are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a}{V^2}$$

$$a = \sum_i \sum_j z_i z_j a_{ij}$$

$$b = \sum_i z_i b_i$$

$$a_{ij} = (1-k_{ij})\sqrt{a_i a_j}$$

$$a_i = \frac{27}{64}\frac{(RT_{c,i})^2}{P_{c,i}}$$

$$b_i = \frac{RT_{c,i}}{8P_{c,i}}$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

omegas [float, optional] Acentric factors of all compounds - Not used in equation of state!, [-]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

For P-V initializations, a numerical solver is used to find T.

References

[1], [2]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = VDWMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], zs=[0.5,␣
↪0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(5.881369844883e-05, 0.00077708723758)
>>> eos.fugacities_l, eos.fugacities_g
([854533.266920, 207126.8497276], [448470.736338, 397826.543999])
```

Attributes

d2delta_dninjs Helper method for calculating the second mole number derivatives (hessian) of *delta*.

d2delta_dzizjs Helper method for calculating the second composition derivatives (hessian) of *delta*.

d3delta_dninjnks Helper method for calculating the third partial mole number derivatives of *delta*.

ddelta_dns Helper method for calculating the mole number derivatives of *delta*.

ddelta_dzs Helper method for calculating the composition derivatives of *delta*.

Methods

a_alpha_and_derivatives_vectorized(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the VDW EOS.
a_alphas_vectorized(T)	Method to calculate the pure-component *a_alphas* for the VDW EOS.
dlnphis_dP(phase)	Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture for the VDW EOS.
dlnphis_dT(phase)	Formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture for the VDW equation of state.
eos_pure	alias of *thermo.eos.VDW*
fugacity_coefficients(Z)	Literature formula for calculating fugacity coefficients for each species in a mixture.

a_alpha_and_derivatives_vectorized(*T*)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the VDW EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a$$

$$\frac{da\alpha}{dT} = 0$$

$$\frac{d^2a\alpha}{dT^2} = 0$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alphas_vectorized(*T*)

Method to calculate the pure-component *a_alphas* for the VDW EOS. This vectorized implementation is added for extra speed.

$$a\alpha = a$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

property d2delta_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \delta}{\partial n_i \partial n_j}\right)_{T,P,n_{k\neq i,j}} = 0$$

Returns

d2delta_dninjs [list[list[float]]] Second mole number derivative of *delta* of each component, [m^3/mol^3]

Notes

This derivative is checked numerically.

property d2delta_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \delta}{\partial x_i \partial x_j}\right)_{T,P,x_{k\neq i,j}} = 0$$

Returns

d2delta_dzizjs [list[float]] Second Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property d3delta_dninjnks

Helper method for calculating the third partial mole number derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \delta}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\neq i,j,k}} = 0$$

Returns

d3delta_dninjnks [list[list[list[float]]]] Third mole number derivative of *delta* of each component, [m^3/mol^4]

Notes

This derivative is checked numerically.

property ddelta_dns

Helper method for calculating the mole number derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial \delta}{\partial n_i}\right)_{T,P,n_{i\neq j}} = 0$$

Returns

ddelta_dns [list[float]] Mole number derivative of *delta* of each component, [m^3/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dzs

Helper method for calculating the composition derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial \delta}{\partial x_i}\right)_{T,P,x_{i\ne j}} = 0$$

Returns

ddelta_dzs [list[float]] Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

dlnphis_dP(*phase*)

Generic formula for calculating the pressure derivaitve of log fugacity coefficients for each species in a mixture for the VDW EOS. Verified numerically.

$$\left(\frac{\partial \ln \phi_i}{\partial P}\right)_{T,nj\ne i}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dP [float] Pressure derivatives of log fugacity coefficient for each species, [1/Pa]

Notes

This expression was derived using SymPy and optimized with the *cse* technique.

dlnphis_dT(*phase*)

Formula for calculating the temperature derivaitve of log fugacity coefficients for each species in a mixture for the VDW equation of state. Verified numerically.

$$\left(\frac{\partial \ln \phi_i}{\partial T}\right)_{P,nj\ne i}$$

Parameters

phase [str] One of 'l' or 'g', [-]

Returns

dlnphis_dT [float] Temperature derivatives of log fugacity coefficient for each species, [1/K]

Notes

This expression was derived using SymPy and optimized with the *cse* technique.

eos_pure
alias of *thermo.eos.VDW*

fugacity_coefficients(Z)
Literature formula for calculating fugacity coefficients for each species in a mixture. Verified numerically. Called by *fugacities* on initialization, or by a solver routine which is performing a flash calculation.

$$\ln \hat \phi_i = \frac{b_i}{V-b} - \ln\left[Z\left(1 - \frac{b}{V}\right)\right] - \frac{2\sqrt{aa_i}}{RTV}$$

Parameters

Z [float] Compressibility of the mixture for a desired phase, [-]

Returns

log_phis [float] Log fugacity coefficient for each species, [-]

References

[1]

7.8.6 Redlich-Kwong Equation of State

class thermo.eos_mix.**RKMIX**(*Tcs*, *Pcs*, *zs*, *omegas=None*, *kijs=None*, *T=None*, *P=None*, *V=None*, *fugacities=True*, *only_l=False*, *only_g=False*)
Bases: *thermo.eos_mix.EpsilonZeroMixingRules*, *thermo.eos_mix.GCEOSMIX*, *thermo.eos.RK*

Class for solving the Redlich Kwong [1] [2] cubic equation of state for a mixture of any number of compounds. Subclasses *thermo.eos.RK* . Solves the EOS on initialization and calculates fugacities for all components in all phases. Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{V-b} - \frac{a}{V\sqrt{T}(V+b)}$$

$$a = \sum_i \sum_j z_i z_j a_{ij}$$

$$b = \sum_i z_i b_i$$

$$a_{ij} = (1-k_{ij})\sqrt{a_i a_j}$$

$$a_i = \left(\frac{R^2(T_{c,i})^2}{9(\sqrt[3]{2}-1)P_{c,i}}\right) = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$$

$$b_i = \left(\frac{(\sqrt[3]{2}-1)}{3}\right)\frac{RT_{c,i}}{P_{c,i}} = \frac{0.08664\cdot RT_{c,i}}{P_{c,i}}$$

Parameters

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

Pcs [float] Critical pressures of all compounds, [Pa]

zs [float] Overall mole fractions of all species, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

omegas [float, optional] Acentric factors of all compounds - Not used in this equation of state!, [-]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

The PV solution for T is iterative.

References

[1], [2]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = RKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], zs=[0.5,␣
↪0.5], kijs=[[0,0],[0,0]])
>>> eos.V_l, eos.V_g
(4.048414781e-05, 0.00070060605863)
```

Attributes

d2delta_dninjs Helper method for calculating the second mole number derivatives (hessian) of *delta*.

d2delta_dzizjs Helper method for calculating the second composition derivatives (hessian) of *delta*.

d3delta_dninjnks Helper method for calculating the third partial mole number derivatives of *delta*.

ddelta_dns Helper method for calculating the mole number derivatives of *delta*.

ddelta_dzs Helper method for calculating the composition derivatives of *delta*.

Methods

`a_alpha_and_derivatives_vectorized(T)`	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the RK EOS.
`a_alphas_vectorized(T)`	Method to calculate the pure-component *a_alphas* for the RK EOS.
`eos_pure`	alias of `thermo.eos.RK`

a_alpha_and_derivatives_vectorized(*T*)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the RK EOS. This vectorized implementation is added for extra speed.

$$a\alpha = \frac{a}{\sqrt{\frac{T}{Tc}}}$$

$$\frac{da\alpha}{dT} = -\frac{a}{2T\sqrt{\frac{T}{Tc}}}$$

$$\frac{d^2a\alpha}{dT^2} = \frac{3a}{4T^2\sqrt{\frac{T}{Tc}}}$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

Examples

```
>>> eos = RKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.a_alpha_and_derivatives_vectorized(115)
([0.1449810919468, 0.30019773677], [-0.000630352573681, -0.00130520755121], [8.
↪2219900915e-06, 1.7024446320e-05])
```

a_alphas_vectorized(*T*)

Method to calculate the pure-component *a_alphas* for the RK EOS. This vectorized implementation is added for extra speed.

$$a\alpha = \frac{a}{\sqrt{\frac{T}{Tc}}}$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Examples

```
>>> eos = RKMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5],␣
↪omegas=[0.04, 0.011], zs=[0.5, 0.5], kijs=[[0,0],[0,0]])
>>> eos.a_alphas_vectorized(115)
[0.1449810919468, 0.30019773677]
```

property d2delta_dninjs

Helper method for calculating the second mole number derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \delta}{\partial n_i \partial n_j}\right)_{T,P,n_{k\ne i,j}} = 2b - b_i - b_j$$

Returns

d2delta_dninjs [list[list[float]]] Second mole number derivative of *delta* of each component, [m^3/mol^3]

Notes

This derivative is checked numerically.

property d2delta_dzizjs

Helper method for calculating the second composition derivatives (hessian) of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^2 \delta}{\partial x_i \partial x_j}\right)_{T,P,x_{k\ne i,j}} = 0$$

Returns

d2delta_dzizjs [list[float]] Second Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

property d3delta_dninjnks

Helper method for calculating the third partial mole number derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial^3 \delta}{\partial n_i \partial n_j \partial n_k}\right)_{T,P,n_{m\ne i,j,k}} = 2(-3b + b_i + b_j + b_k)$$

Returns

d3delta_dninjnks [list[list[list[float]]]] Third mole number derivative of *delta* of each component, [m^3/mol^4]

Notes

This derivative is checked numerically.

property ddelta_dns

Helper method for calculating the mole number derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial \delta}{\partial n_i}\right)_{T,P,n_{i\ne j}} = (b_i - b)$$

Returns

ddelta_dns [list[float]] Mole number derivative of *delta* of each component, [m^3/mol^2]

Notes

This derivative is checked numerically.

property ddelta_dzs

Helper method for calculating the composition derivatives of *delta*. Note this is independent of the phase.

$$\left(\frac{\partial \delta}{\partial x_i}\right)_{T,P,x_{i\ne j}} = b_i$$

Returns

ddelta_dzs [list[float]] Composition derivative of *delta* of each component, [m^3/mol]

Notes

This derivative is checked numerically.

eos_pure

alias of `thermo.eos.RK`

7.8.7 Ideal Gas Equation of State

class thermo.eos_mix.IGMIX(*zs*, *T=None*, *P=None*, *V=None*, *Tcs=None*, *Pcs=None*, *omegas=None*, *kijs=None*, *fugacities=True*, *only_l=False*, *only_g=False*)

Bases: `thermo.eos_mix.EpsilonZeroMixingRules`, `thermo.eos_mix.GCEOSMIX`, `thermo.eos.IG`

Class for solving the ideal gas [1] [2] equation of state for a mixture of any number of compounds. Subclasses `thermo.eos.IG`. Solves the EOS on initialization. Two of *T*, *P*, and *V* are needed to solve the EOS.

$$P = \frac{RT}{V}$$

Parameters

zs [list[float]] Overall mole fractions of all species, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

V [float, optional] Molar volume, [m^3/mol]

Tcs [list[float], optional] Critical temperatures of all compounds, [K]

Pcs [list[float], optional] Critical pressures of all compounds, [Pa]

omegas [list[float], optional] Acentric factors of all compounds - Not used in this equation of state!, [-]

kijs [list[list[float]], optional] n*n size list of lists with binary interaction parameters for the Van der Waals mixing rules, default all 0 and not used[-]

fugacities [bool, optional] Whether or not to calculate fugacity related values (phis, log phis, and fugacities); default True, [-]

only_l [bool, optional] When true, if there is a liquid and a vapor root, only the liquid root (and properties) will be set; default False, [-]

only_g [bool, optional] When true, if there is a liquid and a vapor root, only the vapor root (and properties) will be set; default False, [-]

Notes

Many properties of this object are zero. Many of the arguments are not used and are provided for consistency only.

References

[1], [2]

Examples

T-P initialization, nitrogen-methane at 115 K and 1 MPa:

```
>>> eos = IGMIX(T=115, P=1E6, Tcs=[126.1, 190.6], Pcs=[33.94E5, 46.04E5], omegas=[0.
↪04, .008], zs=[0.5, 0.5])
>>> eos.phase, eos.V_g
('g', 0.0009561632010876225)
```

Methods

a_alpha_and_derivatives_vectorized(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for the Ideal Gas EOS.
a_alphas_vectorized(T)	Method to calculate the pure-component *a_alphas* for the Ideal Gas EOS.
eos_pure	alias of *thermo.eos.IG*

a_alpha_and_derivatives_vectorized(T)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for the Ideal Gas EOS. This vectorized implementation is added for extra speed.

$$a\alpha = 0$$

$$\frac{da\alpha}{dT} = 0$$

$$\frac{d^2 a\alpha}{dT^2} = 0$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alphas_vectorized(*T*)

Method to calculate the pure-component *a_alphas* for the Ideal Gas EOS. This vectorized implementation is added for extra speed.

$$a\alpha = 0$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

eos_pure

alias of `thermo.eos.IG`

7.8.8 Different Mixing Rules

class thermo.eos_mix.**EpsilonZeroMixingRules**

Attributes

d2epsilon_dninjs Helper method for calculating the second mole number derivatives (hessian) of *epsilon*.

d2epsilon_dzizjs Helper method for calculating the second composition derivatives (hessian) of *epsilon*.

d3epsilon_dninjnks Helper method for calculating the third partial mole number derivatives of *epsilon*.

depsilon_dns Helper method for calculating the mole number derivatives of *epsilon*.

depsilon_dzs Helper method for calculating the composition derivatives of *epsilon*.

class thermo.eos_mix.**PSRKMixingRules**

Bases: `object`

Methods

`a_alpha_and_derivatives`(T[, full, quick, ...])	Method to calculate *a_alpha* and its first and second derivatives for an EOS with the PSRK mixing rules.

A = -0.6466271649250525

a_alpha_and_derivatives(*T*, *full=True*, *quick=True*, *pure_a_alphas=True*)

Method to calculate *a_alpha* and its first and second derivatives for an EOS with the PSRK mixing rules. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*.

For use in some methods, this returns only *a_alpha* if *full* is False.

$$\alpha = bRT\left[\sum_i \frac{z_i\alpha_i}{b_i RT} + \frac{1}{A}\left(\frac{G^E}{RT} + \sum_i z_i \ln\left(\frac{b}{b_i}\right)\right)\right]$$

$$\frac{\partial \alpha}{\partial T} = RTb\left[\sum_i\left(\frac{z_i\frac{\partial \alpha_i}{\partial T}}{RTb_i} - \frac{z_i\alpha_i}{RT^2 b_i}\right) + \frac{1}{A}\left(\frac{\frac{\partial G^E}{\partial T}}{RT} - \frac{G^E}{RT^2}\right)\right] + \frac{\alpha}{T}$$

$$\frac{\partial^2 \alpha}{\partial T^2} = b\left[\sum_i\left(\frac{z_i\frac{\partial^2 \alpha_i}{\partial T^2}}{b_i} - \frac{2z_i\frac{\partial \alpha_i}{\partial T}}{Tb_i} + \frac{2z_i\alpha_i}{T^2 b_i}\right) + \frac{2}{T}\left[\sum_i\left(\frac{z_i\frac{\partial \alpha_i}{\partial T}}{b_i} - \frac{z_i\alpha_i}{Tb_i}\right) + \frac{1}{A}\left(\frac{\partial G^E}{\partial T} - \frac{G^E}{T}\right)\right] + \frac{1}{A}\left(\frac{\partial^2 G^E}{\partial T^2} - \frac{2}{T}\frac{\partial G^E}{\partial T}\right.\right.$$

Parameters

T [float] Temperature, [K]

full [bool, optional] If False, calculates and returns only *a_alpha*

quick [bool, optional] Only the quick variant is implemented; it is little faster anyhow

pure_a_alphas [bool, optional] Whether or not to recalculate the a_alpha terms of pure components (for the case of mixtures only) which stay the same as the composition changes (i.e in a PT flash), [-]

Returns

a_alpha [float] Coefficient calculated by PSRK-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by PSRK-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by PSRK-specific method, [J^2/mol^2/Pa/K**2]

u = 1.1

7.8.9 Lists of Equations of State

thermo.eos_mix.**eos_mix_list = [<class 'thermo.eos_mix.PRMIX'>, <class 'thermo.eos_mix.SRKMIX'>, <class 'thermo.eos_mix.PR78MIX'>, <class 'thermo.eos_mix.VDWMIX'>, <class 'thermo.eos_mix.PRSVMIX'>, <class 'thermo.eos_mix.PRSV2MIX'>, <class 'thermo.eos_mix.TWUPRMIX'>, <class 'thermo.eos_mix.TWUSRKMIX'>, <class 'thermo.eos_mix.APISRKMIX'>, <class 'thermo.eos_mix.IGMIX'>, <class 'thermo.eos_mix.RKMIX'>, <class 'thermo.eos_mix.PRMIXTranslatedConsistent'>, <class 'thermo.eos_mix.PRMIXTranslatedPPJP'>, <class 'thermo.eos_mix.SRKMIXTranslatedConsistent'>, <class 'thermo.eos_mix.PRMIXTranslated'>, <class 'thermo.eos_mix.SRKMIXTranslated'>]**

List of all exported EOS classes.

thermo.eos_mix.**eos_mix_no_coeffs_list = [<class 'thermo.eos_mix.PRMIX'>, <class 'thermo.eos_mix.SRKMIX'>, <class 'thermo.eos_mix.PR78MIX'>, <class 'thermo.eos_mix.VDWMIX'>, <class 'thermo.eos_mix.TWUPRMIX'>, <class 'thermo.eos_mix.TWUSRKMIX'>, <class 'thermo.eos_mix.IGMIX'>, <class 'thermo.eos_mix.RKMIX'>, <class 'thermo.eos_mix.PRMIXTranslatedConsistent'>, <class 'thermo.eos_mix.PRMIXTranslated'>, <class 'thermo.eos_mix.SRKMIXTranslated'>, <class 'thermo.eos_mix.PRMIXTranslatedPPJP'>, <class 'thermo.eos_mix.SRKMIXTranslatedConsistent'>]**

List of all exported EOS classes that do not require special parameters or can fill in their special parameters from other specified parameters.

7.9 Cubic Equations of State Utilities (thermo.eos_mix_methods)

This file contains a number of overflow methods for EOSs which for various reasons are better implemented as functions. Documentation is not provided for this file and no methods are intended to be used outside this library.

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

7.9.1 Alpha Function Mixing Rules

These are where the bulk of the time is spent in solving the equation of state. For that reason, these functional forms often duplicate functionality but have different performance characteristics.

Implementations which store N^2 matrices for other calculations:

thermo.eos_mix_methods.**a_alpha_aijs_composition_independent**(*a_alphas*, *kijs*)

Calculates the matrix $(a\alpha)_{ij}$ as well as the array $\sqrt{(a\alpha)_i}$ and the matrix $\frac{1}{\sqrt{(a\alpha)_i}\sqrt{(a\alpha)_j}}$.

$$(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

This routine is efficient in both numba and PyPy, but it is generally better to avoid calculating and storing **any** N^2 matrices. However, this particular calculation only depends on T so in some circumstances this can be feasible.

Parameters

a_alphas [list[float]] EOS attractive terms, [J^2/mol^2/Pa]

kijs [list[list[float]]] Constant kijs, [-]

Returns

a_alpha_ijs [list[list[float]]] Matrix of $(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$, [J^2/mol^2/Pa]

a_alpha_roots [list[float]] Array of $\sqrt{(a\alpha)_i}$ values, [J/mol/Pa^0.5]

a_alpha_ij_roots_inv [list[list[float]]] Matrix of $\frac{1}{\sqrt{(a\alpha)_i}\sqrt{(a\alpha)_j}}$, [mol^2*Pa/J^2]

Examples

```
>>> kijs = [[0,.083],[0.083,0]]
>>> a_alphas = [0.2491099357671155, 0.6486495863528039]
>>> a_alpha_ijs, a_alpha_roots, a_alpha_ij_roots_inv = a_alpha_aijs_composition_
↪independent(a_alphas, kijs)
>>> a_alpha_ijs
[[0.249109935767, 0.36861239374], [0.36861239374, 0.64864958635]]
>>> a_alpha_roots
[0.49910914213, 0.80538784840]
>>> a_alpha_ij_roots_inv
[[4.0142919105, 2.487707997796], [2.487707997796, 1.54166443799]]
```

thermo.eos_mix_methods.**a_alpha_aijs_composition_independent_support_zeros**(*a_alphas*, *kijs*)

thermo.eos_mix_methods.**a_alpha_and_derivatives_full**(*a_alphas*, *da_alpha_dTs*, *d2a_alpha_dT2s*, *T*, *zs*, *kijs*, *a_alpha_ijs=None*, *a_alpha_roots=None*, *a_alpha_ij_roots_inv=None*)

Calculates the *a_alpha* term, and its first two temperature derivatives, for an equation of state along with the matrix quantities calculated in the process.

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$\frac{\partial (a\alpha)}{\partial T} = \sum_i \sum_j z_i z_j \frac{\partial (a\alpha)_{ij}}{\partial T}$$

$$\frac{\partial^2 (a\alpha)}{\partial T^2} = \sum_i \sum_j z_i z_j \frac{\partial^2 (a\alpha)_{ij}}{\partial T^2}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$$

$$\frac{\partial (a\alpha)_{ij}}{\partial T} = \frac{\sqrt{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} \left(1 - k_{ij}\right) \left(\frac{\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)}}{2} + \frac{\operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}}{2}\right)}{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}}$$

$$\frac{\partial^2 (a\alpha)_{ij}}{\partial T^2} = - \frac{\sqrt{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} \left(k_{ij} - 1\right) \left(\frac{\left(\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)} + \operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}\right)^{2}}{4 \operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} - \frac{\left(\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)} + \operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}\right) \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right.}}{2 \operatorname{a\alpha_{j}}{\left(T \right)}}}{\operatorname{a\alpha_{i}}{\left(T \right.}}$$

Parameters

a_alphas [list[float]] EOS attractive terms, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

T [float] Temperature, not used, [K]

zs [list[float]] Mole fractions of each species

kijs [list[list[float]]] Constant kijs, [-]

a_alpha_ijs [list[list[float]], optional] Matrix of $(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$, [J^2/mol^2/Pa]

a_alpha_roots [list[float], optional] Array of $\sqrt{(a\alpha)_i}$ values, [J/mol/Pa^0.5]

a_alpha_ij_roots_inv [list[list[float]], optional] Matrix of $\frac{1}{\sqrt{(a\alpha)_i}\sqrt{(a\alpha)_j}}$, [mol^2*Pa/J^2]

Returns

a_alpha [float] EOS attractive term, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alpha_ijs [list[list[float]], optional] Matrix of $(1-k_{ij})\sqrt{(a\alpha)_i(a\alpha)_j}$, [J^2/mol^2/Pa]

da_alpha_dT_ijs [list[list[float]], optional] Matrix of $\frac{\partial(a\alpha)_{ij}}{\partial T}$, [J^2/mol^2/Pa/K]

d2a_alpha_dT2_ijs [list[list[float]], optional] Matrix of $\frac{\partial^2(a\alpha)_{ij}}{\partial T^2}$, [J^2/mol^2/Pa/K^2]

Examples

```
>>> kijs = [[0,.083],[0.083,0]]
>>> zs = [0.1164203, 0.8835797]
>>> a_alphas = [0.2491099357671155, 0.6486495863528039]
>>> da_alpha_dTs = [-0.0005102028006086241, -0.0011131153520304886]
>>> d2a_alpha_dT2s = [1.8651128859234162e-06, 3.884331923127011e-06]
>>> a_alpha, da_alpha_dT, d2a_alpha_dT2, a_alpha_ijs, da_alpha_dT_ijs, d2a_alpha_
↪dT2_ijs = a_alpha_and_derivatives_full(a_alphas=a_alphas, da_alpha_dTs=da_alpha_
↪dTs, d2a_alpha_dT2s=d2a_alpha_dT2s, T=299.0, zs=zs, kijs=kijs)
>>> a_alpha, da_alpha_dT, d2a_alpha_dT2
(0.58562139582, -0.001018667672, 3.56669817856e-06)
>>> a_alpha_ijs
[[0.2491099357, 0.3686123937], [0.36861239374, 0.64864958635]]
>>> da_alpha_dT_ijs
[[-0.000510202800, -0.0006937567844], [-0.000693756784, -0.00111311535]]
>>> d2a_alpha_dT2_ijs
[[1.865112885e-06, 2.4734471244e-06], [2.4734471244e-06, 3.8843319e-06]]
```

Compute only the alpha term itself:

`thermo.eos_mix_methods.`**`a_alpha_and_derivatives`**(*a_alphas*, *T*, *zs*, *kijs*, *a_alpha_ijs=None*, *a_alpha_roots=None*, *a_alpha_ij_roots_inv=None*)

Faster implementations which do not store N^2 matrices:

thermo.eos_mix_methods.**a_alpha_quadratic_terms**(*a_alphas*, *a_alpha_roots*, *T*, *zs*, *kijs*, *a_alpha_j_rows=None*, *vec0=None*)

Calculates the *a_alpha* term for an equation of state along with the vector quantities needed to compute the fugacities of the mixture. This routine is efficient in both numba and PyPy.

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$(a\alpha)_{ij} = (1 - k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

The secondary values are as follows:

$$\sum_i y_i (a\alpha)_{ij}$$

Parameters

a_alphas [list[float]] EOS attractive terms, [J^2/mol^2/Pa]

a_alpha_roots [list[float]] Square roots of *a_alphas*; provided for speed [J/mol/Pa^0.5]

T [float] Temperature, not used, [K]

zs [list[float]] Mole fractions of each species

kijs [list[list[float]]] Constant kijs, [-]

a_alpha_j_rows [list[float], optional] EOS attractive term row destimation vector (does not need to be zeroed, should be provided to prevent allocations), [J^2/mol^2/Pa]

vec0 [list[float], optional] Empty vector, used in internal calculations, provide to avoid the allocations; does not need to be zeroed, [-]

Returns

a_alpha [float] EOS attractive term, [J^2/mol^2/Pa]

a_alpha_j_rows [list[float]] EOS attractive term row sums, [J^2/mol^2/Pa]

Notes

Tried moving the i=j loop out, no difference in speed, maybe got a bit slower in PyPy.

Examples

```
>>> kijs = [[0,.083],[0.083,0]]
>>> zs = [0.1164203, 0.8835797]
>>> a_alphas = [0.2491099357671155, 0.6486495863528039]
>>> a_alpha_roots = [i**0.5 for i in a_alphas]
>>> a_alpha, a_alpha_j_rows = a_alpha_quadratic_terms(a_alphas, a_alpha_roots, 299.
↪0, zs, kijs)
>>> a_alpha, a_alpha_j_rows
(0.58562139582, [0.35469988173, 0.61604757237])
```

thermo.eos_mix_methods.**a_alpha_and_derivatives_quadratic_terms**(*a_alphas*, *a_alpha_roots*, *da_alpha_dTs*, *d2a_alpha_dT2s*, *T*, *zs*, *kijs*, *a_alpha_j_rows=None*, *da_alpha_dT_j_rows=None*)

Calculates the *a_alpha* term, and its first two temperature derivatives, for an equation of state along with the vector quantities needed to compute the fugacitie and temperature derivatives of fugacities of the mixture. This routine is efficient in both numba and PyPy.

$$a\alpha = \sum_i \sum_j z_i z_j (a\alpha)_{ij}$$

$$\frac{\partial (a\alpha)}{\partial T} = \sum_i \sum_j z_i z_j \frac{\partial (a\alpha)_{ij}}{\partial T}$$

$$\frac{\partial^2 (a\alpha)}{\partial T^2} = \sum_i \sum_j z_i z_j \frac{\partial^2 (a\alpha)_{ij}}{\partial T^2}$$

$$(a\alpha)_{ij} = (1-k_{ij})\sqrt{(a\alpha)_i (a\alpha)_j}$$

$$\frac{\partial (a\alpha)_{ij}}{\partial T} = \frac{\sqrt{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} \left(1 - k_{ij}\right) \left(\frac{\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)}}{2} + \frac{\operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}}{2}\right)}{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}}$$

$$\frac{\partial^2 (a\alpha)_{ij}}{\partial T^2} = - \frac{\sqrt{\operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} \left(k_{ij} - 1\right) \left(\frac{\left(\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)} + \operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}\right)^{2}}{4 \operatorname{a\alpha_{i}}{\left(T \right)} \operatorname{a\alpha_{j}}{\left(T \right)}} - \frac{\left(\operatorname{a\alpha_{i}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)} + \operatorname{a\alpha_{j}}{\left(T \right)} \frac{d}{d T} \operatorname{a\alpha_{i}}{\left(T \right)}\right) \frac{d}{d T} \operatorname{a\alpha_{j}}{\left(T \right)}}{2 \operatorname{a\alpha_{j}}{\left(T \right)}} \cdots\right)}{\operatorname{a\alpha_{i}}{\left(T \cdots\right)}}$$

The secondary values are as follows:

$$\sum_i y_i(a\alpha)_{ij}$$

$$\sum_i y_i \frac{\partial (a\alpha)_{ij}}{\partial T}$$

Parameters

a_alphas [list[float]] EOS attractive terms, [J^2/mol^2/Pa]

a_alpha_roots [list[float]] Square roots of *a_alphas*; provided for speed [J/mol/Pa^0.5]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

T [float] Temperature, not used, [K]

zs [list[float]] Mole fractions of each species

kijs [list[list[float]]] Constant kijs, [-]

Returns

a_alpha [float] EOS attractive term, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alpha_j_rows [list[float]] EOS attractive term row sums, [J^2/mol^2/Pa]

da_alpha_dT_j_rows [list[float]] Temperature derivative of EOS attractive term row sums, [J^2/mol^2/Pa/K]

Examples

```
>>> kijs = [[0,.083],[0.083,0]]
>>> zs = [0.1164203, 0.8835797]
>>> a_alphas = [0.2491099357671155, 0.6486495863528039]
>>> a_alpha_roots = [i**0.5 for i in a_alphas]
>>> da_alpha_dTs = [-0.0005102028006086241, -0.0011131153520304886]
>>> d2a_alpha_dT2s = [1.8651128859234162e-06, 3.884331923127011e-06]
>>> a_alpha_and_derivatives_quadratic_terms(a_alphas, a_alpha_roots, da_alpha_dTs,␣
↪d2a_alpha_dT2s, 299.0, zs, kijs)
(0.58562139582, -0.001018667672, 3.56669817856e-06, [0.35469988173, 0.61604757237],␣
↪[-0.000672387374, -0.001064293501])
```

7.10 Cubic Equations of State Volume Solvers (thermo.eos_volume)

Some of the methods implemented here are numerical while others are analytical.

The cubic EOS can be rearranged into the following polynomial form:

$$0 = Z^3 + (\delta' - B' - 1)Z^2 + [\theta' + \epsilon' - \delta(B'+1)]Z - [\epsilon'(B'+1) + \theta'\eta']$$

$$B' = \frac{bP}{RT}$$

$$\delta' = \frac{\delta P}{RT}$$

$$\theta' = \frac{a\alpha P}{(RT)^2}$$

$$\epsilon' = \epsilon\left(\frac{P}{RT}\right)^2$$

The range of pressures, temperatures, and $a\alpha$ values is so large that almost all analytical solutions produce huge errors in some conditions. Because the EOS volume cannot be under b, this often results in a root being ignored where there should have been a liquid-like root detected.

A number of plots showing the relative error in volume calculation are shown below to demonstrate how different methods work.

7.10.1 Analytical Solvers

thermo.eos_volume.**volume_solutions_Cardano**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Calculate the molar volume solutions to a cubic equation of state using Cardano's formula, and a few tweaks to improve numerical precision. This solution is quite fast in general although it involves powers or trigonometric functions. However, it has numerical issues at many seemingly random areas in the low pressure region.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

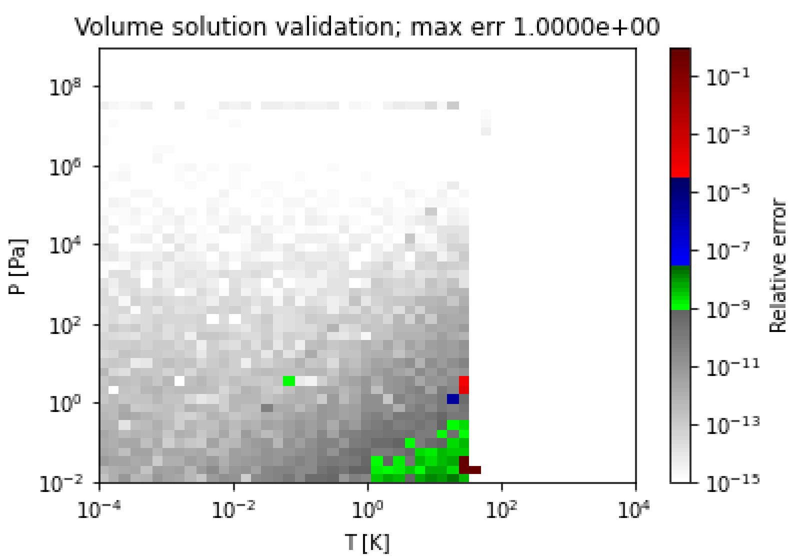
a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

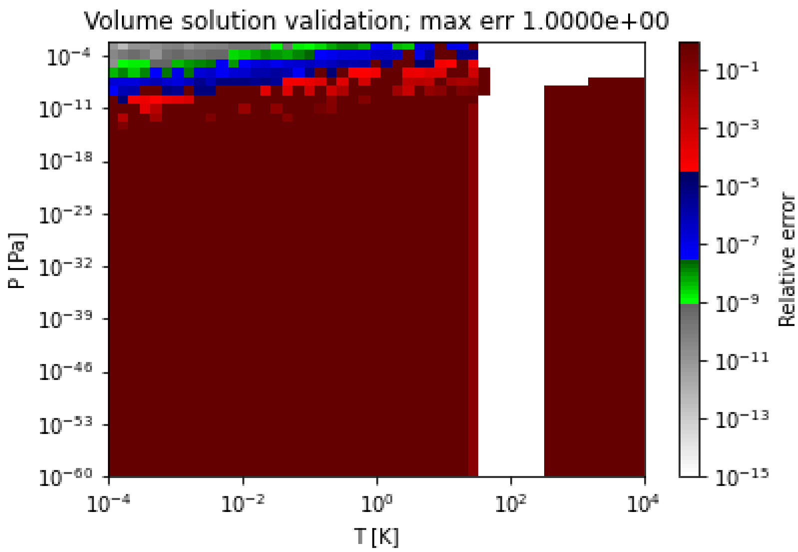
Returns

Vs [list[float]] Three possible molar volumes, [m^3/mol]

Notes

Two sample regions where this method does not obtain the correct solution (PR EOS for hydrogen) are as follows:





References

[1]

thermo.eos_volume.**volume_solutions_fast**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Solution of this form of the cubic EOS in terms of volumes. Returns three values, all with some complex part. This is believed to be the fastest analytical formula, and while it does not suffer from the same errors as Cardano's formula, it has plenty of its own numerical issues.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

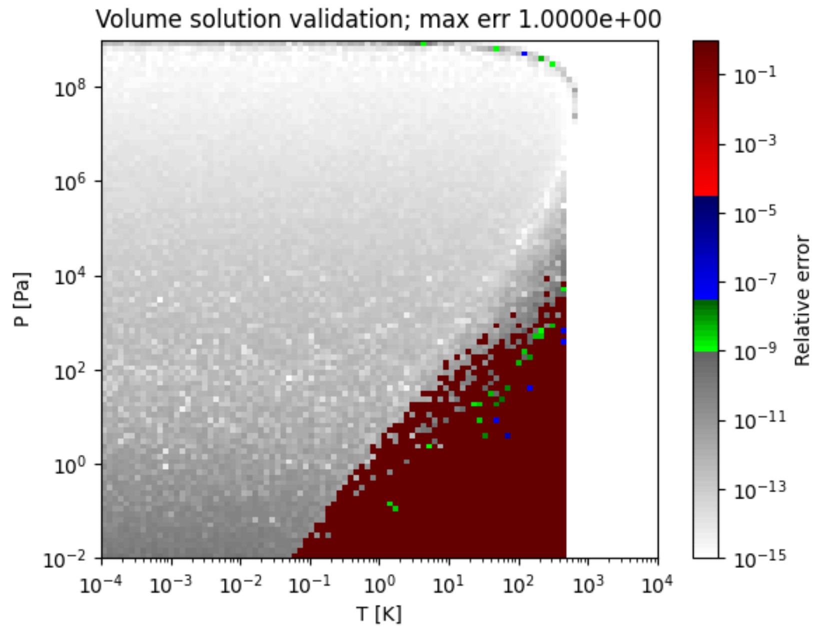
Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Notes

Using explicit formulas, as can be derived in the following example, is faster than most numeric root finding techniques, and finds all values explicitly. It takes several seconds.

```
>>> from sympy import *
>>> P, T, V, R, b, a, delta, epsilon, alpha = symbols('P, T, V, R, b, a, delta,␣
→epsilon, alpha')
>>> Tc, Pc, omega = symbols('Tc, Pc, omega')
>>> CUBIC = R*T/(V-b) - a*alpha/(V*V + delta*V + epsilon) - P
>>> #solve(CUBIC, V)
```

A sample region where this method does not obtain the correct solution (PR EOS for methanol) is as follows:



References

[1]

thermo.eos_volume.**volume_solutions_a1**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Solution of this form of the cubic EOS in terms of volumes. Returns three values, all with some complex part. This uses an analytical solution for the cubic equation with the leading coefficient set to 1 as in the EOS case; and the analytical solution is the one recommended by Mathematica.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

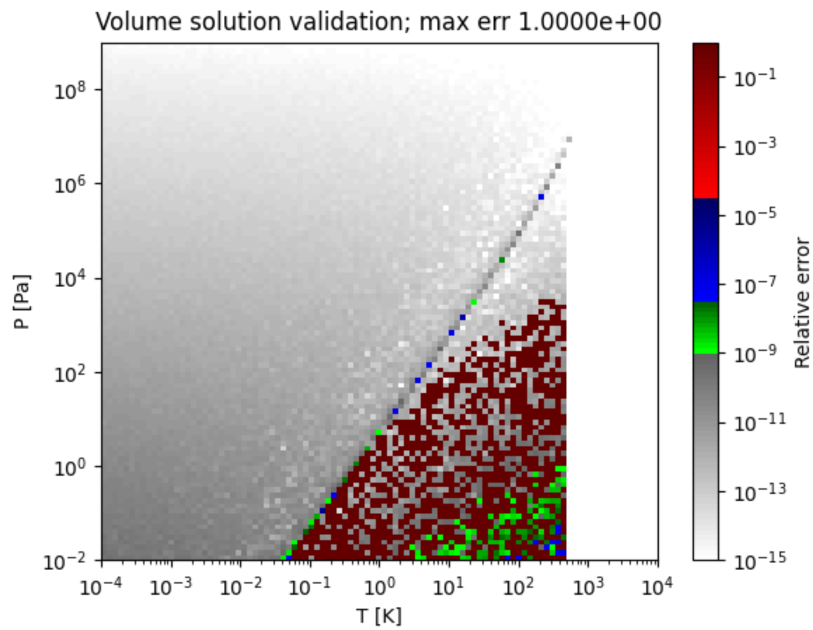
a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Notes

A sample region where this method does not obtain the correct solution (PR EOS for methanol) is as follows:



Examples

Numerical precision is always challenging and has edge cases. The following results all havev imaginary components, but depending on the math library used by the compiler even the first complex digit may not match!

```
>>> volume_solutions_a1(8837.07874361444, 216556124.0631852, 0.0003990176625589891,
↪0.0010590390565805598, -1.5069972655436541e-07, 7.20417995032918e-15)
((0.000738308-7.5337e-20j), (-0.001186094-6.52444e-20j), (0.000127055+6.52444e-20j))
```

thermo.eos_volume.**volume_solutions_a2**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Solution of this form of the cubic EOS in terms of volumes. Returns three values, all with some complex part. This uses an analytical solution for the cubic equation with the leading coefficient set to 1 as in the EOS case; and the analytical solution is the one recommended by Maple.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

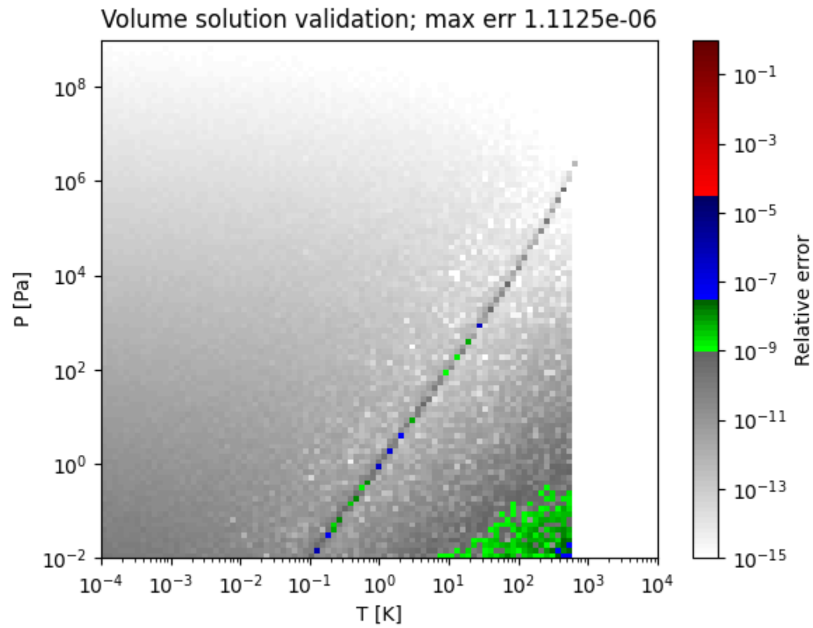
a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Notes

A sample region where this method does not obtain the correct solution (SRK EOS for decane) is as follows:



thermo.eos_volume.**volume_solutions_numpy**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Calculate the molar volume solutions to a cubic equation of state using NumPy's *roots* function, which is a power series iterative matrix solution that is very stable but does not have full precision in some cases.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

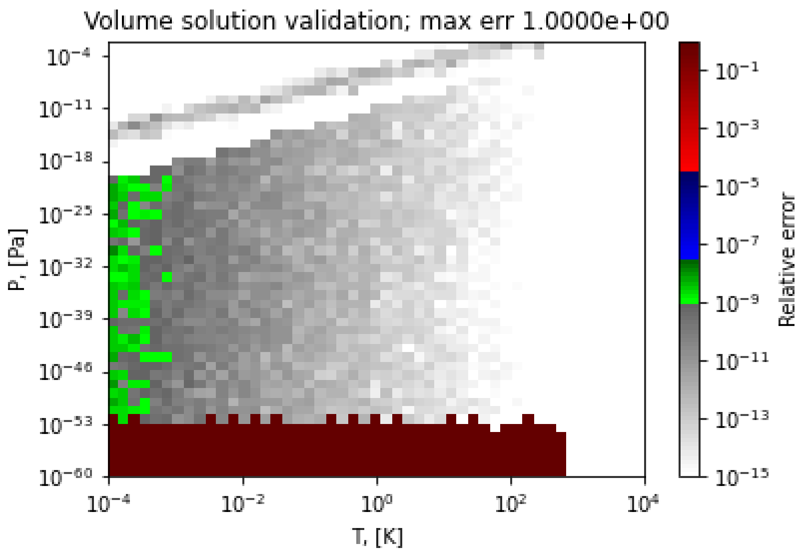
a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [list[float]] Three possible molar volumes, [m^3/mol]

Notes

A sample region where this method does not obtain the correct solution (SRK EOS for ethane) is as follows:



References

[1]

thermo.eos_volume.**volume_solutions_ideal**(*T*, *P*, *b=0.0*, *delta=0.0*, *epsilon=0.0*, *a_alpha=0.0*)

Calculate the ideal-gas molar volume in a format compatible with the other cubic EOS solvers. The ideal gas volume is the first element; and the secodn and third elements are zero. This is implemented to allow the ideal-gas model to be compatible with the cubic models, whose equations do not work with parameters of zero.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float, optional] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float, optional] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float, optional] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float, optional] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [list[float]] Three possible molar volumes, [m^3/mol]

Examples

```
>>> volume_solutions_ideal(T=300, P=1e7)
(0.0002494338785445972, 0.0, 0.0)
```

7.10.2 Numerical Solvers

thermo.eos_volume.**volume_solutions_halley**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Halley's method based solver for cubic EOS volumes based on the idea of initializing from a single liquid-like guess which is solved precisely, deflating the cubic analytically, solving the quadratic equation for the next two volumes, and then performing two halley steps on each of them to obtain the final solutions. This method does not calculate imaginary roots - they are set to zero on detection. This method has been rigorously tested over a wide range of conditions.

The method uses the standard combination of bisection to provide high and low boundaries as well, to keep the iteration always moving forward.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [tuple[float]] Three possible molar volumes, [m^3/mol]

Notes

A sample region where this method works perfectly is shown below:

thermo.eos_volume.**volume_solutions_NR**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*, *tries=0*)

Newton-Raphson based solver for cubic EOS volumes based on the idea of initializing from an analytical solver. This algorithm can only be described as a monstrous mess. It is fairly fast for most cases, but about 3x slower than `volume_solutions_halley`. In the worst case this will fall back to *mpmath*.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

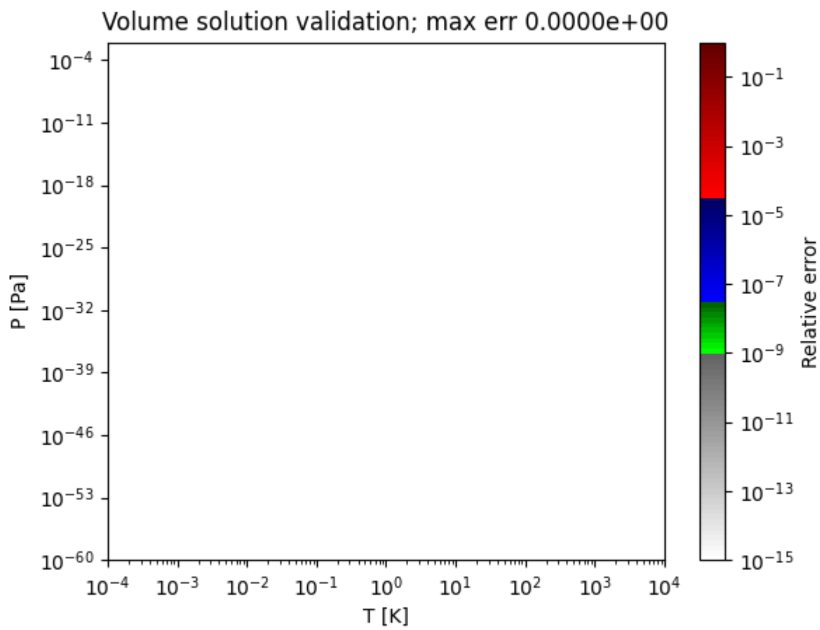
b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

tries [int, optional] Internal parameter as this function will call itself if it needs to; number of previous solve attempts, [-]



Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Notes

Sample regions where this method works perfectly are shown below:

`thermo.eos_volume.`**`volume_solutions_NR_low_P`**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Newton-Raphson based solver for cubic EOS volumes designed specifically for the low-pressure regime. Seeks only two possible solutions - an ideal gas like one, and one near the eos covolume *b* - as the initializations are *R*T/P* and *b*1.000001* .

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

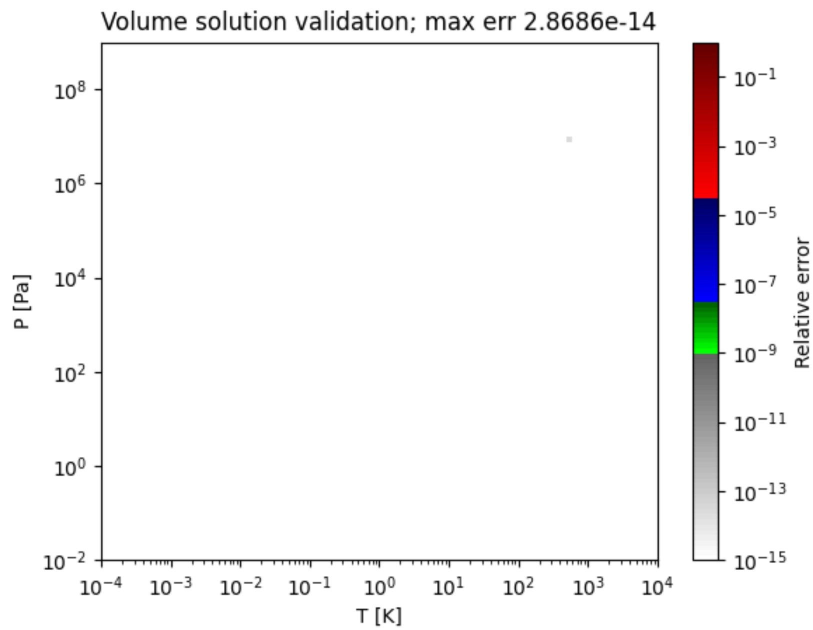
epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

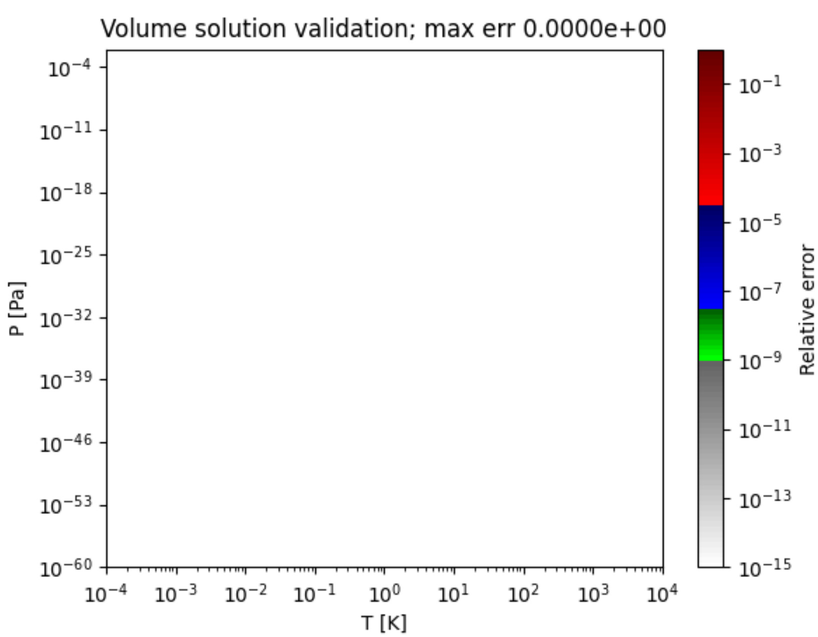
tries [int, optional] Internal parameter as this function will call itself if it needs to; number of previous solve attempts, [-]

Returns

Vs [tuple[complex]] Three possible molar volumes (third one is hardcoded to 1j), [m^3/mol]



Volume solution validation; max err 2.8686e-14
P [Pa]
T [K]
Relative error



Volume solution validation; max err 0.0000e+00
P [Pa]
T [K]
Relative error

Notes

The algorithm is NR, with some checks that will switch the solver to *brenth* some of the time.

7.10.3 Higher-Precision Solvers

`thermo.eos_volume.volume_solutions_mpmath`(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*, *dps=50*)

Solution of this form of the cubic EOS in terms of volumes, using the *mpmath* arbitrary precision library. The number of decimal places returned is controlled by the *dps* parameter.

This function is the reference implementation which provides exactly correct solutions; other algorithms are compared against this one.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

dps [int] Number of decimal places in the result by *mpmath*, [-]

Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Notes

Although *mpmath* has a cubic solver, it has been found to fail to solve in some cases. Accordingly, the algorithm is as follows:

Working precision is *dps* plus 40 digits; and if P < 1e-10 Pa, it is *dps* plus 400 digits. The input parameters are converted exactly to *mpf* objects on input.

polyroots from mpmath is used with *maxsteps=2000*, and extra precision of 15 digits. If the solution does not converge, 20 extra digits are added up to 8 times. If no solution is found, mpmath's *findroot* is called on the pressure error function using three initial guesses from another solver.

Needless to say, this function is quite slow.

References

[1]

Examples

Test case which presented issues for PR EOS (three roots were not being returned):

```
>>> volume_solutions_mpmath(0.01, 1e-05, 2.5405184201558786e-05, 5.081036840311757e-
↪05, -6.454233843151321e-10, 0.3872747173781095)
(mpf('0.0000254054613415548712260258773060137'), mpf('4.
↪66038025602155259976574392093252'), mpf('8309.80218708657190094424659859346'))
```

thermo.eos_volume.**volume_solutions_mpmath_float**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Simple wrapper around *volume_solutions_mpmath* which uses the default parameters and returns the values as floats.

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

dps [int] Number of decimal places in the result by *mpmath*, [-]

Returns

Vs [tuple[complex]] Three possible molar volumes, [m^3/mol]

Examples

Test case which presented issues for PR EOS (three roots were not being returned):

```
>>> volume_solutions_mpmath_float(0.01, 1e-05, 2.5405184201558786e-05, 5.
↪081036840311757e-05, -6.454233843151321e-10, 0.3872747173781095)
((2.540546134155487e-05+0j), (4.660380256021552+0j), (8309.802187086572+0j))
```

thermo.eos_volume.**volume_solutions_sympy**(*T*, *P*, *b*, *delta*, *epsilon*, *a_alpha*)

Solution of this form of the cubic EOS in terms of volumes, using the *sympy* mathematical library with real numbers.

This function is generally slow, and somehow still has more than desired error in the real and complex result.

$$V_0 = -\frac{-\frac{3(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P} + \frac{(-Pb+P\delta-RT)^2}{P^2}}{3\sqrt[3]{\frac{\sqrt{-4\left(-\frac{3(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P}+\frac{(-Pb+P\delta-RT)^2}{P^2}\right)^3+\left(\frac{27(-Pb\epsilon-RT\epsilon-a\alpha b)}{P}-\frac{9(-Pb+P\delta-RT)(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P^2}+\frac{2(-Pb+P\delta-RT)^3}{P^3}\right)^2}}{2} + \cdots}}$$

$$V_1 = -\frac{-\frac{3(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P} + \frac{(-Pb+P\delta- \cdots}{P^2}}{3\left(-\frac{1}{2}-\frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{\sqrt{-4\left(-\frac{3(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P}+\frac{(-Pb+P\delta-RT)^2}{P^2}\right)^3+\left(\frac{27(-Pb\epsilon-RT\epsilon-a\alpha b)}{P}-\frac{9(-Pb+P\delta-RT)(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P^2}+\frac{2(-Pb \cdots}{}\right.}}{2}}}$$

$$V_2 = -\frac{-\frac{3(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P} + \frac{(-Pb+P\delta-\cdots}{P^2}\cdots}{3\left(-\frac{1}{2}+\frac{\sqrt{3}i}{2}\right)\sqrt[3]{\frac{\sqrt{-4\left(-\frac{3(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P}+\frac{(-Pb+P\delta-RT)^2}{P^2}\right)^3+\left(\frac{27(-Pb\epsilon-RT\epsilon-a\alpha b)}{P}-\frac{9(-Pb+P\delta-RT)(-Pb\delta+P\epsilon-RT\delta+a\alpha)}{P^2}+\frac{2(-Pb-\cdots}{\cdots}\cdots}}{2}\cdots}}$$

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

b [float] Coefficient calculated by EOS-specific method, [m^3/mol]

delta [float] Coefficient calculated by EOS-specific method, [m^3/mol]

epsilon [float] Coefficient calculated by EOS-specific method, [m^6/mol^2]

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Returns

Vs [tuple[sympy.Rational]] Three possible molar volumes, [m^3/mol]

Notes

The solution can be derived as follows:

```
>>> from sympy import *
>>> P, T, V, R, b, delta, epsilon = symbols('P, T, V, R, b, delta, epsilon')
>>> a_alpha = Symbol(r'a \alpha')
>>> CUBIC = R*T/(V-b) - a_alpha/(V*V + delta*V + epsilon) - P
>>> V_slns = solve(CUBIC, V)
```

References

[1]

Examples

```
>>> Vs = volume_solutions_sympy(0.01, 1e-05, 2.5405184201558786e-05, 5.
↪081036840311757e-05, -6.454233843151321e-10, 0.3872747173781095)
>>> [complex(v) for v in Vs]
[(2.540546e-05+2.402202278e-12j), (4.660380256-2.40354958e-12j), (8309.80218+1.
↪348096981e-15j)]
```

7.11 Cubic Equation of State Alpha Functions (thermo.eos_alpha_functions)

This module contains implementations of the calculation of pure-component EOS $a\alpha$ parameters in a vectorized way. Functions for calculating their temperature derivatives as may be necessary are included as well.

For certain alpha functions, a class is available to provide these functions to and class that inherits from it.

A mixing rule must be used on the *a_alphas* to get the overall *a_alpha* term.

7.11.1 Vectorized Alpha Functions

thermo.eos_alpha_functions.**PR_a_alphas_vectorized**(*T*, *Tcs*, *ais*, *kappas*, *a_alphas=None*)

Calculates the *a_alpha* terms for the Peng-Robinson equation of state given the critical temperatures *Tcs*, constants *ais*, and *kappas*.

$$a_i\alpha(T)_i = a_i[1 + \kappa_i(1 - \sqrt{T_{r,i}})]^2$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$, [Pa*m^6/mol^2]

kappas [list[float]] *kappa* parameters of Peng-Robinson EOS; formulas vary, but the original form uses $\kappa_i = 0.37464 + 1.54226\omega_i - 0.26992\omega_i^2$, [-]

a_alphas [list[float], optional] Vector for pure component *a_alpha* terms in the cubic EOS to be calculated and stored in, [Pa*m^6/mol^2]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

Examples

```
>>> Tcs = [469.7, 507.4, 540.3]
>>> ais = [2.0698956357716662, 2.7018068455659545, 3.3725793885832323]
>>> kappas = [0.74192743008, 0.819919992, 0.8800122140799999]
>>> PR_a_alphas_vectorized(322.29, Tcs=Tcs, ais=ais, kappas=kappas)
[2.6306811679, 3.6761503348, 4.8593286234]
```

`thermo.eos_alpha_functions.`**`SRK_a_alphas_vectorized`**(*T*, *Tcs*, *ais*, *ms*, *a_alphas=None*)

Calculates the *a_alpha* terms for the SRK equation of state given the critical temperatures *Tcs*, constants *ais*, and *kappas*.

$$a_i\alpha(T)_i = \left[1 + m_i\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\right]^2$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$, [Pa*m^6/mol^2]

ms [list[float]] *m* parameters of SRK EOS; formulas vary, but the original form uses $m_i = 0.480 + 1.574\omega_i - 0.176\omega_i^2$, [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

Examples

```
>>> Tcs = [469.7, 507.4, 540.3]
>>> ais = [1.9351940385541342, 2.525982668162287, 3.1531036708059315]
>>> ms = [0.8610138239999999, 0.9436976, 1.007889024]
>>> SRK_a_alphas_vectorized(322.29, Tcs=Tcs, ais=ais, ms=ms)
[2.549485814512, 3.586598245260, 4.76614806648]
```

`thermo.eos_alpha_functions.`**`PRSV_a_alphas_vectorized`**(*T*, *Tcs*, *ais*, *kappa0s*, *kappa1s*, *a_alphas=None*)

Calculates the *a_alpha* terms for the Peng-Robinson-Stryjek-Vera equation of state given the critical temperatures *Tcs*, constants *ais*, PRSV parameters *kappa0s* and *kappa1s*.

$$a_i\alpha_i = a_i\left(\left(\kappa_0 + \kappa_1\left(\sqrt{\frac{T}{T_{c,i}}} + 1\right)\left(-\frac{T}{T_{c,i}} + \frac{7}{10}\right)\right)\left(-\sqrt{\frac{T}{T_{c,i}}} + 1\right) + 1\right)^2$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$, [Pa*m^6/mol^2]

kappa0s [list[float]] *kappa0* parameters of PRSV EOS; the original form uses $\kappa_{0,i} = 0.378893 + 1.4897153\omega_i - 0.17131848\omega_i^2 + 0.0196554\omega_i^3$, [-]

kappa1s [list[float]] Fit parameters, can be set to 0 if unknown [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

Examples

```
>>> Tcs = [507.6]
>>> ais = [2.6923169620277805]
>>> kappa0s = [0.8074380841890093]
>>> kappa1s = [0.05104]
>>> PRSV_a_alphas_vectorized(299.0, Tcs=Tcs, ais=ais, kappa0s=kappa0s,␣
↪kappa1s=kappa1s)
[3.81298569831]
```

thermo.eos_alpha_functions.**PRSV2_a_alphas_vectorized**(*T*, *Tcs*, *ais*, *kappa0s*, *kappa1s*, *kappa2s*, *kappa3s*, *a_alphas=None*)

Calculates the *a_alpha* terms for the Peng-Robinson-Stryjek-Vera 2 equation of state given the critical temperatures *Tcs*, constants *ais*, PRSV2 parameters *kappa0s, `kappa1s*, *kappa2s*, and *kappa3s*.

$$a_i\alpha_i = a_i\left(\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\left(\kappa_{0,i} + \left(\kappa_{1,i} + \kappa_{2,i}\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\left(-\frac{T}{T_{c,i}} + \kappa_{3,i}\right)\right)\left(\sqrt{\frac{T}{T_{c,i}}} + 1\right)\left(-\frac{T}{T_{c,i}} + \frac{7}{10}\right)\right) + 1\right)^2$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$, [Pa*m^6/mol^2]

kappa0s [list[float]] *kappa0* parameters of PRSV EOS; the original form uses $\kappa_{0,i} = 0.378893 + 1.4897153\omega_i - 0.17131848\omega_i^2 + 0.0196554\omega_i^3$, [-]

kappa1s [list[float]] Fit parameters, can be set to 0 if unknown [-]

kappa2s [list[float]] Fit parameters, can be set to 0 if unknown [-]

kappa3s [list[float]] Fit parameters, can be set to 0 if unknown [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

Examples

```
>>> PRSV2_a_alphas_vectorized(400.0, Tcs=[507.6], ais=[2.6923169620277805],␣
↪kappa0s=[0.8074380841890093], kappa1s=[0.05104], kappa2s=[0.8634], kappa3s=[0.
↪460])
[3.2005700986984]
```

thermo.eos_alpha_functions.**APISRK_a_alphas_vectorized**(*T*, *Tcs*, *ais*, *S1s*, *S2s*, *a_alphas=None*)

Calculates the *a_alpha* terms for the API SRK equation of state given the critical temperatures *Tcs*, constants *ais*, and API parameters *S1s* and *S2s*.

$$a_i\alpha(T)_i = a_i\left[1 + S_{1,i}\left(1 - \sqrt{T_{r,i}}\right) + S_{2,i}\frac{1 - \sqrt{T_{r,i}}}{\sqrt{T_{r,i}}}\right]^2$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = \frac{0.42748 \cdot R^2 (T_{c,i})^2}{P_{c,i}}$, [Pa*m^6/mol^2]

S1s [list[float]] *S1* parameters of API SRK EOS; regressed or estimated with $S_{1,i} = 0.48508 + 1.55171\omega_i - 0.15613\omega_i^2$, [-]

S2s [list[float]] *S2* parameters of API SRK EOS; regressed or set to zero, [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

Examples

```
>>> APISRK_a_alphas_vectorized(T=430.0, Tcs=[514.0], ais=[1.2721974560809934], ␣
↪S1s=[1.678665], S2s=[-0.216396])
[1.60465652994097]
```

thermo.eos_alpha_functions.**RK_a_alphas_vectorized**(*T*, *Tcs*, *ais*, *a_alphas=None*)

Calculates the *a_alpha* terms for the RK equation of state given the critical temperatures *Tcs*, and *a* parameters *ais*.

$$a_i \alpha_i = \frac{a_i}{\sqrt{\frac{T}{T_{c,i}}}}$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = \frac{0.42748 \cdot R^2 (T_{c,i})^2}{P_{c,i}}$, [Pa*m^6/mol^2]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

Examples

```
>>> Tcs = [469.7, 507.4, 540.3]
>>> ais = [1.9351940385541342, 2.525982668162287, 3.1531036708059315]
>>> RK_a_alphas_vectorized(322.29, Tcs=Tcs, ais=ais)
[2.3362073307, 3.16943743055, 4.0825575798]
```

7.11.2 Vectorized Alpha Functions With Derivatives

thermo.eos_alpha_functions.**PR_a_alpha_and_derivatives_vectorized**(*T*, *Tcs*, *ais*, *kappas*, *a_alphas=None*, *da_alpha_dTs=None*, *d2a_alpha_dT2s=None*)

Calculates the *a_alpha* terms and their first two temperature derivatives for the Peng-Robinson equation of state given the critical temperatures *Tcs*, constants *ais*, and *kappas*.

$$a_i\alpha(T)_i = a_i[1 + \kappa_i(1 - \sqrt{T_{r,i}})]^2$$

$$\frac{da_i\alpha_i}{dT} = -\frac{a_i\kappa_i}{T^{0.5}T_{ci}^{0.5}}\left(\kappa_i\left(-\frac{T^{0.5}}{T_{ci}^{0.5}} + 1\right) + 1\right)$$

$$\frac{d^2a_i\alpha_i}{dT^2} = 0.5a_i\kappa_i\left(-\frac{1}{T^{1.5}T_{ci}^{0.5}}\left(\kappa_i\left(\frac{T^{0.5}}{T_{ci}^{0.5}}-1\right)-1\right)+\frac{\kappa_i}{TT_{ci}}\right)$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$ [Pa*m^6/mol^2]

kappas [list[float]] *kappa* parameters of Peng-Robinson EOS; formulas vary, but the original form uses $\kappa_i = 0.37464 + 1.54226\omega_i - 0.26992\omega_i^2$, [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

da_alpha_dTs [list[float]] First temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K)]

d2a_alpha_dT2s [list[float]] Second temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K^2)]

Examples

```
>>> Tcs = [469.7, 507.4, 540.3]
>>> ais = [2.0698956357716662, 2.7018068455659545, 3.3725793885832323]
>>> kappas = [0.74192743008, 0.819919992, 0.8800122140799999]
>>> PR_a_alpha_and_derivatives_vectorized(322.29, Tcs=Tcs, ais=ais, kappas=kappas)
([2.63068116797, 3.67615033489, 4.859328623453], [-0.0044497546430, -0.
↪00638993749167, -0.0085372308846], [1.066668360e-05, 1.546687574587e-05, 2.
↪07440632117e-05])
```

`thermo.eos_alpha_functions.SRK_a_alpha_and_derivatives_vectorized`(*T*, *Tcs*, *ais*, *ms*, *a_alphas=None*, *da_alpha_dTs=None*, *d2a_alpha_dT2s=None*)

Calculates the *a_alpha* terms and their first and second temperature derivatives for the SRK equation of state given the critical temperatures *Tcs*, constants *ais*, and *kappas*.

$$a_i\alpha(T)_i = \left[1 + m_i\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\right]^2$$

$$\frac{da_i\alpha_i}{dT} = \frac{a_im_i}{T}\sqrt{\frac{T}{T_{c,i}}}\left(m_i\left(\sqrt{\frac{T}{Tc,i}}-1\right)-1\right)$$

$$\frac{d^2a_i\alpha_i}{dT^2} = \frac{a_im_i\sqrt{\frac{T}{T_{c,i}}}}{2T^2}\left(m_i+1\right)$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = \frac{0.42748\cdot R^2(T_{c,i})^2}{P_{c,i}}$, [Pa*m^6/mol^2]

ms [list[float]] m parameters of SRK EOS; formulas vary, but the original form uses $m_i = 0.480 + 1.574\omega_i - 0.176\omega_i^2$, [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

da_alpha_dTs [list[float]] First temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K)]

d2a_alpha_dT2s [list[float]] Second temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K^2)]

Examples

```
>>> Tcs = [469.7, 507.4, 540.3]
>>> ais = [1.9351940385541342, 2.525982668162287, 3.1531036708059315]
>>> ms = [0.8610138239999999, 0.9436976, 1.007889024]
>>> SRK_a_alpha_and_derivatives_vectorized(322.29, Tcs=Tcs, ais=ais, ms=ms)
([2.549485814512, 3.586598245260, 4.76614806648], [-0.004915469296196, -0.
↪00702410108423, -0.00936320876945], [1.236441916324e-05, 1.77752796719e-05, 2.
↪37231823137e-05])
```

thermo.eos_alpha_functions.**PRSV_a_alpha_and_derivatives_vectorized**(*T, Tcs, ais, kappa0s, kappa1s, a_alphas=None, da_alpha_dTs=None, d2a_alpha_dT2s=None*)

Calculates the *a_alpha* terms and their first and second derivative for the Peng-Robinson-Stryjek-Vera equation of state given the critical temperatures *Tcs*, constants *ais*, PRSV parameters *kappa0s* and *kappa1s*.

$$a_i\alpha_i = a_i\left(\left(\kappa_0 + \kappa_1\left(\sqrt{\frac{T}{T_{c,i}}} + 1\right)\left(-\frac{T}{T_{c,i}} + \frac{7}{10}\right)\right)\left(-\sqrt{\frac{T}{T_{c,i}}} + 1\right) + 1\right)^2$$

$$\frac{da_i\alpha_i}{dT} = a_i\left(\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\left(\kappa_{0,i} + \kappa_{1,i}\left(\sqrt{\frac{T}{T_{c,i}}} + 1\right)\left(-\frac{T}{T_{c,i}} + \frac{7}{10}\right)\right) + 1\right)\left(2\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\left(-\frac{\kappa_{1,i}\left(\sqrt{\frac{T}{T_{c,i}}} + 1\right)}{T_{c,i}} + \right.\right.$$

$$\frac{d^2a_i\alpha_i}{dT^2} = \frac{a_i\left(\left(\kappa_{1,i}\left(\sqrt{\frac{T}{T_{c,i}}} - 1\right)\left(\frac{20\left(\sqrt{\frac{T}{T_{c,i}}}+1\right)}{T_{c,i}} + \frac{\sqrt{\frac{T}{T_{c,i}}}\left(\frac{10T}{T_{c,i}}-7\right)}{T}\right) - \frac{\sqrt{\frac{T}{T_{c,i}}}\left(10\kappa_{0,i}-\kappa_{1,i}\left(\sqrt{\frac{T}{T_{c,i}}}+1\right)\left(\frac{10T}{T_{c,i}}-7\right)\right)}{T}\right)^2 - \frac{\sqrt{\frac{T}{T_{c,i}}}\left(\right.}{}\right.}{}$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] a parameters of cubic EOS, $a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$, [Pa*m^6/mol^2]

kappa0s [list[float]] *kappa0* parameters of PRSV EOS; the original form uses $\kappa_{0,i} = 0.378893 + 1.4897153\omega_i - 0.17131848\omega_i^2 + 0.0196554\omega_i^3$, [-]

kappa1s [list[float]] Fit parameters, can be set to 0 if unknown [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

da_alpha_dTs [list[float]] First temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K)]

d2a_alpha_dT2s [list[float]] Second temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K^2)]

Examples

```
>>> Tcs = [507.6]
>>> ais = [2.6923169620277805]
>>> kappa0s = [0.8074380841890093]
>>> kappa1s = [0.05104]
>>> PRSV_a_alpha_and_derivatives_vectorized(299.0, Tcs=Tcs, ais=ais,␣
↪kappa0s=kappa0s, kappa1s=kappa1s)
([3.8129856983], [-0.0069769034748], [2.00265608110e-05])
```

`thermo.eos_alpha_functions.`**`PRSV2_a_alpha_and_derivatives_vectorized`**(*T, Tcs, ais, kappa0s, kappa1s, kappa2s, kappa3s, a_alphas=None, da_alpha_dTs=None, d2a_alpha_dT2s=None*)

Calculates the *a_alpha* terms and their first and second derivatives for the Peng-Robinson-Stryjek-Vera 2 equation of state given the critical temperatures *Tcs*, constants *ais*, PRSV2 parameters *kappa0s, `kappa1s*, *kappa2s*, and *kappa3s*.

$$a_i\alpha_i = a_i\left(\left(1-\sqrt{\frac{T}{T_{c,i}}}\right)\left(\kappa_{0,i}+\left(\kappa_{1,i}+\kappa_{2,i}\left(1-\sqrt{\frac{T}{T_{c,i}}}\right)\left(-\frac{T}{T_{c,i}}+\kappa_{3,i}\right)\right)\left(\sqrt{\frac{T}{T_{c,i}}}+1\right)\left(-\frac{T}{T_{c,i}}+\frac{7}{10}\right)\right)+1\right)^2$$

$$\frac{da_i\alpha_i}{dT} = a_i\left(\left(1-\sqrt{\frac{T}{T_{c,i}}}\right)\left(\kappa_{0,i}+\left(\kappa_{1,i}+\kappa_{2,i}\left(1-\sqrt{\frac{T}{T_{c,i}}}\right)\left(-\frac{T}{T_{c,i}}+\kappa_{3,i}\right)\right)\left(\sqrt{\frac{T}{T_{c,i}}}+1\right)\left(-\frac{T}{T_{c,i}}+\frac{7}{10}\right)\right)+1\right)\left(\cdots\right.$$

$$\frac{d^2a_i\alpha_i}{dT^2} = -\frac{a_i\left(\left(\left(10\kappa_{0,i}-\left(\kappa_{1,i}+\kappa_{2,i}\left(\sqrt{\frac{T}{T_{c,i}}}-1\right)\left(\frac{T}{T_{c,i}}-\kappa_{3,i}\right)\right)\left(\sqrt{\frac{T}{T_{c,i}}}+1\right)\left(\frac{10T}{T_{c,i}}-7\right)\right)\left(\sqrt{\frac{T}{T_{c,i}}}-1\right)-10\right)\left(\left(\sqrt{\cdots}\right.\right.\right.}{\cdots}$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = 0.45724\frac{R^2T_{c,i}^2}{P_{c,i}}$, [Pa*m^6/mol^2]

kappa0s [list[float]] *kappa0* parameters of PRSV EOS; the original form uses $\kappa_{0,i} = 0.378893 + 1.4897153\omega_i - 0.17131848\omega_i^2 + 0.0196554\omega_i^3$, [-]

kappa1s [list[float]] Fit parameters, can be set to 0 if unknown [-]

kappa2s [list[float]] Fit parameters, can be set to 0 if unknown [-]

kappa3s [list[float]] Fit parameters, can be set to 0 if unknown [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

da_alpha_dTs [list[float]] First temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K)]

d2a_alpha_dT2s [list[float]] Second temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K^2)]

Examples

```
>>> PRSV2_a_alpha_and_derivatives_vectorized(400.0, Tcs=[507.6], ais=[2.
↪6923169620277805], kappa0s=[0.8074380841890093], kappa1s=[0.05104], kappa2s=[0.
↪8634], kappa3s=[0.460])
([3.2005700986], [-0.005301195971], [1.11181477576e-05])
```

`thermo.eos_alpha_functions.`**`APISRK_a_alpha_and_derivatives_vectorized`**(*T, Tcs, ais, S1s, S2s, a_alphas=None, da_alpha_dTs=None, d2a_alpha_dT2s=None*)

Calculates the *a_alpha* terms and their first two temperature derivatives for the API SRK equation of state given the critical temperatures *Tcs*, constants *ais*, and API parameters *S1s* and *S2s*.

$$a_i\alpha(T)_i = a_i \left[1 + S_{1,i}\left(1-\sqrt{T_{r,i}}\right) + S_{2,i}\frac{1-\sqrt{T_{r,i}}}{\sqrt{T_{r,i}}}\right]^2$$

$$\frac{d a_i\alpha_i}{dT} = a_i\frac{T_{c,i}}{T^{2}} \left(- S_{2,i} \left(\sqrt{\frac{T}{T_{c,i}}} - 1\right) + \sqrt{\frac{T}{T_{c,i}}} \left(S_{1,i} \sqrt{\frac{T}{T_{c,i}}} + S_{2,i}\right)\right) \left(S_{2,i} \left(\sqrt{\frac{T}{T_{c,i}}} - 1\right) + \sqrt{\frac{T}{T_{c,i}}} \left(S_{1,i} \left(\sqrt{\frac{T}{T_{c,i}}} - 1\right) -$$

$$\frac{d^2 a_i\alpha_i}{dT^2} = a_i\frac{1}{2 T^{3}} \left(S_{1,i}^{2} T \sqrt{\frac{T}{T_{c,i}}} - S_{1,i} S_{2,i} T \sqrt{\frac{T}{T_{c,i}}} + 3 S_{1,i} S_{2,i} T_{c,i} \sqrt{\frac{T}{T_{c,i}}} + S_{1,i} T \sqrt{\frac{T}{T_{c,i}}} - 3 S_{2,i}^{2} T_{c,i} \sqrt{\frac{T}{T_{c,i}}} + 4 S_{2,i}^{2} T_{c,i} + 3 S_{2,}$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i=\frac{0.42748\cdot R^2(T_{c,i})^{2}}{P_{c,i}}$, [Pa*m^6/mol^2]

S1s [list[float]] *S1* parameters of API SRK EOS; regressed or estimated with $S_{1,i} = 0.48508 + 1.55171\omega_i - 0.15613\omega_i^2$, [-]

S2s [list[float]] *S2* parameters of API SRK EOS; regressed or set to zero, [-]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

da_alpha_dTs [list[float]] First temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K)]

d2a_alpha_dT2s [list[float]] Second temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K^2)]

Examples

```
>>> APISRK_a_alpha_and_derivatives_vectorized(T=430.0, Tcs=[514.0], ais=[1.
↪2721974560809934],  S1s=[1.678665], S2s=[-0.216396])
([1.60465652994], [-0.0043155855337], [8.9931026263e-06])
```

thermo.eos_alpha_functions.**RK_a_alpha_and_derivatives_vectorized**(*T*, *Tcs*, *ais*, *a_alphas=None*, *da_alpha_dTs=None*, *d2a_alpha_dT2s=None*)

Calculates the *a_alpha* terms and their first and second temperature derivatives for the RK equation of state given the critical temperatures *Tcs*, and *a* parameters *ais*.

$$a_i\alpha_i = \frac{a_i}{\sqrt{\frac{T}{T_{c,i}}}}$$

$$\frac{da_i\alpha_i}{dT} = -\frac{a_i}{2T\sqrt{\frac{T}{T_{c,i}}}}$$

$$\frac{d^2a_i\alpha_i}{dT^2} = \frac{3a_i}{4T^2\sqrt{\frac{T}{T_{c,i}}}}$$

Parameters

T [float] Temperature, [K]

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

ais [list[float]] *a* parameters of cubic EOS, $a_i = \frac{0.42748 \cdot R^2(T_{c,i})^2}{P_{c,i}}$, [Pa*m^6/mol^2]

Returns

a_alphas [list[float]] Pure component *a_alpha* terms in the cubic EOS, [Pa*m^6/mol^2]

da_alpha_dTs [list[float]] First temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K)]

d2a_alpha_dT2s [list[float]] Second temperature derivative of pure component *a_alpha*, [Pa*m^6/(mol^2*K^2)]

Examples

```
>>> Tcs = [469.7, 507.4, 540.3]
>>> ais = [1.9351940385541342, 2.525982668162287, 3.1531036708059315]
>>> RK_a_alpha_and_derivatives_vectorized(322.29, Tcs=Tcs, ais=ais)
([2.3362073307, 3.16943743055, 4.08255757984], [-0.00362438693525, -0.0049170582868,
↪ -0.00633367088622], [1.6868597855e-05, 2.28849403652e-05, 2.94781294155e-05])
```

7.11.3 Class With Alpha Functions

The class-based ones van save a little code when implementing a new EOS. If there is not a standalone function available for an alpha function, it has not yet been accelerated in a nice vectorized way.

class thermo.eos_alpha_functions.**a_alpha_base**

Bases: object

class thermo.eos_alpha_functions.**Almeida_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Almeida et al. (1991) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Almeida et al. (1991) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = e^{c_1\left(-\frac{T}{T_{c,i}}+1\right)\left|\frac{T}{T_{c,i}}-1\right|^{c_2-1}+c_3\left(-1+\frac{T_{c,i}}{T}\right)}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Androulakis_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Androulakis et al. (1989) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Androulakis et al. (1989) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more

documentation. Three coefficients needed.

$$\alpha = c_1\left(-\left(\frac{T}{T_{c,i}}\right)^{\frac{2}{3}} + 1\right) + c_2\left(-\left(\frac{T}{T_{c,i}}\right)^{\frac{2}{3}} + 1\right)^2 + c_3\left(-\left(\frac{T}{T_{c,i}}\right)^{\frac{2}{3}} + 1\right)^3 + 1$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Chen_Yang_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Hamid and Yang (2017) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Hamid and Yang (2017) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Seven coefficients needed.

$$\alpha = e^{\left(-c_3^{\ln\left(\frac{T}{T_{c,i}}\right)} + 1\right)\left(-\frac{Tc_2}{T_{c,i}} + c_1\right)}$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Coquelet_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Coquelet et al. (2004) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Coquelet et al. (2004) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = e^{c_1\left(-\frac{T}{T_{c,i}}+1\right)\left(c_2\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)^2+c_3\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)^3+1\right)^2}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Gasem_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Gasem (2001) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Gasem (2001) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = e^{\left(-\left(\frac{T}{T_{c,i}}\right)^{c_3}+1\right)\left(\frac{Tc_2}{T_{c,i}}+c_1\right)}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Gibbons_Laughton_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Gibbons and Laughton (1984) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Gibbons and Laughton (1984) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficients needed.

$$\alpha = c_{1} \left(\frac{T}{T_{c,i}} - 1\right) + c_{2} \left(\sqrt{\frac{T}{T_{c,i}}} - 1\right) + 1$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Haghtalab_a_alpha**

Bases: `thermo.eos_alpha_functions.a_alpha_base`

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Haghtalab et al. (2010) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Haghtalab et al. (2010) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = e^{\left(- c_{3}^{\ln{\left(\frac{T}{T_{c,i}} \right)}} + 1\right) \left(- \frac{T c_{2}}{T_{c,i}} + c_{1}\right)}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Harmens_Knapp_a_alpha**

Bases: `thermo.eos_alpha_functions.a_alpha_base`

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Harmens and Knapp (1980) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Harmens and Knapp (1980) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficients needed.

$$\alpha = \left(c_{1}\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)-c_{2}\left(1-\frac{T_{c,i}}{T}\right)+1\right)^{2}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Heyen_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Heyen (1980) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Heyen (1980) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficients needed.

$$\alpha = e^{c_{1}\left(-\left(\frac{T}{T_{c,i}}\right)^{c_{2}}+1\right)}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Mathias_1983_a_alpha**
Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Mathias (1983) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)
Method to calculate *a_alpha* and its first and second derivatives according to Mathias (1983) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficients needed.

$$\alpha = \left(c_{1}\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)-c_{2}\left(-\frac{T}{T_{c,i}}+0.7\right)\left(-\frac{T}{T_{c,i}}+1\right)+1\right)^{2}$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Mathias_Copeman_a_alpha**
Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Mathias and Copeman (1983) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)
Method to calculate *a_alpha* and its first and second derivatives according to Mathias and Copeman (1983) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more

documentation. Three coefficients needed.

$$\alpha = \left(c_1\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)+c_2\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)^2+c_3\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)^3+1\right)^2$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Mathias_Copeman_poly_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure	
a_alpha_and_derivatives_vectorized	
a_alpha_pure	
a_alphas_vectorized	

a_alpha_and_derivatives_pure(T)

a_alpha_and_derivatives_vectorized(T)

a_alpha_pure(T)

a_alphas_vectorized(T)

class thermo.eos_alpha_functions.**Mathias_Copeman_untruncated_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Mathias and Copeman (1983) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Mathias and Copeman (1983) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more

documentation. Three coefficients needed.

$$\alpha = \left(c_{1} \left(- \sqrt{\frac{T}{T_{c,i}}} + 1\right) + c_{2} \left(- \sqrt{\frac{T}{T_{c,i}}} + 1\right)^{2} + c_{3} \left(- \sqrt{\frac{T}{T_{c,i}}} + 1\right)^{3} + 1\right)^{2}$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Melhem_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Melhem et al. (1989) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Melhem et al. (1989) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficients needed.

$$\alpha = e^{c_{1} \left(- \frac{T}{T_{c,i}} + 1\right) + c_{2} \left(- \sqrt{\frac{T}{T_{c,i}}} + 1\right)^{2}}$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Poly_a_alpha**

Bases: object

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives given that there is a polynomial equation for α.
a_alpha_pure(T)	Method to calculate *a_alpha* given that there is a polynomial equation for α.

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives given that there is a polynomial equation for α.

$$a\alpha = a \cdot \text{poly}(T)$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alpha_pure(T)

Method to calculate *a_alpha* given that there is a polynomial equation for α.

$$a\alpha = a \cdot \text{poly}(T)$$

Parameters

T [float] Temperature, [K]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

class thermo.eos_alpha_functions.**Saffari_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Saffari and Zahedi (2013) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Saffari and Zahedi (2013) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = e^{\frac{Tc_1}{T_{c,i}} + c_2 \ln\left(\frac{T}{T_{c,i}}\right) + c_3\left(-\sqrt{\frac{T}{T_{c,i}}} + 1\right)}$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Schwartzentruber_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Schwartzentruber et al. (1990) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Schwartzentruber et al. (1990) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = \left(c_{4}\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)-\left(-\sqrt{\frac{T}{T_{c,i}}}+1\right)\left(\frac{T^{2}c_{3}}{Tc^{2}}+\frac{Tc_{2}}{T_{c,i}}+c_{1}\right)+1\right)^{2}$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Soave_1972_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Soave (1972) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Soave (1972) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Same as *SRK.a_alpha_and_derivatives* but slower and requiring *alpha_coeffs* to be set. One coefficient needed.

$$\alpha = \left(c_{0} \left(- \sqrt{\frac{T}{T_{c,i}}} + 1\right) + 1\right)^{2}$$

References

[1], [2]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Soave_1984_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Soave (1984) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)

Method to calculate *a_alpha* and its first and second derivatives according to Soave (1984) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficients needed.

$$\alpha = c_{1} \left(- \frac{T}{T_{c,i}} + 1\right) + c_{2} \left(-1 + \frac{T_{c,i}}{T}\right) + 1$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Soave_1979_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Soave (1979) [1].

a_alpha_and_derivatives_vectorized	
a_alpha_pure	
a_alphas_vectorized	

a_alpha_and_derivatives_pure(T)
Method to calculate *a_alpha* and its first and second derivatives according to Soave (1979) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. Three coefficients are needed.

$$\alpha = 1 + (1 - T_r)(M + \frac{N}{T_r})$$

References

[1]

a_alpha_and_derivatives_vectorized(T)

a_alpha_pure(T)

a_alphas_vectorized(T)

class thermo.eos_alpha_functions.**Soave_1993_a_alpha**
Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Soave (1983) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(T)
Method to calculate *a_alpha* and its first and second derivatives according to Soave (1983) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Two coefficient needed.

$$\alpha = c_1 \left(-\frac{T}{T_{c,i}} + 1\right) + c_2 \left(-\sqrt{\frac{T}{T_{c,i}}} + 1\right)^2 + 1$$

References

[1]

a_alpha_pure(T)

class thermo.eos_alpha_functions.**Trebble_Bishnoi_a_alpha**
Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Trebble and Bishnoi (1987) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Trebble and Bishnoi (1987) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. One coefficient needed.

$$\alpha = e^{c_1\left(-\frac{T}{T_{c,i}}+1\right)}$$

References

[1]

a_alpha_pure(*T*)

class thermo.eos_alpha_functions.**Twu91_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Twu et al. (1991) [1].
a_alpha_and_derivatives_vectorized(T)	Method to calculate the pure-component *a_alphas* and their first and second derivatives for TWU91 alpha function EOS.

a_alpha_pure	
a_alphas_vectorized	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Twu et al. (1991) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Three coefficients needed.

$$\alpha = \left(\frac{T}{T_{c,i}}\right)^{c_3(c_2-1)} e^{c_1\left(-\left(\frac{T}{T_{c,i}}\right)^{c_2 c_3}+1\right)}$$

References

[1]

a_alpha_and_derivatives_vectorized(*T*)

Method to calculate the pure-component *a_alphas* and their first and second derivatives for TWU91 alpha function EOS. This vectorized implementation is added for extra speed.

$$\alpha = \left(\frac{T}{T_{c,i}}\right)^{c_3(c_2-1)} e^{c_1\left(-\left(\frac{T}{T_{c,i}}\right)^{c_2c_3}+1\right)}$$

Parameters

T [float] Temperature, [K]

Returns

a_alphas [list[float]] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dTs [list[float]] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2s [list[float]] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K**2]

a_alpha_pure(*T*)

a_alphas_vectorized(*T*)

class thermo.eos_alpha_functions.**TwuPR95_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function.
a_alpha_pure(T)	Method to calculate $a\alpha$ for the Twu alpha function.

a_alpha_and_derivatives_vectorized	
a_alphas_vectorized	

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.125283, 0.911807, 1.948150;

L1, M1, N1 = 0.511614, 0.784054, 2.812520

For supercritical conditions:

L0, M0, N0 = 0.401219, 4.963070, -0.2;

L1, M1, N1 = 0.024955, 1.248089, -8.

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

The derivatives are somewhat long and are not described here for brevity; they are obtainable from the following SymPy expression.

```
>>> from sympy import *
>>> T, Tc, omega, N1, N0, M1, M0, L1, L0 = symbols('T, Tc, omega, N1, N0, M1,␣
↪M0, L1, L0')
>>> Tr = T/Tc
>>> alpha0 = Tr**(N0*(M0-1))*exp(L0*(1-Tr**(N0*M0)))
>>> alpha1 = Tr**(N1*(M1-1))*exp(L1*(1-Tr**(N1*M1)))
>>> alpha = alpha0 + omega*(alpha1-alpha0)
>>> diff(alpha, T)
>>> diff(alpha, T, T)
```

a_alpha_and_derivatives_vectorized(*T*)

a_alpha_pure(*T*)

Method to calculate $a\alpha$ for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.125283, 0.911807, 1.948150;

L1, M1, N1 = 0.511614, 0.784054, 2.812520

For supercritical conditions:

L0, M0, N0 = 0.401219, 4.963070, -0.2;

L1, M1, N1 = 0.024955, 1.248089, -8.

Parameters

T [float] Temperature at which to calculate the value, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

a_alphas_vectorized(*T*)

class thermo.eos_alpha_functions.**TwuSRK95_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function.
a_alpha_pure(T)	Method to calculate $a\alpha$ for the Twu alpha function.

a_alpha_and_derivatives_vectorized	
a_alphas_vectorized	

a_alpha_and_derivatives_pure(*T*)

Method to calculate $a\alpha$ and its first and second derivatives for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.141599, 0.919422, 2.496441

L1, M1, N1 = 0.500315, 0.799457, 3.291790

For supercritical conditions:

L0, M0, N0 = 0.441411, 6.500018, -0.20

L1, M1, N1 = 0.032580, 1.289098, -8.0

Parameters

T [float] Temperature at which to calculate the values, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

da_alpha_dT [float] Temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K]

d2a_alpha_dT2 [float] Second temperature derivative of coefficient calculated by EOS-specific method, [J^2/mol^2/Pa/K^2]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

The derivatives are somewhat long and are not described here for brevity; they are obtainable from the following SymPy expression.

```
>>> from sympy import *
>>> T, Tc, omega, N1, N0, M1, M0, L1, L0 = symbols('T, Tc, omega, N1, N0, M1,␣
↪M0, L1, L0')
>>> Tr = T/Tc
>>> alpha0 = Tr**(N0*(M0-1))*exp(L0*(1-Tr**(N0*M0)))
>>> alpha1 = Tr**(N1*(M1-1))*exp(L1*(1-Tr**(N1*M1)))
>>> alpha = alpha0 + omega*(alpha1-alpha0)
>>> diff(alpha, T)
>>> diff(alpha, T, T)
```

a_alpha_and_derivatives_vectorized(*T*)

a_alpha_pure(*T*)

Method to calculate $a\alpha$ for the Twu alpha function. Uses the set values of *Tc*, *omega* and *a*.

$$\alpha = \alpha^{(0)} + \omega(\alpha^{(1)} - \alpha^{(0)})$$

$$\alpha^{(i)} = T_r^{N(M-1)} \exp[L(1 - T_r^{NM})]$$

For sub-critical conditions:

L0, M0, N0 = 0.141599, 0.919422, 2.496441

L1, M1, N1 = 0.500315, 0.799457, 3.291790

For supercritical conditions:

L0, M0, N0 = 0.441411, 6.500018, -0.20

L1, M1, N1 = 0.032580, 1.289098, -8.0

Parameters

T [float] Temperature at which to calculate the value, [-]

Returns

a_alpha [float] Coefficient calculated by EOS-specific method, [J^2/mol^2/Pa]

Notes

This method does not alter the object's state and the temperature provided can be a different than that of the object.

a_alphas_vectorized(*T*)

class thermo.eos_alpha_functions.**Yu_Lu_a_alpha**

Bases: *thermo.eos_alpha_functions.a_alpha_base*

Methods

a_alpha_and_derivatives_pure(T)	Method to calculate *a_alpha* and its first and second derivatives according to Yu and Lu (1987) [1].

a_alpha_pure	

a_alpha_and_derivatives_pure(*T*)

Method to calculate *a_alpha* and its first and second derivatives according to Yu and Lu (1987) [1]. Returns *a_alpha*, *da_alpha_dT*, and *d2a_alpha_dT2*. See *GCEOS.a_alpha_and_derivatives* for more documentation. Four coefficients needed.

$$\alpha = 10^{c_{4} \left(- \frac{T}{T_{c,i}} + 1\right) \left(\frac{T^{2} c_{3}}{Tc^{2}} + \frac{T c_{2}}{T_{c,i}} + c_{1}\right)}$$

References

[1]

a_alpha_pure(*T*)

7.11.4 Pure Alpha Functions

thermo.eos_alpha_functions.**Twu91_alpha_pure**(*T*, *Tc*, *c0*, *c1*, *c2*)

thermo.eos_alpha_functions.**Soave_1972_alpha_pure**(*T*, *Tc*, *c0*)

thermo.eos_alpha_functions.**Soave_1979_alpha_pure**(*T*, *Tc*, *M*, *N*)

thermo.eos_alpha_functions.**Heyen_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Harmens_Knapp_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Mathias_1983_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Mathias_Copeman_untruncated_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Gibbons_Laughton_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Soave_1984_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Yu_Lu_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*, *c4*)

thermo.eos_alpha_functions.**Trebble_Bishnoi_alpha_pure**(*T*, *Tc*, *c1*) thermo.eos_alpha_functions.**Melhem_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Androulakis_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Schwartzentruber_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*, *c4*)

thermo.eos_alpha_functions.**Almeida_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Soave_1993_alpha_pure**(*T*, *Tc*, *c1*, *c2*)

thermo.eos_alpha_functions.**Gasem_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Coquelet_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Haghtalab_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Saffari_alpha_pure**(*T*, *Tc*, *c1*, *c2*, *c3*)

thermo.eos_alpha_functions.**Chen_Yang_alpha_pure**(*T*, *Tc*, *omega*, *c1*, *c2*, *c3*, *c4*, *c5*, *c6*, *c7*)

7.12 Equilibrium State (thermo.equilibrium)

This module contains an object designed to store the result of a flash calculation and provide convinient access to all properties of the calculated phases and bulks.

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

7.12.1 EquilibriumState

class thermo.equilibrium.**EquilibriumState**(*T*, *P*, *zs*, *gas*, *liquids*, *solids*, *betas*, *flash_specs=None*, *flash_convergence=None*, *constants=None*, *correlations=None*, *flasher=None*, *settings=<thermo.bulk.BulkSettings object>*)

Class to represent a thermodynamic equilibrium state with one or more phases in it. This object is designed to be the output of the `thermo.flash.Flash` interface and to provide easy acess to all properties of the mixture.

Properties like `Cp` are calculated using the mixing rules configured by the `BulkSettings` object. For states with a single phase, this will always reduce to the properties of that phase.

This interface allows calculation of thermodynamic properties, and transport properties. Both molar and mass outputs are provided, as separate calls (ex. `Cp` and `Cp_mass`).

Parameters

T [float] Temperature of state, [K]

P [float] Pressure of state, [Pa]

zs [list[float]] Overall mole fractions of all species in the state, [-]

gas [*Phase*] The calcualted gas phase object, if one was found, [-]

liquids [list[*Phase*]] A list of liquid phase objects, if any were found, [-]

solids [list[*Phase*]] A list of solid phase objects, if any were found, [-]

betas [list[float]] Molar phase fractions of every phase, ordered [*gas beta*, *liquid beta0*, *liquid beta1*, . . . , *solid beta0*, *solid beta1*, . . .]

flash_specs [dict[str][float], optional] A dictionary containing the specifications for the flash calculations, [-]

flash_convergence [dict[str][float], optional] A dictionary containing the convergence results for the flash calculations; this is to help support development of the library only and the contents of this dictionary is subject to change, [-]

constants [*ChemicalConstantsPackage*, optional] Package of chemical constants; all cases these properties are accessible as attributes of this object, [-] *EquilibriumState* object, [-]

correlations [*PropertyCorrelationsPackage*, optional] Package of chemical T-dependent properties; these properties are accessible as attributes of this object object, [-]

flasher [*Flash* object, optional] This reference can be provided to this object to allow the object to return properties which are themselves calculated from results of flash calculations, [-]

settings [*BulkSettings*, optional] Object containing settings for calculating bulk and transport properties, [-]

Examples

The following sample shows a flash for the CO2-n-hexane system with all constants provided, using no data from thermo.

```
>>> from thermo import *
>>> constants = ChemicalConstantsPackage(names=['carbon dioxide', 'hexane'], CASs=[
↪'124-38-9', '110-54-3'], MWs=[44.0095, 86.17536], omegas=[0.2252, 0.2975],␣
↪Pcs=[7376460.0, 3025000.0], Tbs=[194.67, 341.87], Tcs=[304.2, 507.6], Tms=[216.65,
↪ 178.075])
>>> correlations = PropertyCorrelationsPackage(constants=constants, skip_
↪missing=True,
...                                                                    ␣
↪HeatCapacityGases=[HeatCapacityGas(poly_fit=(50.0, 1000.0, [-3.1115474168865828e-
↪21, 1.39156078498805e-17, -2.5430881416264243e-14, 2.4175307893014295e-11, -1.
↪2437314771044867e-08, 3.1251954264658904e-06, -0.00021220221928610925, 0.
↪000884685506352987, 29.266811602924644])),
...                                                                    ␣
↪HeatCapacityGas(poly_fit=(200.0, 1000.0, [1.3740654453881647e-21, -8.
↪344496203280677e-18, 2.2354782954548568e-14, -3.4659555330048226e-11, 3.
↪410703030634579e-08, -2.1693611029230923e-05, 0.008373280796376588, -1.
↪356180511425385, 175.67091124888998]))])
>>> eos_kwargs = {'Pcs': constants.Pcs, 'Tcs': constants.Tcs, 'omegas': constants.
↪omegas}
>>> gas = CEOSGas(PRMIX, eos_kwargs, HeatCapacityGases=correlations.
↪HeatCapacityGases)
```

(continues on next page)

(continued from previous page)

```
>>> liq = CEOSLiquid(PRMIX, eos_kwargs, HeatCapacityGases=correlations.
↪HeatCapacityGases)
>>> flasher = FlashVL(constants, correlations, liquid=liq, gas=gas)
>>> state = flasher.flash(P=1e5, T=196.0, zs=[0.5, 0.5])
>>> type(state) is EquilibriumState
True
>>> state.phase_count
2
>>> state.bulk.Cp()
108.3164692
>>> state.flash_specs
{'zs': [0.5, 0.5], 'T': 196.0, 'P': 100000.0}
>>> state.Tms
[216.65, 178.075]
>>> state.liquid0.H()
-34376.4853
>>> state.gas.H()
-3608.0551
```

Attributes

gas_count [int] Number of gas phases present (0 or 1), [-]

liquid_count [int] Number of liquid phases present, [-]

solid_count [int] Number of solid phases present, [-]

phase_count [int] Number of phases present, [-]

gas_beta [float] Molar phase fraction of the gas phase; 0 if no gas phase is present, [-]

liquids_betas [list[float]] Liquid molar phase fractions, [-]

solids_betas [list[float]] Solid molar phase fractions, [-]

liquid_zs [list[float]] Overall mole fractions of each component in the overall liquid phase, [-]

liquid_bulk [*Bulk*] Liquid phase bulk, [-]

solid_zs [list[float]] Overall mole fractions of each component in the overall solid phase, [-]

solid_bulk [*Bulk*] Solid phase bulk, [-]

bulk [*Bulk*] Overall phase bulk, [-]

IDs Alias of CASs.

LF Method to return the liquid fraction of the equilibrium state.

VF Method to return the vapor fraction of the equilibrium state.

betas_liquids Method to calculate and return the fraction of the liquid phase that each liquid phase is, by molar phase fraction.

betas_mass Method to calculate and return the mass fraction of all of the phases in the system.

betas_mass_liquids Method to calculate and return the fraction of the liquid phase that each liquid phase is, by mass phase fraction.

betas_mass_states Method to return the mass phase fractions of each of the three fundamental *types* of phases.

betas_states Method to return the molar phase fractions of each of the three fundamental *types* of phases.

betas_volume Method to calculate and return the volume fraction of all of the phases in the system.

betas_volume_liquids Method to calculate and return the fraction of the liquid phase that each liquid phase is, by volume phase fraction.

betas_volume_states Method to return the volume phase fractions of each of the three fundamental *types* of phases.

heaviest_liquid The liquid-like phase with the highest mass density, [-]

lightest_liquid The liquid-like phase with the lowest mass density, [-]

phase Method to calculate and return a string representing the phase of the mixture.

quality Method to return the mass vapor fraction of the equilibrium state.

water_index The index of the component water in the components.

water_phase The liquid-like phase with the highest water mole fraction, [-]

water_phase_index The liquid-like phase with the highest mole fraction of water, [-]

atomss Breakdown of each component into its elements and their counts, as a dict, [-].

Carcinogens Status of each component in cancer causing registries, [-].

CASs CAS registration numbers for each component, [-].

Ceilings Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

charges Charge number (valence) for each component, [-].

conductivities Electrical conductivities for each component, [S/m].

dipoles Dipole moments for each component, [debye].

economic_statuses Status of each component in in relation to import and export from various regions, [-].

formulas Formulas of each component, [-].

Gfgs Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

Gfgs_mass Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

GWPs Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

Hcs Higher standard molar heats of combustion for each component, [J/mol].

Hcs_lower Lower standard molar heats of combustion for each component, [J/mol].

Hcs_lower_mass Lower standard heats of combustion for each component, [J/kg].

Hcs_mass Higher standard heats of combustion for each component, [J/kg].

Hfgs Ideal gas standard molar enthalpies of formation for each component, [J/mol].

Hfgs_mass Ideal gas standard enthalpies of formation for each component, [J/kg].

Hfus_Tms Molar heats of fusion for each component at their respective melting points, [J/mol].

Hfus_Tms_mass Heats of fusion for each component at their respective melting points, [J/kg].

Hsub_Tts Heats of sublimation for each component at their respective triple points, [J/mol].

Hsub_Tts_mass Heats of sublimation for each component at their respective triple points, [J/kg].

Hvap_298s Molar heats of vaporization for each component at 298.15 K, [J/mol].

Hvap_298s_mass Heats of vaporization for each component at 298.15 K, [J/kg].

Hvap_Tbs Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

Hvap_Tbs_mass Heats of vaporization for each component at their respective normal boiling points, [J/kg].

InChI_Keys InChI Keys for each component, [-].

InChIs InChI strings for each component, [-].

legal_statuses Status of each component in in relation to import and export rules from various regions, [-].

LFLs Lower flammability limits for each component, [-].

logPs Octanol-water partition coefficients for each component, [-].

molecular_diameters Lennard-Jones molecular diameters for each component, [angstrom].

MWs Similatiry variables for each component, [g/mol].

names Names for each component, [-].

ODPs Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

omegas Acentric factors for each component, [-].

Parachors Parachors for each component, [N^0.25*m^2.75/mol].

Pcs Critical pressures for each component, [Pa].

phase_STPs Standard states ('g', 'l', or 's') for each component, [-].

Psat_298s Vapor pressures for each component at 298.15 K, [Pa].

PSRK_groups PSRK subgroup: count groups for each component, [-].

Pts Triple point pressures for each component, [Pa].

PubChems Pubchem IDs for each component, [-].

rhocs Molar densities at the critical point for each component, [mol/m^3].

rhocs_mass Densities at the critical point for each component, [kg/m^3].

rhol_STPs Molar liquid densities at STP for each component, [mol/m^3].

rhol_STPs_mass Liquid densities at STP for each component, [kg/m^3].

RIs Refractive indexes for each component, [-].

S0gs Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

S0gs_mass Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

Sfgs Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

Sfgs_mass Ideal gas standard entropies of formation for each component, [J/(kg*K)].

similarity_variables Similarity variables for each component, [mol/g].

Skins Whether each compound can be absorbed through the skin or not, [-].

smiless SMILES identifiers for each component, [-].

STELs Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

StielPolars Stiel polar factors for each component, [-].

Stockmayers Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

Tautoignitions Autoignition temperatures for each component, [K].

Tbs Boiling temperatures for each component, [K].

Tcs Critical temperatures for each component, [K].

Tflashs Flash point temperatures for each component, [K].

Tms Melting temperatures for each component, [K].

Tts Triple point temperatures for each component, [K].

TWAs Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

UFLs Upper flammability limits for each component, [-].

UNIFAC_Dortmund_groups UNIFAC_Dortmund_group: count groups for each component, [-].

UNIFAC_groups UNIFAC_group: count groups for each component, [-].

Van_der_Waals_areas Unnormalized Van der Waals areas for each component, [m^2/mol].

Van_der_Waals_volumes Unnormalized Van der Waals volumes for each component, [m^3/mol].

Vcs Critical molar volumes for each component, [m^3/mol].

Vml_STPs Liquid molar volumes for each component at STP, [m^3/mol].

Vml_Tms Liquid molar volumes for each component at their respective melting points, [m^3/mol].

Zcs Critical compressibilities for each component, [-].

UNIFAC_Rs UNIFAC R parameters for each component, [-].

UNIFAC_Qs UNIFAC Q parameters for each component, [-].

rhos_Tms Solid molar densities for each component at their respective melting points, [mol/m^3].

Vms_Tms Solid molar volumes for each component at their respective melting points, [m^3/mol].

rhos_Tms_mass Solid mass densities for each component at their melting point, [kg/m^3].

solubility_parameters Solubility parameters for each component at 298.15 K, [Pa^0.5].

Vml_60Fs Liquid molar volumes for each component at 60 °F, [m^3/mol].

rhol_60Fs Liquid molar densities for each component at 60 °F, [mol/m^3].

rhol_60Fs_mass Liquid mass densities for each component at 60 °F, [kg/m^3].

conductivity_Ts Temperatures at which the electrical conductivities for each component were measured, [K].

RI_Ts Temperatures at which the refractive indexes were reported for each component, [K].

Vmg_STPs Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

rhog_STPs Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

rhog_STPs_mass Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

sigma_STPs Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

sigma_Tms Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

sigma_Tbs Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

Hf_STPs Standard state molar enthalpies of formation for each component, [J/mol].

Hf_STPs_mass Standard state mass enthalpies of formation for each component, [J/kg].

VaporPressures Wrapper to obtain the list of VaporPressures objects of the associated *PropertyCorrelationsPackage*.

VolumeLiquids Wrapper to obtain the list of VolumeLiquids objects of the associated *PropertyCorrelationsPackage*.

VolumeGases Wrapper to obtain the list of VolumeGases objects of the associated *PropertyCorrelationsPackage*.

VolumeSolids Wrapper to obtain the list of VolumeSolids objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityGases Wrapper to obtain the list of HeatCapacityGases objects of the associated *PropertyCorrelationsPackage*.

HeatCapacitySolids Wrapper to obtain the list of HeatCapacitySolids objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityLiquids Wrapper to obtain the list of HeatCapacityLiquids objects of the associated *PropertyCorrelationsPackage*.

EnthalpyVaporizations Wrapper to obtain the list of EnthalpyVaporizations objects of the associated *PropertyCorrelationsPackage*.

EnthalpySublimations Wrapper to obtain the list of EnthalpySublimations objects of the associated *PropertyCorrelationsPackage*.

SublimationPressures Wrapper to obtain the list of SublimationPressures objects of the associated *PropertyCorrelationsPackage*.

PermittivityLiquids Wrapper to obtain the list of PermittivityLiquids objects of the associated *PropertyCorrelationsPackage*.

ViscosityLiquids Wrapper to obtain the list of ViscosityLiquids objects of the associated *PropertyCorrelationsPackage*.

ViscosityGases Wrapper to obtain the list of ViscosityGases objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityLiquids Wrapper to obtain the list of ThermalConductivityLiquids objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityGases Wrapper to obtain the list of ThermalConductivityGases objects of the associated *PropertyCorrelationsPackage*.

SurfaceTensions Wrapper to obtain the list of SurfaceTensions objects of the associated *PropertyCorrelationsPackage*.

VolumeGasMixture Wrapper to obtain the list of VolumeGasMixture objects of the associated *PropertyCorrelationsPackage*.

VolumeLiquidMixture Wrapper to obtain the list of VolumeLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

VolumeSolidMixture Wrapper to obtain the list of VolumeSolidMixture objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityGasMixture Wrapper to obtain the list of HeatCapacityGasMixture objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityLiquidMixture Wrapper to obtain the list of HeatCapacityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

HeatCapacitySolidMixture Wrapper to obtain the list of HeatCapacitySolidMixture objects of the associated *PropertyCorrelationsPackage*.

ViscosityGasMixture Wrapper to obtain the list of ViscosityGasMixture objects of the associated *PropertyCorrelationsPackage*.

ViscosityLiquidMixture Wrapper to obtain the list of ViscosityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityGasMixture Wrapper to obtain the list of ThermalConductivityGasMixture objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityLiquidMixture Wrapper to obtain the list of ThermalConductivityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

SurfaceTensionMixture Wrapper to obtain the list of SurfaceTensionMixture objects of the associated *PropertyCorrelationsPackage*.

Methods

A()	Method to calculate and return the Helmholtz energy of the phase.
API([phase])	Method to calculate and return the API of the phase.
A_dep()	Method to calculate and return the departure Helmholtz energy of the phase.
A_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas Helmholtz energy of formation of the phase (as if the phase was an ideal gas).
A_ideal_gas([phase])	Method to calculate and return the ideal-gas Helmholtz energy of the phase.
A_mass([phase])	Method to calculate and return mass Helmholtz energy of the phase.
A_reactive()	Method to calculate and return the Helmholtz free energy of the phase on a reactive basis.
Bvirial([phase])	Method to calculate and return the *B* virial coefficient of the phase at its current conditions.
Cp()	Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase.

continues on next page

Table 65 – continued from previous page

Cp_Cv_ratio()	Method to calculate and return the Cp/Cv ratio of the phase.
Cp_Cv_ratio_ideal_gas([phase])	Method to calculate and return the ratio of the ideal-gas heat capacity to its constant-volume heat capacity.
Cp_dep([phase])	Method to calculate and return the difference between the actual Cp and the ideal-gas heat capacity C_p^{ig} of the phase.
Cp_ideal_gas([phase])	Method to calculate and return the ideal-gas heat capacity of the phase.
Cp_mass([phase])	Method to calculate and return mass constant pressure heat capacity of the phase.
Cv()	Method to calculate and return the constant-volume heat capacity Cv of the phase.
Cv_dep([phase])	Method to calculate and return the difference between the actual Cv and the ideal-gas constant volume heat capacity C_v^{ig} of the phase.
Cv_ideal_gas([phase])	Method to calculate and return the ideal-gas constant volume heat capacity of the phase.
Cv_mass([phase])	Method to calculate and return mass constant volume heat capacity of the phase.
G()	Method to calculate and return the Gibbs free energy of the phase.
G_dep()	Method to calculate and return the departure Gibbs free energy of the phase.
G_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas Gibbs free energy of formation of the phase (as if the phase was an ideal gas).
G_ideal_gas([phase])	Method to calculate and return the ideal-gas Gibbs free energy of the phase.
G_mass([phase])	Method to calculate and return mass Gibbs energy of the phase.
G_reactive()	Method to calculate and return the Gibbs free energy of the phase on a reactive basis.
H()	Method to calculate and return the constant-temperature and constant phase-fraction enthalpy of the bulk phase.
H_C_ratio([phase])	Method to calculate and return the atomic ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.
H_C_ratio_mass([phase])	Method to calculate and return the mass ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.
H_dep([phase])	Method to calculate and return the difference between the actual H and the ideal-gas enthalpy of the phase.
H_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas enthalpy of formation of the phase (as if the phase was an ideal gas).
H_ideal_gas([phase])	Method to calculate and return the ideal-gas enthalpy of the phase.

continued on next page

Table 65 – continued from previous page

H_mass([phase])	Method to calculate and return mass enthalpy of the phase.
H_reactive()	Method to calculate and return the constant-temperature and constant phase-fraction reactive enthalpy of the bulk phase.
Hc([phase])	Method to calculate and return the molar ideal-gas higher heat of combustion of the object, [J/mol]
Hc_lower([phase])	Method to calculate and return the molar ideal-gas lower heat of combustion of the object, [J/mol]
Hc_lower_mass([phase])	Method to calculate and return the mass ideal-gas lower heat of combustion of the object, [J/mol]
Hc_lower_normal([phase])	Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the normal gas volume, [J/m^3]
Hc_lower_standard([phase])	Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the standard gas volume, [J/m^3]
Hc_mass([phase])	Method to calculate and return the mass ideal-gas higher heat of combustion of the object, [J/mol]
Hc_normal([phase])	Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the normal gas volume, [J/m^3]
Hc_standard([phase])	Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the standard gas volume, [J/m^3]
Joule_Thomson()	Method to calculate and return the Joule-Thomson coefficient of the bulk according to the selected calculation methodology.
Ks(phase[, phase_ref])	Method to calculate and return the K-values of each phase.
MW([phase])	Method to calculate and return the molecular weight of the phase.
PIP()	Method to calculate and return the phase identification parameter of the phase.
Pmc([phase])	Method to calculate and return the mechanical critical pressure of the phase.
S()	Method to calculate and return the constant-temperature and constant phase-fraction entropy of the bulk phase.
SG([phase])	Method to calculate and return the standard liquid specific gravity of the phase, using constant liquid pure component densities not calculated by the phase object, at 60 °F.
SG_gas([phase])	Method to calculate and return the specific gravity of the phase with respect to a gas reference density.
S_dep([phase])	Method to calculate and return the difference between the actual *S* and the ideal-gas entropy of the phase.
S_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas entropy of formation of the phase (as if the phase was an ideal gas).

continues on next page

Table 65 – continued from previous page

S_ideal_gas([phase])	Method to calculate and return the ideal-gas entropy of the phase.
S_mass([phase])	Method to calculate and return mass entropy of the phase.
S_reactive()	Method to calculate and return the constant-temperature and constant phase-fraction reactive entropy of the bulk phase.
Tmc([phase])	Method to calculate and return the mechanical critical temperature of the phase.
U()	Method to calculate and return the internal energy of the phase.
U_dep()	Method to calculate and return the departure internal energy of the phase.
U_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas internal energy of formation of the phase (as if the phase was an ideal gas).
U_ideal_gas([phase])	Method to calculate and return the ideal-gas internal energy of the phase.
U_mass([phase])	Method to calculate and return mass internal energy of the phase.
U_reactive()	Method to calculate and return the internal energy of the phase on a reactive basis.
V()	Method to calculate and return the molar volume of the bulk phase.
V_dep()	Method to calculate and return the departure (from ideal gas behavior) molar volume of the phase.
V_gas([phase])	Method to calculate and return the ideal-gas molar volume of the phase at the chosen reference temperature and pressure, according to the temperature variable *T_gas_ref* and pressure variable *P_gas_ref* of the thermo.bulk.BulkSettings.
V_gas_normal([phase])	Method to calculate and return the ideal-gas molar volume of the phase at the normal temperature and pressure, according to the temperature variable *T_normal* and pressure variable *P_normal* of the thermo.bulk.BulkSettings.
V_gas_standard([phase])	Method to calculate and return the ideal-gas molar volume of the phase at the standard temperature and pressure, according to the temperature variable *T_standard* and pressure variable *P_standard* of the thermo.bulk.BulkSettings.
V_ideal_gas([phase])	Method to calculate and return the ideal-gas molar volume of the phase.
V_iter([phase, force])	Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure.
V_liquid_ref([phase])	Method to calculate and return the liquid reference molar volume according to the temperature variable *T_liquid_volume_ref* of thermo.bulk.BulkSettings and the composition of the phase.

continues on next page

Table 65 – continued from previous page

V_liquids_ref()	Method to calculate and return the liquid reference molar volumes according to the temperature variable *T_liquid_volume_ref* of thermo.bulk.BulkSettings.
V_mass([phase])	Method to calculate and return the specific volume of the phase.
Vfgs([phase])	Method to calculate and return the ideal-gas volume fractions of the components of the phase.
Vfls([phase])	Method to calculate and return the ideal-liquid volume fractions of the components of the phase, using the standard liquid densities at the temperature variable *T_liquid_volume_ref* of thermo.bulk.BulkSettings and the composition of the phase.
Vmc([phase])	Method to calculate and return the mechanical critical volume of the phase.
Wobbe_index([phase])	Method to calculate and return the molar Wobbe index of the object, [J/mol].
Wobbe_index_lower([phase])	Method to calculate and return the molar lower Wobbe index of the
Wobbe_index_lower_mass([phase])	Method to calculate and return the lower mass Wobbe index of the object, [J/kg].
Wobbe_index_lower_normal([phase])	Method to calculate and return the volumetric normal lower Wobbe index of the object, [J/m^3].
Wobbe_index_lower_standard([phase])	Method to calculate and return the volumetric standard lower Wobbe index of the object, [J/m^3].
Wobbe_index_mass([phase])	Method to calculate and return the mass Wobbe index of the object, [J/kg].
Wobbe_index_normal([phase])	Method to calculate and return the volumetric normal Wobbe index of the object, [J/m^3].
Wobbe_index_standard([phase])	Method to calculate and return the volumetric standard Wobbe index of the object, [J/m^3].
Z()	Method to calculate and return the compressibility factor of the phase.
Zmc([phase])	Method to calculate and return the mechanical critical compressibility of the phase.
alpha([phase])	Method to calculate and return the thermal diffusivity of the equilibrium state.
atom_fractions([phase])	Method to calculate and return the atomic composition of the phase; returns a dictionary of atom fraction (by count), containing only those elements who are present.
atom_mass_fractions([phase])	Method to calculate and return the atomic mass fractions of the phase; returns a dictionary of atom fraction (by mass), containing only those elements who are present.
d2P_dT2()	Method to calculate and return the second temperature derivative of pressure of the bulk according to the selected calculation methodology.

continues on next page

Table 65 – continued from previous page

d2P_dT2_frozen()	Method to calculate and return the second constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions.
d2P_dTdV()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the bulk according to the selected calculation methodology.
d2P_dTdV_frozen()	Method to calculate and return the second derivative of pressure with respect to volume and temperature of the bulk phase, at constant phase fractions and phase compositions.
d2P_dV2()	Method to calculate and return the second volume derivative of pressure of the bulk according to the selected calculation methodology.
d2P_dV2_frozen()	Method to calculate and return the constant-temperature second derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions.
dA_dP()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.
dA_dP_T()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.
dA_dP_V()	Method to calculate and return the constant-volume pressure derivative of Helmholtz energy.
dA_dT()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dA_dT_P()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dA_dT_V()	Method to calculate and return the constant-volume temperature derivative of Helmholtz energy.
dA_dV_P()	Method to calculate and return the constant-pressure volume derivative of Helmholtz energy.
dA_dV_T()	Method to calculate and return the constant-temperature volume derivative of Helmholtz energy.
dA_mass_dP()	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.
dA_mass_dP_T()	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.
dA_mass_dP_V()	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant volume.
dA_mass_dT()	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.

continues on next page

Table 65 – continued from previous page

dA_mass_dT_P()	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dT_V()	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant volume.
dA_mass_dV_P()	Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dV_T()	Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant temperature.
dCv_dP_T()	Method to calculate the pressure derivative of Cv, constant volume heat capacity, at constant temperature.
dCv_dT_P()	Method to calculate the temperature derivative of Cv, constant volume heat capacity, at constant pressure.
dCv_mass_dP_T()	Method to calculate and return the pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature.
dCv_mass_dT_P()	Method to calculate and return the temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure.
dG_dP()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dP_T()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dP_V()	Method to calculate and return the constant-volume pressure derivative of Gibbs free energy.
dG_dT()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.
dG_dT_P()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.
dG_dT_V()	Method to calculate and return the constant-volume temperature derivative of Gibbs free energy.
dG_dV_P()	Method to calculate and return the constant-pressure volume derivative of Gibbs free energy.
dG_dV_T()	Method to calculate and return the constant-temperature volume derivative of Gibbs free energy.
dG_mass_dP()	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.
dG_mass_dP_T()	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.
dG_mass_dP_V()	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant volume.

continues on next page

Table 65 – continued from previous page

dG_mass_dT()	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dT_P()	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dT_V()	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant volume.
dG_mass_dV_P()	Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dV_T()	Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant temperature.
dH_dP()	Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.
dH_dP_T()	Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.
dH_dT()	Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase.
dH_dT_P()	Method to calculate and return the temperature derivative of enthalpy of the phase at constant pressure.
dH_mass_dP()	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.
dH_mass_dP_T()	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.
dH_mass_dP_V()	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant volume.
dH_mass_dT()	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dT_P()	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dT_V()	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant volume.
dH_mass_dV_P()	Method to calculate and return the volume derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dV_T()	Method to calculate and return the volume derivative of mass enthalpy of the phase at constant temperature.
dP_dP_A()	Method to calculate and return the pressure derivative of pressure of the phase at constant Helmholtz energy.
dP_dP_G()	Method to calculate and return the pressure derivative of pressure of the phase at constant Gibbs energy.

continues on next page

Table 65 – continued from previous page

dP_dP_H()	Method to calculate and return the pressure derivative of pressure of the phase at constant enthalpy.
dP_dP_S()	Method to calculate and return the pressure derivative of pressure of the phase at constant entropy.
dP_dP_U()	Method to calculate and return the pressure derivative of pressure of the phase at constant internal energy.
dP_dT()	Method to calculate and return the first temperature derivative of pressure of the bulk according to the selected calculation methodology.
dP_dT_A()	Method to calculate and return the temperature derivative of pressure of the phase at constant Helmholtz energy.
dP_dT_G()	Method to calculate and return the temperature derivative of pressure of the phase at constant Gibbs energy.
dP_dT_H()	Method to calculate and return the temperature derivative of pressure of the phase at constant enthalpy.
dP_dT_S()	Method to calculate and return the temperature derivative of pressure of the phase at constant entropy.
dP_dT_U()	Method to calculate and return the temperature derivative of pressure of the phase at constant internal energy.
dP_dT_frozen()	Method to calculate and return the constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions.
dP_dV()	Method to calculate and return the first volume derivative of pressure of the bulk according to the selected calculation methodology.
dP_dV_A()	Method to calculate and return the volume derivative of pressure of the phase at constant Helmholtz energy.
dP_dV_G()	Method to calculate and return the volume derivative of pressure of the phase at constant Gibbs energy.
dP_dV_H()	Method to calculate and return the volume derivative of pressure of the phase at constant enthalpy.
dP_dV_S()	Method to calculate and return the volume derivative of pressure of the phase at constant entropy.
dP_dV_U()	Method to calculate and return the volume derivative of pressure of the phase at constant internal energy.
dP_dV_frozen()	Method to calculate and return the constant-temperature derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions.
dP_drho_A()	Method to calculate and return the density derivative of pressure of the phase at constant Helmholtz energy.
dP_drho_G()	Method to calculate and return the density derivative of pressure of the phase at constant Gibbs energy.
dP_drho_H()	Method to calculate and return the density derivative of pressure of the phase at constant enthalpy.

continues on next page

Table 65 – continued from previous page

dP_drho_S()	Method to calculate and return the density derivative of pressure of the phase at constant entropy.
dP_drho_U()	Method to calculate and return the density derivative of pressure of the phase at constant internal energy.
dS_dP()	Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.
dS_dP_T()	Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.
dS_dV_P()	Method to calculate and return the volume derivative of entropy of the phase at constant pressure.
dS_dV_T()	Method to calculate and return the volume derivative of entropy of the phase at constant temperature.
dS_mass_dP()	Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.
dS_mass_dP_T()	Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.
dS_mass_dP_V()	Method to calculate and return the pressure derivative of mass entropy of the phase at constant volume.
dS_mass_dT()	Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.
dS_mass_dT_P()	Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.
dS_mass_dT_V()	Method to calculate and return the temperature derivative of mass entropy of the phase at constant volume.
dS_mass_dV_P()	Method to calculate and return the volume derivative of mass entropy of the phase at constant pressure.
dS_mass_dV_T()	Method to calculate and return the volume derivative of mass entropy of the phase at constant temperature.
dT_dP_A()	Method to calculate and return the pressure derivative of temperature of the phase at constant Helmholtz energy.
dT_dP_G()	Method to calculate and return the pressure derivative of temperature of the phase at constant Gibbs energy.
dT_dP_H()	Method to calculate and return the pressure derivative of temperature of the phase at constant enthalpy.
dT_dP_S()	Method to calculate and return the pressure derivative of temperature of the phase at constant entropy.
dT_dP_U()	Method to calculate and return the pressure derivative of temperature of the phase at constant internal energy.
dT_dT_A()	Method to calculate and return the temperature derivative of temperature of the phase at constant Helmholtz energy.
dT_dT_G()	Method to calculate and return the temperature derivative of temperature of the phase at constant Gibbs energy.

continues on next page

Table 65 – continued from previous page

dT_dT_H()	Method to calculate and return the temperature derivative of temperature of the phase at constant enthalpy.
dT_dT_S()	Method to calculate and return the temperature derivative of temperature of the phase at constant entropy.
dT_dT_U()	Method to calculate and return the temperature derivative of temperature of the phase at constant internal energy.
dT_dV_A()	Method to calculate and return the volume derivative of temperature of the phase at constant Helmholtz energy.
dT_dV_G()	Method to calculate and return the volume derivative of temperature of the phase at constant Gibbs energy.
dT_dV_H()	Method to calculate and return the volume derivative of temperature of the phase at constant enthalpy.
dT_dV_S()	Method to calculate and return the volume derivative of temperature of the phase at constant entropy.
dT_dV_U()	Method to calculate and return the volume derivative of temperature of the phase at constant internal energy.
dT_drho_A()	Method to calculate and return the density derivative of temperature of the phase at constant Helmholtz energy.
dT_drho_G()	Method to calculate and return the density derivative of temperature of the phase at constant Gibbs energy.
dT_drho_H()	Method to calculate and return the density derivative of temperature of the phase at constant enthalpy.
dT_drho_S()	Method to calculate and return the density derivative of temperature of the phase at constant entropy.
dT_drho_U()	Method to calculate and return the density derivative of temperature of the phase at constant internal energy.
dU_dP()	Method to calculate and return the constant-temperature pressure derivative of internal energy.
dU_dP_T()	Method to calculate and return the constant-temperature pressure derivative of internal energy.
dU_dP_V()	Method to calculate and return the constant-volume pressure derivative of internal energy.
dU_dT()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
dU_dT_P()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
dU_dT_V()	Method to calculate and return the constant-volume temperature derivative of internal energy.
dU_dV_P()	Method to calculate and return the constant-pressure volume derivative of internal energy.
dU_dV_T()	Method to calculate and return the constant-temperature volume derivative of internal energy.

continues on next page

Table 65 – continued from previous page

dU_mass_dP()	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.
dU_mass_dP_T()	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.
dU_mass_dP_V()	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant volume.
dU_mass_dT()	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.
dU_mass_dT_P()	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.
dU_mass_dT_V()	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant volume.
dU_mass_dV_P()	Method to calculate and return the volume derivative of mass internal energy of the phase at constant pressure.
dU_mass_dV_T()	Method to calculate and return the volume derivative of mass internal energy of the phase at constant temperature.
dV_dP_A()	Method to calculate and return the pressure derivative of volume of the phase at constant Helmholtz energy.
dV_dP_G()	Method to calculate and return the pressure derivative of volume of the phase at constant Gibbs energy.
dV_dP_H()	Method to calculate and return the pressure derivative of volume of the phase at constant enthalpy.
dV_dP_S()	Method to calculate and return the pressure derivative of volume of the phase at constant entropy.
dV_dP_U()	Method to calculate and return the pressure derivative of volume of the phase at constant internal energy.
dV_dT_A()	Method to calculate and return the temperature derivative of volume of the phase at constant Helmholtz energy.
dV_dT_G()	Method to calculate and return the temperature derivative of volume of the phase at constant Gibbs energy.
dV_dT_H()	Method to calculate and return the temperature derivative of volume of the phase at constant enthalpy.
dV_dT_S()	Method to calculate and return the temperature derivative of volume of the phase at constant entropy.
dV_dT_U()	Method to calculate and return the temperature derivative of volume of the phase at constant internal energy.
dV_dV_A()	Method to calculate and return the volume derivative of volume of the phase at constant Helmholtz energy.

continues on next page

Table 65 – continued from previous page

dV_dV_G()	Method to calculate and return the volume derivative of volume of the phase at constant Gibbs energy.
dV_dV_H()	Method to calculate and return the volume derivative of volume of the phase at constant enthalpy.
dV_dV_S()	Method to calculate and return the volume derivative of volume of the phase at constant entropy.
dV_dV_U()	Method to calculate and return the volume derivative of volume of the phase at constant internal energy.
dV_drho_A()	Method to calculate and return the density derivative of volume of the phase at constant Helmholtz energy.
dV_drho_G()	Method to calculate and return the density derivative of volume of the phase at constant Gibbs energy.
dV_drho_H()	Method to calculate and return the density derivative of volume of the phase at constant enthalpy.
dV_drho_S()	Method to calculate and return the density derivative of volume of the phase at constant entropy.
dV_drho_U()	Method to calculate and return the density derivative of volume of the phase at constant internal energy.
drho_dP_A()	Method to calculate and return the pressure derivative of density of the phase at constant Helmholtz energy.
drho_dP_G()	Method to calculate and return the pressure derivative of density of the phase at constant Gibbs energy.
drho_dP_H()	Method to calculate and return the pressure derivative of density of the phase at constant enthalpy.
drho_dP_S()	Method to calculate and return the pressure derivative of density of the phase at constant entropy.
drho_dP_U()	Method to calculate and return the pressure derivative of density of the phase at constant internal energy.
drho_dT_A()	Method to calculate and return the temperature derivative of density of the phase at constant Helmholtz energy.
drho_dT_G()	Method to calculate and return the temperature derivative of density of the phase at constant Gibbs energy.
drho_dT_H()	Method to calculate and return the temperature derivative of density of the phase at constant enthalpy.
drho_dT_S()	Method to calculate and return the temperature derivative of density of the phase at constant entropy.
drho_dT_U()	Method to calculate and return the temperature derivative of density of the phase at constant internal energy.
drho_dV_A()	Method to calculate and return the volume derivative of density of the phase at constant Helmholtz energy.
drho_dV_G()	Method to calculate and return the volume derivative of density of the phase at constant Gibbs energy.
drho_dV_H()	Method to calculate and return the volume derivative of density of the phase at constant enthalpy.
drho_dV_S()	Method to calculate and return the volume derivative of density of the phase at constant entropy.

continues on next page

Table 65 – continued from previous page

drho_dV_U()	Method to calculate and return the volume derivative of density of the phase at constant internal energy.
drho_drho_A()	Method to calculate and return the density derivative of density of the phase at constant Helmholtz energy.
drho_drho_G()	Method to calculate and return the density derivative of density of the phase at constant Gibbs energy.
drho_drho_H()	Method to calculate and return the density derivative of density of the phase at constant enthalpy.
drho_drho_S()	Method to calculate and return the density derivative of density of the phase at constant entropy.
drho_drho_U()	Method to calculate and return the density derivative of density of the phase at constant internal energy.
isentropic_exponent()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.
isentropic_exponent_PT()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $P^{(1-k)}T^k = \text{const}$.
isentropic_exponent_PV()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.
isentropic_exponent_TV()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $TV^{k-1} = \text{const}$.
isobaric_expansion()	Method to calculate and return the isobatic expansion coefficient of the bulk according to the selected calculation methodology.
isothermal_bulk_modulus()	Method to calculate and return the isothermal bulk modulus of the phase.
k()	Calculate and return the thermal conductivity of the bulk according to the selected thermal conductivity settings in BulkSettings, the settings in ThermalConductivityGasMixture and ThermalConductivityLiquidMixture, and the configured pure-component settings in ThermalConductivityGas and ThermalConductivityLiquid.
kappa()	Method to calculate and return the isothermal compressibility of the bulk according to the selected calculation methodology.
log_zs()	Method to calculate and return the log of mole fractions specified.
molar_water_content([phase])	Method to calculate and return the molar water content; this is the g/mol of the fluid which is coming from water, [g/mol].

continues on next page

Table 65 – continued from previous page

mu()	Calculate and return the viscosity of the bulk according to the selected viscosity settings in BulkSettings, the settings in *ViscosityGasMixture* and *ViscosityLiquidMixture*, and the configured pure-component settings in *ViscosityGas* and *ViscosityLiquid*.
nu([phase])	Method to calculate and return the kinematic viscosity of the equilibrium state.
pseudo_Pc([phase])	Method to calculate and return the pseudocritical pressure calculated using Kay's rule (linear mole fractions):
pseudo_Tc([phase])	Method to calculate and return the pseudocritical temperature calculated using Kay's rule (linear mole fractions):
pseudo_Vc([phase])	Method to calculate and return the pseudocritical volume calculated using Kay's rule (linear mole fractions):
pseudo_Zc([phase])	Method to calculate and return the pseudocritical compressibility calculated using Kay's rule (linear mole fractions):
rho()	Method to calculate and return the molar density of the phase.
rho_mass([phase])	Method to calculate and return mass density of the phase.
rho_mass_liquid_ref([phase])	Method to calculate and return the liquid reference mass density according to the temperature variable *T_liquid_volume_ref* of *thermo.bulk.BulkSettings* and the composition of the phase.
sigma()	Calculate and return the surface tension of the bulk according to the selected surface tension settings in BulkSettings, the settings in *SurfaceTensionMixture* and the configured pure-component settings in *SurfaceTension*.
speed_of_sound()	Method to calculate and return the molar speed of sound of the bulk according to the selected calculation methodology.
speed_of_sound_mass()	Method to calculate and return the speed of sound of the phase.
value(name[, phase])	Method to retrieve a property from a string.
ws([phase])	Method to calculate and return the mass fractions of the phase, [-]
ws_no_water([phase])	Method to calculate and return the mass fractions of all species in the phase, normalized to a water-free basis (the mass fraction of water returned is zero).
zs_no_water([phase])	Method to calculate and return the mole fractions of all species in the phase, normalized to a water-free basis (the mole fraction of water returned is zero).

A()

Method to calculate and return the Helmholtz energy of the phase.

$$A = U - TS$$

Returns

A [float] Helmholtz energy, [J/mol]

`API`(*phase=None*)
Method to calculate and return the API of the phase.

$$\text{API gravity} = \frac{141.5}{\text{SG}} - 131.5$$

Returns

API [float] API of the fluid [-]

`A_dep`()
Method to calculate and return the departure Helmholtz energy of the phase.

$$A_{dep} = U_{dep} - TS_{dep}$$

Returns

A_dep [float] Departure Helmholtz energy, [J/mol]

`A_formation_ideal_gas`(*phase=None*)
Method to calculate and return the ideal-gas Helmholtz energy of formation of the phase (as if the phase was an ideal gas).

$$A_{reactive}^{ig} = U_{reactive}^{ig} - T_{ref}^{ig} S_{reactive}^{ig}$$

Returns

A_formation_ideal_gas [float] Helmholtz energy of formation of the phase on a reactive basis as an ideal gas, [J/(mol)]

`A_ideal_gas`(*phase=None*)
Method to calculate and return the ideal-gas Helmholtz energy of the phase.

$$A^{ig} = U^{ig} - TS^{ig}$$

Returns

A_ideal_gas [float] Ideal gas Helmholtz free energy, [J/(mol)]

`A_mass`(*phase=None*)
Method to calculate and return mass Helmholtz energy of the phase.

$$A_{mass} = \frac{1000 A_{molar}}{MW}$$

Returns

A_mass [float] Mass Helmholtz energy, [J/(kg)]

`A_reactive`()
Method to calculate and return the Helmholtz free energy of the phase on a reactive basis.

$$A_{reactive} = U_{reactive} - TS_{reactive}$$

Returns

A_reactive [float] Helmholtz free energy of the phase on a reactive basis, [J/(mol)]

`Bvirial`(*phase=None*)
Method to calculate and return the *B* virial coefficient of the phase at its current conditions.

Returns

Bvirial [float] Virial coefficient, [m^3/mol]

`property CASs`

CAS registration numbers for each component, [-].

Returns

CASs [list[str]] CAS registration numbers for each component, [-].

`property Carcinogens`

Status of each component in cancer causing registries, [-].

Returns

Carcinogens [list[dict]] Status of each component in cancer causing registries, [-].

`property Ceilings`

Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Returns

Ceilings [list[tuple[(float, str)]]] Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

`Cp()`

Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase. This is a phase-fraction weighted calculation.

$$C_p = \sum_i^p C_{p,i}\beta_i$$

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

`Cp_Cv_ratio()`

Method to calculate and return the Cp/Cv ratio of the phase.

$$\frac{C_p}{C_v}$$

Returns

Cp_Cv_ratio [float] Cp/Cv ratio, [-]

`Cp_Cv_ratio_ideal_gas(`*phase=None*`)`

Method to calculate and return the ratio of the ideal-gas heat capacity to its constant-volume heat capacity.

$$\frac{C_p^{ig}}{C_v^{ig}}$$

Returns

Cp_Cv_ratio_ideal_gas [float] Cp/Cv for the phase as an ideal gas, [-]

`Cp_dep(`*phase=None*`)`

Method to calculate and return the difference between the actual *Cp* and the ideal-gas heat capacity C_p^{ig} of the phase.

$$C_p^{dep} = C_p - C_p^{ig}$$

Returns

Cp_dep [float] Departure ideal gas heat capacity, [J/(mol*K)]

`Cp_ideal_gas`(*phase=None*)
Method to calculate and return the ideal-gas heat capacity of the phase.

$$C_p^{ig} = \sum_i z_i C_{p,i}^{ig}$$

Returns

Cp [float] Ideal gas heat capacity, [J/(mol*K)]

`Cp_mass`(*phase=None*)
Method to calculate and return mass constant pressure heat capacity of the phase.

$$Cp_{mass} = \frac{1000 Cp_{molar}}{MW}$$

Returns

Cp_mass [float] Mass heat capacity, [J/(kg*K)]

`Cv`()
Method to calculate and return the constant-volume heat capacity *Cv* of the phase.

$$C_v = T\left(\frac{\partial P}{\partial T}\right)_V^2 / \left(\frac{\partial P}{\partial V}\right)_T + Cp$$

Returns

Cv [float] Constant volume molar heat capacity, [J/(mol*K)]

`Cv_dep`(*phase=None*)
Method to calculate and return the difference between the actual *Cv* and the ideal-gas constant volume heat capacity C_v^{ig} of the phase.

$$C_v^{dep} = C_v - C_v^{ig}$$

Returns

Cv_dep [float] Departure ideal gas constant volume heat capacity, [J/(mol*K)]

`Cv_ideal_gas`(*phase=None*)
Method to calculate and return the ideal-gas constant volume heat capacity of the phase.

$$C_v^{ig} = \sum_i z_i C_{p,i}^{ig} - R$$

Returns

Cv [float] Ideal gas constant volume heat capacity, [J/(mol*K)]

`Cv_mass`(*phase=None*)
Method to calculate and return mass constant volume heat capacity of the phase.

$$Cv_{mass} = \frac{1000 Cv_{molar}}{MW}$$

Returns

Cv_mass [float] Mass constant volume heat capacity, [J/(kg*K)]

`property EnthalpySublimations`
Wrapper to obtain the list of EnthalpySublimations objects of the associated *PropertyCorrelationsPackage*.

property EnthalpyVaporizations
Wrapper to obtain the list of EnthalpyVaporizations objects of the associated *PropertyCorrelationsPackage*.

G()
Method to calculate and return the Gibbs free energy of the phase.

$$G = H - TS$$

Returns

G [float] Gibbs free energy, [J/mol]

property GWPs
Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

Returns

GWPs [list[float]] Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

G_dep()
Method to calculate and return the departure Gibbs free energy of the phase.

$$G_{dep} = H_{dep} - TS_{dep}$$

Returns

G_dep [float] Departure Gibbs free energy, [J/mol]

G_formation_ideal_gas(*phase=None*)
Method to calculate and return the ideal-gas Gibbs free energy of formation of the phase (as if the phase was an ideal gas).

$$G_{reactive}^{ig} = H_{reactive}^{ig} - T_{ref}^{ig} S_{reactive}^{ig}$$

Returns

G_formation_ideal_gas [float] Gibbs free energy of formation of the phase on a reactive basis as an ideal gas, [J/(mol)]

G_ideal_gas(*phase=None*)
Method to calculate and return the ideal-gas Gibbs free energy of the phase.

$$G^{ig} = H^{ig} - TS^{ig}$$

Returns

G_ideal_gas [float] Ideal gas free energy, [J/(mol)]

G_mass(*phase=None*)
Method to calculate and return mass Gibbs energy of the phase.

$$G_{mass} = \frac{1000 G_{molar}}{MW}$$

Returns

G_mass [float] Mass Gibbs energy, [J/(kg)]

G_reactive()
Method to calculate and return the Gibbs free energy of the phase on a reactive basis.

$$G_{reactive} = H_{reactive} - TS_{reactive}$$

Returns

G_reactive [float] Gibbs free energy of the phase on a reactive basis, [J/(mol)]

`property Gfgs`

Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

Returns

Gfgs [list[float]] Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

`property Gfgs_mass`

Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

Returns

Gfgs_mass [list[float]] Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

`H()`

Method to calculate and return the constant-temperature and constant phase-fraction enthalpy of the bulk phase. This is a phase-fraction weighted calculation.

$$H = \sum_i^p H_i \beta_i$$

Returns

H [float] Molar enthalpy, [J/(mol)]

`H_C_ratio(`*phase=None*`)`

Method to calculate and return the atomic ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.

Returns

H_C_ratio [float] H/C ratio on a molar basis, [-]

Notes

None is returned if no species are present that have carbon atoms.

`H_C_ratio_mass(`*phase=None*`)`

Method to calculate and return the mass ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.

Returns

H_C_ratio_mass [float] H/C ratio on a mass basis, [-]

Notes

None is returned if no species are present that have carbon atoms.

H_dep(*phase=None*)

Method to calculate and return the difference between the actual H and the ideal-gas enthalpy of the phase.

$$H^{dep} = H - H^{ig}$$

Returns

H_dep [float] Departure enthalpy, [J/(mol)]

H_formation_ideal_gas(*phase=None*)

Method to calculate and return the ideal-gas enthalpy of formation of the phase (as if the phase was an ideal gas).

$$H_{reactive}^{ig} = \sum_i z_i H_{f,i}$$

Returns

H_formation_ideal_gas [float] Enthalpy of formation of the phase on a reactive basis as an ideal gas, [J/mol]

H_ideal_gas(*phase=None*)

Method to calculate and return the ideal-gas enthalpy of the phase.

$$H^{ig} = \sum_i z_i H_i^{ig}$$

Returns

H [float] Ideal gas enthalpy, [J/(mol)]

H_mass(*phase=None*)

Method to calculate and return mass enthalpy of the phase.

$$H_{mass} = \frac{1000 H_{molar}}{MW}$$

Returns

H_mass [float] Mass enthalpy, [J/kg]

H_reactive()

Method to calculate and return the constant-temperature and constant phase-fraction reactive enthalpy of the bulk phase. This is a phase-fraction weighted calculation.

$$H_{\text{reactive}} = \sum_i^p H_{\text{reactive},i} \beta_i$$

Returns

H_reactive [float] Reactive molar enthalpy, [J/(mol)]

Hc(*phase=None*)

Method to calculate and return the molar ideal-gas higher heat of combustion of the object, [J/mol]

Returns

Hc [float] Molar higher heat of combustion, [J/(mol)]

`Hc_lower`(*phase=None*)
Method to calculate and return the molar ideal-gas lower heat of combustion of the object, [J/mol]

Returns

Hc_lower [float] Molar lower heat of combustion, [J/(mol)]

`Hc_lower_mass`(*phase=None*)
Method to calculate and return the mass ideal-gas lower heat of combustion of the object, [J/mol]

Returns

Hc_lower_mass [float] Mass lower heat of combustion, [J/(kg)]

`Hc_lower_normal`(*phase=None*)
Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the normal gas volume, [J/m^3]

Returns

Hc_lower_normal [float] Volumetric (normal) lower heat of combustion, [J/(m^3)]

`Hc_lower_standard`(*phase=None*)
Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the standard gas volume, [J/m^3]

Returns

Hc_lower_standard [float] Volumetric (standard) lower heat of combustion, [J/(m^3)]

`Hc_mass`(*phase=None*)
Method to calculate and return the mass ideal-gas higher heat of combustion of the object, [J/mol]

Returns

Hc_mass [float] Mass higher heat of combustion, [J/(kg)]

`Hc_normal`(*phase=None*)
Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the normal gas volume, [J/m^3]

Returns

Hc_normal [float] Volumetric (normal) higher heat of combustion, [J/(m^3)]

`Hc_standard`(*phase=None*)
Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the standard gas volume, [J/m^3]

Returns

Hc_normal [float] Volumetric (standard) higher heat of combustion, [J/(m^3)]

`property Hcs`
Higher standard molar heats of combustion for each component, [J/mol].

Returns

Hcs [list[float]] Higher standard molar heats of combustion for each component, [J/mol].

`property Hcs_lower`
Lower standard molar heats of combustion for each component, [J/mol].

Returns

Hcs_lower [list[float]] Lower standard molar heats of combustion for each component, [J/mol].

property Hcs_lower_mass
Lower standard heats of combustion for each component, [J/kg].

Returns

Hcs_lower_mass [list[float]] Lower standard heats of combustion for each component, [J/kg].

property Hcs_mass
Higher standard heats of combustion for each component, [J/kg].

Returns

Hcs_mass [list[float]] Higher standard heats of combustion for each component, [J/kg].

property HeatCapacityGasMixture
Wrapper to obtain the list of HeatCapacityGasMixture objects of the associated *PropertyCorrelationsPackage*.

property HeatCapacityGases
Wrapper to obtain the list of HeatCapacityGases objects of the associated *PropertyCorrelationsPackage*.

property HeatCapacityLiquidMixture
Wrapper to obtain the list of HeatCapacityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

property HeatCapacityLiquids
Wrapper to obtain the list of HeatCapacityLiquids objects of the associated *PropertyCorrelationsPackage*.

property HeatCapacitySolidMixture
Wrapper to obtain the list of HeatCapacitySolidMixture objects of the associated *PropertyCorrelationsPackage*.

property HeatCapacitySolids
Wrapper to obtain the list of HeatCapacitySolids objects of the associated *PropertyCorrelationsPackage*.

property Hf_STPs
Standard state molar enthalpies of formation for each component, [J/mol].

Returns

Hf_STPs [list[float]] Standard state molar enthalpies of formation for each component, [J/mol].

property Hf_STPs_mass
Standard state mass enthalpies of formation for each component, [J/kg].

Returns

Hf_STPs_mass [list[float]] Standard state mass enthalpies of formation for each component, [J/kg].

property Hfgs
Ideal gas standard molar enthalpies of formation for each component, [J/mol].

Returns

Hfgs [list[float]] Ideal gas standard molar enthalpies of formation for each component, [J/mol].

property Hfgs_mass
Ideal gas standard enthalpies of formation for each component, [J/kg].

Returns

Hfgs_mass [list[float]] Ideal gas standard enthalpies of formation for each component, [J/kg].

property Hfus_Tms
Molar heats of fusion for each component at their respective melting points, [J/mol].

Returns

Hfus_Tms [list[float]] Molar heats of fusion for each component at their respective melting points, [J/mol].

property Hfus_Tms_mass
Heats of fusion for each component at their respective melting points, [J/kg].

Returns

Hfus_Tms_mass [list[float]] Heats of fusion for each component at their respective melting points, [J/kg].

property Hsub_Tts
Heats of sublimation for each component at their respective triple points, [J/mol].

Returns

Hsub_Tts [list[float]] Heats of sublimation for each component at their respective triple points, [J/mol].

property Hsub_Tts_mass
Heats of sublimation for each component at their respective triple points, [J/kg].

Returns

Hsub_Tts_mass [list[float]] Heats of sublimation for each component at their respective triple points, [J/kg].

property Hvap_298s
Molar heats of vaporization for each component at 298.15 K, [J/mol].

Returns

Hvap_298s [list[float]] Molar heats of vaporization for each component at 298.15 K, [J/mol].

property Hvap_298s_mass
Heats of vaporization for each component at 298.15 K, [J/kg].

Returns

Hvap_298s_mass [list[float]] Heats of vaporization for each component at 298.15 K, [J/kg].

property Hvap_Tbs
Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

Returns

Hvap_Tbs [list[float]] Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

property Hvap_Tbs_mass
Heats of vaporization for each component at their respective normal boiling points, [J/kg].

Returns

Hvap_Tbs_mass [list[float]] Heats of vaporization for each component at their respective normal boiling points, [J/kg].

property IDs

Alias of CASs.

property InChI_Keys

InChI Keys for each component, [-].

Returns

InChI_Keys [list[str]] InChI Keys for each component, [-].

property InChIs

InChI strings for each component, [-].

Returns

InChIs [list[str]] InChI strings for each component, [-].

Joule_Thomson()

Method to calculate and return the Joule-Thomson coefficient of the bulk according to the selected calculation methodology.

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H$$

Returns

mu_JT [float] Joule-Thomson coefficient [K/Pa]

Ks(*phase*, *phase_ref=None*)

Method to calculate and return the K-values of each phase. These are NOT just liquid-vapor K values; these are thermodynamic K values. The reference phase can be specified with *phase_ref*, and then the K-values will be with respect to that phase.

$$K_i = \frac{z_{i,\text{phase}}}{z_{i,\text{ref phase}}}$$

If no reference phase is provided, the following criteria is used to select one:

- If the flash algorithm provided a reference phase, use that
- Otherwise use the liquid0 phase if one is present
- Otherwise use the solid0 phase if one is present
- Otherwise use the gas phase if one is present

Returns

Ks [list[float]] Equilibrium K values, [-]

property LF

Method to return the liquid fraction of the equilibrium state. If no liquid is present, 0 is always returned.

Returns

LF [float] Liquid molar fraction, [-]

property LFLs

Lower flammability limits for each component, [-].

Returns

LFLs [list[float]] Lower flammability limits for each component, [-].

`MW(phase=None)`
Method to calculate and return the molecular weight of the phase.

$$\text{MW} = \sum_i z_i \text{MW}_i$$

Returns

MW [float] Molecular weight of the phase, [g/mol]

property MWs
Similatiry variables for each component, [g/mol].

Returns

MWs [list[float]] Similatiry variables for each component, [g/mol].

property ODPs
Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

Returns

ODPs [list[float]] Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

`PIP()`
Method to calculate and return the phase identification parameter of the phase.

$$\Pi = V \left[\frac{\frac{\partial^2 P}{\partial V \partial T}}{\frac{\partial P}{\partial T}} - \frac{\frac{\partial^2 P}{\partial V^2}}{\frac{\partial P}{\partial V}}\right]$$

Returns

PIP [float] Phase identification parameter, [-]

property PSRK_groups
PSRK subgroup: count groups for each component, [-].

Returns

PSRK_groups [list[dict]] PSRK subgroup: count groups for each component, [-].

P_REF_IG = 101325.0

P_REF_IG_INV = 9.869232667160129e-06

property Parachors
Parachors for each component, [N^0.25*m^2.75/mol].

Returns

Parachors [list[float]] Parachors for each component, [N^0.25*m^2.75/mol].

property Pcs
Critical pressures for each component, [Pa].

Returns

Pcs [list[float]] Critical pressures for each component, [Pa].

property PermittivityLiquids
Wrapper to obtain the list of PermittivityLiquids objects of the associated *PropertyCorrelationsPackage*.

`Pmc`(*phase=None*)
Method to calculate and return the mechanical critical pressure of the phase.

Returns

Pmc [float] Mechanical critical pressure, [Pa]

property Psat_298s
Vapor pressures for each component at 298.15 K, [Pa].

Returns

Psat_298s [list[float]] Vapor pressures for each component at 298.15 K, [Pa].

property Pts
Triple point pressures for each component, [Pa].

Returns

Pts [list[float]] Triple point pressures for each component, [Pa].

property PubChems
Pubchem IDs for each component, [-].

Returns

PubChems [list[int]] Pubchem IDs for each component, [-].

property RI_Ts
Temperatures at which the refractive indexes were reported for each component, [K].

Returns

RI_Ts [list[float]] Temperatures at which the refractive indexes were reported for each component, [K].

property RIs
Refractive indexes for each component, [-].

Returns

RIs [list[float]] Refractive indexes for each component, [-].

`S()`
Method to calculate and return the constant-temperature and constant phase-fraction entropy of the bulk phase. This is a phase-fraction weighted calculation.

$$S = \sum_i^p S_i \beta_i$$

Returns

S [float] Molar entropy, [J/(mol*K)]

property S0gs
Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

Returns

S0gs [list[float]] Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

property S0gs_mass
Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

Returns

S0gs_mass [list[float]] Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

SG(*phase=None*)
Method to calculate and return the standard liquid specific gravity of the phase, using constant liquid pure component densities not calculated by the phase object, at 60 °F.

Returns

SG [float] Specific gravity of the liquid, [-]

Notes

The reference density of water is from the IAPWS-95 standard - 999.0170824078306 kg/m^3.

SG_gas(*phase=None*)
Method to calculate and return the specific gravity of the phase with respect to a gas reference density.

Returns

SG_gas [float] Specific gravity of the gas, [-]

Notes

The reference molecular weight of air used is 28.9586 g/mol.

property STELs
Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Returns

STELs [list[tuple[(float, str)]]] Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

S_dep(*phase=None*)
Method to calculate and return the difference between the actual S and the ideal-gas entropy of the phase.

$$S^{dep} = S - S^{ig}$$

Returns

S_dep [float] Departure entropy, [J/(mol*K)]

S_formation_ideal_gas(*phase=None*)
Method to calculate and return the ideal-gas entropy of formation of the phase (as if the phase was an ideal gas).

$$S_{reactive}^{ig} = \sum_i z_i S_{f,i}$$

Returns

S_formation_ideal_gas [float] Entropy of formation of the phase on a reactive basis as an ideal gas, [J/(mol*K)]

S_ideal_gas(*phase=None*)
Method to calculate and return the ideal-gas entropy of the phase.

$$S^{ig} = \sum_i z_i S_i^{ig} - R\ln\left(\frac{P}{P_{ref}}\right) - R\sum_i z_i \ln(z_i)$$

Returns

S [float] Ideal gas molar entropy, [J/(mol*K)]

`S_mass(phase=None)`

Method to calculate and return mass entropy of the phase.

$$S_{mass} = \frac{1000 S_{molar}}{MW}$$

Returns

S_mass [float] Mass enthalpy, [J/(kg*K)]

`S_reactive()`

Method to calculate and return the constant-temperature and constant phase-fraction reactive entropy of the bulk phase. This is a phase-fraction weighted calculation.

$$S_{\text{reactive}} = \sum_i^p S_{\text{reactive},i}\beta_i$$

Returns

S_reactive [float] Reactive molar entropy, [J/(mol*K)]

`property Sfgs`

Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

Returns

Sfgs [list[float]] Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

`property Sfgs_mass`

Ideal gas standard entropies of formation for each component, [J/(kg*K)].

Returns

Sfgs_mass [list[float]] Ideal gas standard entropies of formation for each component, [J/(kg*K)].

`property Skins`

Whether each compound can be absorbed through the skin or not, [-].

Returns

Skins [list[bool]] Whether each compound can be absorbed through the skin or not, [-].

`property StielPolars`

Stiel polar factors for each component, [-].

Returns

StielPolars [list[float]] Stiel polar factors for each component, [-].

`property Stockmayers`

Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

Returns

Stockmayers [list[float]] Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

property SublimationPressures
Wrapper to obtain the list of SublimationPressures objects of the associated *PropertyCorrelationsPackage*.

property SurfaceTensionMixture
Wrapper to obtain the list of SurfaceTensionMixture objects of the associated *PropertyCorrelationsPackage*.

property SurfaceTensions
Wrapper to obtain the list of SurfaceTensions objects of the associated *PropertyCorrelationsPackage*.

property TWAs
Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Returns

TWAs [list[tuple[(float, str)]]] Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

T_REF_IG = 298.15

T_REF_IG_INV = 0.0033540164346805303

property Tautoignitions
Autoignition temperatures for each component, [K].

Returns

Tautoignitions [list[float]] Autoignition temperatures for each component, [K].

property Tbs
Boiling temperatures for each component, [K].

Returns

Tbs [list[float]] Boiling temperatures for each component, [K].

property Tcs
Critical temperatures for each component, [K].

Returns

Tcs [list[float]] Critical temperatures for each component, [K].

property Tflashs
Flash point temperatures for each component, [K].

Returns

Tflashs [list[float]] Flash point temperatures for each component, [K].

property ThermalConductivityGasMixture
Wrapper to obtain the list of ThermalConductivityGasMixture objects of the associated *PropertyCorrelationsPackage*.

property ThermalConductivityGases
Wrapper to obtain the list of ThermalConductivityGases objects of the associated *PropertyCorrelationsPackage*.

property ThermalConductivityLiquidMixture
Wrapper to obtain the list of ThermalConductivityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

property ThermalConductivityLiquids
Wrapper to obtain the list of ThermalConductivityLiquids objects of the associated *PropertyCorrelationsPackage*.

Tmc(*phase=None*)
Method to calculate and return the mechanical critical temperature of the phase.

Returns

Tmc [float] Mechanical critical temperature, [K]

property Tms
Melting temperatures for each component, [K].

Returns

Tms [list[float]] Melting temperatures for each component, [K].

property Tts
Triple point temperatures for each component, [K].

Returns

Tts [list[float]] Triple point temperatures for each component, [K].

U()
Method to calculate and return the internal energy of the phase.

$$U = H - PV$$

Returns

U [float] Internal energy, [J/mol]

property UFLs
Upper flammability limits for each component, [-].

Returns

UFLs [list[float]] Upper flammability limits for each component, [-].

property UNIFAC_Dortmund_groups
UNIFAC_Dortmund_group: count groups for each component, [-].

Returns

UNIFAC_Dortmund_groups [list[dict]] UNIFAC_Dortmund_group: count groups for each component, [-].

property UNIFAC_Qs
UNIFAC Q parameters for each component, [-].

Returns

UNIFAC_Qs [list[float]] UNIFAC Q parameters for each component, [-].

property UNIFAC_Rs
UNIFAC R parameters for each component, [-].

Returns

UNIFAC_Rs [list[float]] UNIFAC R parameters for each component, [-].

property UNIFAC_groups
UNIFAC_group: count groups for each component, [-].

Returns

UNIFAC_groups [list[dict]] UNIFAC_group: count groups for each component, [-].

`U_dep()`

Method to calculate and return the departure internal energy of the phase.

$$U_{dep} = H_{dep} - PV_{dep}$$

Returns

U_dep [float] Departure internal energy, [J/mol]

`U_formation_ideal_gas(`*phase=None*`)`

Method to calculate and return the ideal-gas internal energy of formation of the phase (as if the phase was an ideal gas).

$$U_{reactive}^{ig} = H_{reactive}^{ig} - P_{ref}^{ig} V^{ig}$$

Returns

U_formation_ideal_gas [float] Internal energy of formation of the phase on a reactive basis as an ideal gas, [J/(mol)]

`U_ideal_gas(`*phase=None*`)`

Method to calculate and return the ideal-gas internal energy of the phase.

$$U^{ig} = H^{ig} - PV^{ig}$$

Returns

U_ideal_gas [float] Ideal gas internal energy, [J/(mol)]

`U_mass(`*phase=None*`)`

Method to calculate and return mass internal energy of the phase.

$$U_{mass} = \frac{1000 U_{molar}}{MW}$$

Returns

U_mass [float] Mass internal energy, [J/(kg)]

`U_reactive()`

Method to calculate and return the internal energy of the phase on a reactive basis.

$$U_{reactive} = H_{reactive} - PV$$

Returns

U_reactive [float] Internal energy of the phase on a reactive basis, [J/(mol)]

`V()`

Method to calculate and return the molar volume of the bulk phase. This is a phase-fraction weighted calculation.

$$V = \sum_i^p V_i \beta_i$$

Returns

V [float] Molar volume, [m^3/mol]

`property VF`

Method to return the vapor fraction of the equilibrium state. If no vapor/gas is present, 0 is always returned.

Returns

VF [float] Vapor molar fraction, [-]

V_dep()

Method to calculate and return the departure (from ideal gas behavior) molar volume of the phase.

$$V_{dep} = V - \frac{RT}{P}$$

Returns

V_dep [float] Departure molar volume, [m^3/mol]

V_gas(*phase=None*)

Method to calculate and return the ideal-gas molar volume of the phase at the chosen reference temperature and pressure, according to the temperature variable *T_gas_ref* and pressure variable *P_gas_ref* of the `thermo.bulk.BulkSettings`.

$$V^{ig} = \frac{RT_{ref}}{P_{ref}}$$

Returns

V_gas [float] Ideal gas molar volume at the reference temperature and pressure, [m^3/mol]

V_gas_normal(*phase=None*)

Method to calculate and return the ideal-gas molar volume of the phase at the normal temperature and pressure, according to the temperature variable *T_normal* and pressure variable *P_normal* of the `thermo.bulk.BulkSettings`.

$$V^{ig} = \frac{RT_{norm}}{P_{norm}}$$

Returns

V_gas_normal [float] Ideal gas molar volume at normal temperature and pressure, [m^3/mol]

V_gas_standard(*phase=None*)

Method to calculate and return the ideal-gas molar volume of the phase at the standard temperature and pressure, according to the temperature variable *T_standard* and pressure variable *P_standard* of the `thermo.bulk.BulkSettings`.

$$V^{ig} = \frac{RT_{std}}{P_{std}}$$

Returns

V_gas_standard [float] Ideal gas molar volume at standard temperature and pressure, [m^3/mol]

V_ideal_gas(*phase=None*)

Method to calculate and return the ideal-gas molar volume of the phase.

$$V^{ig} = \frac{RT}{P}$$

Returns

V [float] Ideal gas molar volume, [m^3/mol]

V_iter(*phase=None*, *force=False*)

Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure. This often means the return value of this method is an mpmath *mpf*. This dummy method simply returns the implemented V method.

Returns

V [float or mpf] Molar volume, [m^3/mol]

V_liquid_ref(*phase=None*)

Method to calculate and return the liquid reference molar volume according to the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings` and the composition of the phase.

$$V = \sum_i z_i V_i$$

Returns

V_liquid_ref [float] Liquid molar volume at the reference condition, [m^3/mol]

V_liquids_ref()

Method to calculate and return the liquid reference molar volumes according to the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings`.

Returns

V_liquids_ref [list[float]] Liquid molar volumes at the reference condition, [m^3/mol]

V_mass(*phase=None*)

Method to calculate and return the specific volume of the phase.

$$V_{mass} = \frac{1000 \cdot VM}{MW}$$

Returns

V_mass [float] Specific volume of the phase, [m^3/kg]

property Van_der_Waals_areas

Unnormalized Van der Waals areas for each component, [m^2/mol].

Returns

Van_der_Waals_areas [list[float]] Unnormalized Van der Waals areas for each component, [m^2/mol].

property Van_der_Waals_volumes

Unnormalized Van der Waals volumes for each component, [m^3/mol].

Returns

Van_der_Waals_volumes [list[float]] Unnormalized Van der Waals volumes for each component, [m^3/mol].

property VaporPressures

Wrapper to obtain the list of VaporPressures objects of the associated `PropertyCorrelationsPackage`.

property Vcs

Critical molar volumes for each component, [m^3/mol].

Returns

Vcs [list[float]] Critical molar volumes for each component, [m^3/mol].

Vfgs(*phase=None*)
Method to calculate and return the ideal-gas volume fractions of the components of the phase. This is the same as the mole fractions.

Returns

Vfgs [list[float]] Ideal-gas volume fractions of the components of the phase, [-]

Vfls(*phase=None*)
Method to calculate and return the ideal-liquid volume fractions of the components of the phase, using the standard liquid densities at the temperature variable *T_liquid_volume_ref* of *thermo.bulk.BulkSettings* and the composition of the phase.

Returns

Vfls [list[float]] Ideal-liquid volume fractions of the components of the phase, [-]

property ViscosityGasMixture
Wrapper to obtain the list of ViscosityGasMixture objects of the associated *PropertyCorrelationsPackage*.

property ViscosityGases
Wrapper to obtain the list of ViscosityGases objects of the associated *PropertyCorrelationsPackage*.

property ViscosityLiquidMixture
Wrapper to obtain the list of ViscosityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

property ViscosityLiquids
Wrapper to obtain the list of ViscosityLiquids objects of the associated *PropertyCorrelationsPackage*.

Vmc(*phase=None*)
Method to calculate and return the mechanical critical volume of the phase.

Returns

Vmc [float] Mechanical critical volume, [m^3/mol]

property Vmg_STPs
Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

Returns

Vmg_STPs [list[float]] Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

property Vml_60Fs
Liquid molar volumes for each component at 60 °F, [m^3/mol].

Returns

Vml_60Fs [list[float]] Liquid molar volumes for each component at 60 °F, [m^3/mol].

property Vml_STPs
Liquid molar volumes for each component at STP, [m^3/mol].

Returns

Vml_STPs [list[float]] Liquid molar volumes for each component at STP, [m^3/mol].

property Vml_Tms
Liquid molar volumes for each component at their respective melting points, [m^3/mol].

Returns

Vml_Tms [list[float]] Liquid molar volumes for each component at their respective melting points, [m^3/mol].

property Vms_Tms
Solid molar volumes for each component at their respective melting points, [m^3/mol].

Returns

Vms_Tms [list[float]] Solid molar volumes for each component at their respective melting points, [m^3/mol].

property VolumeGasMixture
Wrapper to obtain the list of VolumeGasMixture objects of the associated *PropertyCorrelationsPackage*.

property VolumeGases
Wrapper to obtain the list of VolumeGases objects of the associated *PropertyCorrelationsPackage*.

property VolumeLiquidMixture
Wrapper to obtain the list of VolumeLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

property VolumeLiquids
Wrapper to obtain the list of VolumeLiquids objects of the associated *PropertyCorrelationsPackage*.

property VolumeSolidMixture
Wrapper to obtain the list of VolumeSolidMixture objects of the associated *PropertyCorrelationsPackage*.

property VolumeSolids
Wrapper to obtain the list of VolumeSolids objects of the associated *PropertyCorrelationsPackage*.

Wobbe_index(*phase=None*)
Method to calculate and return the molar Wobbe index of the object, [J/mol].

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index [float] Molar Wobbe index, [J/(mol)]

Wobbe_index_lower(*phase=None*)

Method to calculate and return the molar lower Wobbe index of the object, [J/mol].

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower [float] Molar lower Wobbe index, [J/(mol)]

Wobbe_index_lower_mass(*phase=None*)
Method to calculate and return the lower mass Wobbe index of the object, [J/kg].

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower_mass [float] Mass lower Wobbe index, [J/(kg)]

Wobbe_index_lower_normal(*phase=None*)

Method to calculate and return the volumetric normal lower Wobbe index of the object, [J/m^3]. The normal gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower_normal [float] Volumetric normal lower Wobbe index, [J/(m^3)]

Wobbe_index_lower_standard(*phase=None*)

Method to calculate and return the volumetric standard lower Wobbe index of the object, [J/m^3]. The standard gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower_standard [float] Volumetric standard lower Wobbe index, [J/(m^3)]

Wobbe_index_mass(*phase=None*)

Method to calculate and return the mass Wobbe index of the object, [J/kg].

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_mass [float] Mass Wobbe index, [J/(kg)]

Wobbe_index_normal(*phase=None*)

Method to calculate and return the volumetric normal Wobbe index of the object, [J/m^3]. The normal gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index [float] Volumetric normal Wobbe index, [J/(m^3)]

Wobbe_index_standard(*phase=None*)

Method to calculate and return the volumetric standard Wobbe index of the object, [J/m^3]. The standard gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_standard [float] Volumetric standard Wobbe index, [J/(m^3)]

Z()

Method to calculate and return the compressibility factor of the phase.

$$Z = \frac{PV}{RT}$$

Returns

Z [float] Compressibility factor, [-] `property Zcs`

Critical compressibilities for each component, [-].

Returns

Zcs [list[float]] Critical compressibilities for each component, [-].

`Zmc`(*phase=None*)

Method to calculate and return the mechanical critical compressibility of the phase.

Returns

Zmc [float] Mechanical critical compressibility, [-]

`alpha`(*phase=None*)

Method to calculate and return the thermal diffusivity of the equilibrium state.

$$\alpha = \frac{k}{\rho Cp}$$

Returns

alpha [float] Thermal diffusivity, [m^2/s]

`atom_fractions`(*phase=None*)

Method to calculate and return the atomic composition of the phase; returns a dictionary of atom fraction (by count), containing only those elements who are present.

Returns

atom_fractions [dict[str: float]] Atom fractions, [-]

`atom_mass_fractions`(*phase=None*)

Method to calculate and return the atomic mass fractions of the phase; returns a dictionary of atom fraction (by mass), containing only those elements who are present.

Returns

atom_mass_fractions [dict[str: float]] Atom mass fractions, [-]

`property atomss`

Breakdown of each component into its elements and their counts, as a dict, [-].

Returns

atomss [list[dict]] Breakdown of each component into its elements and their counts, as a dict, [-].

`property betas_liquids`

Method to calculate and return the fraction of the liquid phase that each liquid phase is, by molar phase fraction. If the system is VLLL with phase fractions of 0.125 vapor, and [.25, .125, .5] for the three liquids phases respectively, the return value would be [0.28571428, 0.142857142, 0.57142857].

Returns

betas_liquids [list[float]] Molar phase fractions of the overall liquid phase, [-]

`property betas_mass`

Method to calculate and return the mass fraction of all of the phases in the system.

Returns

betas_mass [list[float]] Mass phase fractions of all the phases, ordered vapor, liquid, then solid , [-]

property betas_mass_liquids
Method to calculate and return the fraction of the liquid phase that each liquid phase is, by mass phase fraction. If the system is VLLL with mass phase fractions of 0.125 vapor, and [.25, .125, .5] for the three liquids phases respectively, the return value would be [0.28571428, 0.142857142, 0.57142857].

Returns

betas_mass_liquids [list[float]] Mass phase fractions of the overall liquid phase, [-]

property betas_mass_states
Method to return the mass phase fractions of each of the three fundamental *types* of phases.

Returns

betas_mass_states [list[float, 3]] List containing the mass phase fraction of gas, liquid, and solid, [-]

property betas_states
Method to return the molar phase fractions of each of the three fundamental *types* of phases.

Returns

betas_states [list[float, 3]] List containing the molar phase fraction of gas, liquid, and solid, [-]

property betas_volume
Method to calculate and return the volume fraction of all of the phases in the system.

Returns

betas_volume [list[float]] Volume phase fractions of all the phases, ordered vapor, liquid, then solid , [-]

property betas_volume_liquids
Method to calculate and return the fraction of the liquid phase that each liquid phase is, by volume phase fraction. If the system is VLLL with volume phase fractions of 0.125 vapor, and [.25, .125, .5] for the three liquids phases respectively, the return value would be [0.28571428, 0.142857142, 0.57142857].

Returns

betas_volume_liquids [list[float]] Volume phase fractions of the overall liquid phase, [-]

property betas_volume_states
Method to return the volume phase fractions of each of the three fundamental *types* of phases.

Returns

betas_volume_states [list[float, 3]] List containing the volume phase fraction of gas, liquid, and solid, [-]

property charges
Charge number (valence) for each component, [-].

Returns

charges [list[float]] Charge number (valence) for each component, [-].

property conductivities
Electrical conductivities for each component, [S/m].

Returns

conductivities [list[float]] Electrical conductivities for each component, [S/m].

property conductivity_Ts
Temperatures at which the electrical conductivities for each component were measured, [K].

Returns

conductivity_Ts [list[float]] Temperatures at which the electrical conductivities for each component were measured, [K].

`d2P_dT2()`

Method to calculate and return the second temperature derivative of pressure of the bulk according to the selected calculation methodology.

Returns

d2P_dT2 [float] Second temperature derivative of pressure, [Pa/K^2]

`d2P_dT2_frozen()`

Method to calculate and return the second constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial^2 P}{\partial T^2}\right)_{V,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial^2 P}{\partial T^2}\right)_{i,V_i,\beta_i,zs_i}$$

Returns

d2P_dT2_frozen [float] Frozen constant-volume second derivative of pressure with respect to temperature of the bulk phase, [Pa/K^2]

`d2P_dTdV()`

Method to calculate and return the second derivative of pressure with respect to temperature and volume of the bulk according to the selected calculation methodology.

Returns

d2P_dTdV [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

`d2P_dTdV_frozen()`

Method to calculate and return the second derivative of pressure with respect to volume and temperature of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial^2 P}{\partial V \partial T}\right)_{\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial^2 P}{\partial V \partial T}\right)_{i,\beta_i,zs_i}$$

Returns

d2P_dTdV_frozen [float] Frozen second derivative of pressure with respect to volume and temperature of the bulk phase, [Pa*mol^2/m^6]

`d2P_dV2()`

Method to calculate and return the second volume derivative of pressure of the bulk according to the selected calculation methodology.

Returns

d2P_dV2 [float] Second volume derivative of pressure, [Pa*mol^2/m^6]

`d2P_dV2_frozen()`

Method to calculate and return the constant-temperature second derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial^2 P}{\partial V^2}\right)_{T,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial^2 P}{\partial V^2}\right)_{i,T,\beta_i,zs_i}$$

Returns

d2P_dV2_frozen [float] Frozen constant-temperature second derivative of pressure with respect to volume of the bulk phase, [Pa*mol^2/m^6]

dA_dP()

Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial U}{\partial P}\right)_T$$

Returns

dA_dP [float] Constant-temperature pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dP_T()

Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial U}{\partial P}\right)_T$$

Returns

dA_dP [float] Constant-temperature pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dP_V()

Method to calculate and return the constant-volume pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_V = \left(\frac{\partial H}{\partial P}\right)_V - V - S\left(\frac{\partial T}{\partial P}\right)_V - T\left(\frac{\partial S}{\partial P}\right)_V$$

Returns

dA_dP_V [float] Constant-volume pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dT()

Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial U}{\partial T}\right)_P$$

Returns

dA_dT [float] Constant-pressure temperature derivative of Helmholtz energy, [J/(mol*K)]

dA_dT_P()

Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial U}{\partial T}\right)_P$$

Returns

dA_dT [float] Constant-pressure temperature derivative of Helmholtz energy, [J/(mol*K)]

dA_dT_V()

Method to calculate and return the constant-volume temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_V = \left(\frac{\partial H}{\partial T}\right)_V - V\left(\frac{\partial P}{\partial T}\right)_V - T\left(\frac{\partial S}{\partial T}\right)_V - S$$

Returns

dA_dT_V [float] Constant-volume temperature derivative of Helmholtz energy, [J/(mol*K)]

dA_dV_P()

Method to calculate and return the constant-pressure volume derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial V}\right)_P = \left(\frac{\partial A}{\partial T}\right)_P \left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dA_dV_P [float] Constant-pressure volume derivative of Helmholtz energy, [J/(m^3)]

dA_dV_T()

Method to calculate and return the constant-temperature volume derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial V}\right)_T = \left(\frac{\partial A}{\partial P}\right)_T \left(\frac{\partial P}{\partial V}\right)_T$$

Returns

dA_dV_T [float] Constant-temperature volume derivative of Helmholtz energy, [J/(m^3)]

dA_mass_dP()

Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.

$$\left(\frac{\partial A_{\text{mass}}}{\partial P}\right)_T$$

Returns

dA_mass_dP [float] The pressure derivative of mass Helmholtz energy of the phase at constant temperature, [J/mol/Pa]

dA_mass_dP_T()

Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.

$$\left(\frac{\partial A_{\text{mass}}}{\partial P}\right)_T$$

Returns

dA_mass_dP_T [float] The pressure derivative of mass Helmholtz energy of the phase at constant temperature, [J/mol/Pa]

dA_mass_dP_V()

Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant volume.

$$\left(\frac{\partial A_{\text{mass}}}{\partial P}\right)_V$$

Returns

dA_mass_dP_V [float] The pressure derivative of mass Helmholtz energy of the phase at constant volume, [J/mol/Pa]

`dA_mass_dT()`

Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.

$$\left(\frac{\partial A_{\text{mass}}}{\partial T}\right)_P$$

Returns

dA_mass_dT [float] The temperature derivative of mass Helmholtz energy of the phase at constant pressure, [J/mol/K]

`dA_mass_dT_P()`

Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.

$$\left(\frac{\partial A_{\text{mass}}}{\partial T}\right)_P$$

Returns

dA_mass_dT_P [float] The temperature derivative of mass Helmholtz energy of the phase at constant pressure, [J/mol/K]

`dA_mass_dT_V()`

Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant volume.

$$\left(\frac{\partial A_{\text{mass}}}{\partial T}\right)_V$$

Returns

dA_mass_dT_V [float] The temperature derivative of mass Helmholtz energy of the phase at constant volume, [J/mol/K]

`dA_mass_dV_P()`

Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant pressure.

$$\left(\frac{\partial A_{\text{mass}}}{\partial V}\right)_P$$

Returns

dA_mass_dV_P [float] The volume derivative of mass Helmholtz energy of the phase at constant pressure, [J/mol/m^3/mol]

dA_mass_dV_T()

Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant temperature.

$$\left(\frac{\partial A_{\text{mass}}}{\partial V}\right)_T$$

Returns

dA_mass_dV_T [float] The volume derivative of mass Helmholtz energy of the phase at constant temperature, [J/mol/m^3/mol]

dCv_dP_T()

Method to calculate the pressure derivative of Cv, constant volume heat capacity, at constant temperature.

$$\left(\frac{\partial C_v}{\partial P}\right)_T = -T\,\mathrm{dPdT_V}(P)\frac{d}{dP}\,\mathrm{dVdT_P}(P) - T\,\mathrm{dVdT_P}(P)\frac{d}{dP}\,\mathrm{dPdT_V}(P) + \frac{d}{dP}\,\mathrm{Cp}(P)$$

Returns

dCv_dP_T [float] Pressure derivative of constant volume heat capacity at constant temperature, [J/mol/K/Pa]

Notes

Requires *d2V_dTdP*, *d2P_dTdP*, and *d2H_dTdP*.

dCv_dT_P()

Method to calculate the temperature derivative of Cv, constant volume heat capacity, at constant pressure.

$$\left(\frac{\partial C_v}{\partial T}\right)_P = -\frac{T\,\mathrm{dPdT_V}^2(T)\frac{d}{dT}\,\mathrm{dPdV_T}(T)}{\mathrm{dPdV_T}^2(T)} + \frac{2T\,\mathrm{dPdT_V}(T)\frac{d}{dT}\,\mathrm{dPdT_V}(T)}{\mathrm{dPdV_T}(T)} + \frac{\mathrm{dPdT_V}^2(T)}{\mathrm{dPdV_T}(T)} + \frac{d}{dT}\,\mathrm{Cp}(T)$$

Returns

dCv_dT_P [float] Temperature derivative of constant volume heat capacity at constant pressure, [J/mol/K^2]

Notes

Requires *d2P_dT2_PV*, *d2P_dVdT_TP*, and *d2H_dT2*.

dCv_mass_dP_T()

Method to calculate and return the pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature.

$$\left(\frac{\partial Cv_{\text{mass}}}{\partial P}\right)_T$$

Returns

dCv_mass_dP_T [float] The pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature, [J/(mol*K)/Pa]

dCv_mass_dT_P()

Method to calculate and return the temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure.

$$\left(\frac{\partial Cv_{\text{mass}}}{\partial T}\right)_P$$

Returns

dCv_mass_dT_P [float] The temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure, [J/(mol*K)/K]

dG_dP()

Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dG_dP [float] Constant-temperature pressure derivative of Gibbs free energy, [J/(mol*Pa)]

dG_dP_T()

Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dG_dP [float] Constant-temperature pressure derivative of Gibbs free energy, [J/(mol*Pa)]

dG_dP_V()

Method to calculate and return the constant-volume pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_V = -T\left(\frac{\partial S}{\partial P}\right)_V - S\left(\frac{\partial T}{\partial P}\right)_V + \left(\frac{\partial H}{\partial P}\right)_V$$

Returns

dG_dP_V [float] Constant-volume pressure derivative of Gibbs free energy, [J/(mol*Pa)]

dG_dT()

Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dG_dT [float] Constant-pressure temperature derivative of Gibbs free energy, [J/(mol*K)]

dG_dT_P()

Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dG_dT [float] Constant-pressure temperature derivative of Gibbs free energy, [J/(mol*K)]

`dG_dT_V()`

Method to calculate and return the constant-volume temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_V = -T\left(\frac{\partial S}{\partial T}\right)_V - S + \left(\frac{\partial H}{\partial T}\right)_V$$

Returns

dG_dT_V [float] Constant-volume temperature derivative of Gibbs free energy, [J/(mol*K)]

`dG_dV_P()`

Method to calculate and return the constant-pressure volume derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial V}\right)_P = \left(\frac{\partial G}{\partial T}\right)_P \left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dG_dV_P [float] Constant-pressure volume derivative of Gibbs free energy, [J/(m^3)]

`dG_dV_T()`

Method to calculate and return the constant-temperature volume derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial V}\right)_T = \left(\frac{\partial G}{\partial P}\right)_T \left(\frac{\partial P}{\partial V}\right)_T$$

Returns

dG_dV_T [float] Constant-temperature volume derivative of Gibbs free energy, [J/(m^3)]

`dG_mass_dP()`

Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.

$$\left(\frac{\partial G_{\text{mass}}}{\partial P}\right)_T$$

Returns

dG_mass_dP [float] The pressure derivative of mass Gibbs free energy of the phase at constant temperature, [J/mol/Pa]

`dG_mass_dP_T()`

Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.

$$\left(\frac{\partial G_{\text{mass}}}{\partial P}\right)_T$$

Returns

dG_mass_dP_T [float] The pressure derivative of mass Gibbs free energy of the phase at constant temperature, [J/mol/Pa]

`dG_mass_dP_V()`

Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant volume.

$$\left(\frac{\partial G_{\text{mass}}}{\partial P}\right)_{V}$$

Returns

dG_mass_dP_V [float] The pressure derivative of mass Gibbs free energy of the phase at constant volume, [J/mol/Pa]

`dG_mass_dT()`

Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.

$$\left(\frac{\partial G_{\text{mass}}}{\partial T}\right)_{P}$$

Returns

dG_mass_dT [float] The temperature derivative of mass Gibbs free energy of the phase at constant pressure, [J/mol/K]

`dG_mass_dT_P()`

Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.

$$\left(\frac{\partial G_{\text{mass}}}{\partial T}\right)_{P}$$

Returns

dG_mass_dT_P [float] The temperature derivative of mass Gibbs free energy of the phase at constant pressure, [J/mol/K]

`dG_mass_dT_V()`

Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant volume.

$$\left(\frac{\partial G_{\text{mass}}}{\partial T}\right)_{V}$$

Returns

dG_mass_dT_V [float] The temperature derivative of mass Gibbs free energy of the phase at constant volume, [J/mol/K]

`dG_mass_dV_P()`

Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant pressure.

$$\left(\frac{\partial G_{\text{mass}}}{\partial V}\right)_{P}$$

Returns

dG_mass_dV_P [float] The volume derivative of mass Gibbs free energy of the phase at constant pressure, [J/mol/m^3/mol]

`dG_mass_dV_T()`

Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant temperature.

$$\left(\frac{\partial G_{\text{mass}}}{\partial V}\right)_T$$

Returns

dG_mass_dV_T [float] The volume derivative of mass Gibbs free energy of the phase at constant temperature, [J/mol/m^3/mol]

`dH_dP()`

Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.

Returns

dH_dP_T [float] Pressure derivative of enthalpy, [J/(mol*Pa)]

`dH_dP_T()`

Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.

Returns

dH_dP_T [float] Pressure derivative of enthalpy, [J/(mol*Pa)]

`dH_dT()`

Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase. This is a phase-fraction weighted calculation.

$$C_p = \sum_i^p C_{p,i}\beta_i$$

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

`dH_dT_P()`

Method to calculate and return the temperature derivative of enthalpy of the phase at constant pressure.

Returns

dH_dT_P [float] Temperature derivative of enthalpy, [J/(mol*K)]

`dH_mass_dP()`

Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.

$$\left(\frac{\partial H_{\text{mass}}}{\partial P}\right)_T$$

Returns

dH_mass_dP [float] The pressure derivative of mass enthalpy of the phase at constant temperature, [J/mol/Pa]

dH_mass_dP_T()

Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.

$$\left(\frac{\partial H_{\text{mass}}}{\partial P}\right)_T$$

Returns

dH_mass_dP_T [float] The pressure derivative of mass enthalpy of the phase at constant temperature, [J/mol/Pa]

dH_mass_dP_V()

Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant volume.

$$\left(\frac{\partial H_{\text{mass}}}{\partial P}\right)_V$$

Returns

dH_mass_dP_V [float] The pressure derivative of mass enthalpy of the phase at constant volume, [J/mol/Pa]

dH_mass_dT()

Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.

$$\left(\frac{\partial H_{\text{mass}}}{\partial T}\right)_P$$

Returns

dH_mass_dT [float] The temperature derivative of mass enthalpy of the phase at constant pressure, [J/mol/K]

dH_mass_dT_P()

Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.

$$\left(\frac{\partial H_{\text{mass}}}{\partial T}\right)_P$$

Returns

dH_mass_dT_P [float] The temperature derivative of mass enthalpy of the phase at constant pressure, [J/mol/K]

dH_mass_dT_V()

Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant volume.

$$\left(\frac{\partial H_{\text{mass}}}{\partial T}\right)_V$$

Returns

dH_mass_dT_V [float] The temperature derivative of mass enthalpy of the phase at constant volume, [J/mol/K]

`dH_mass_dV_P()`

Method to calculate and return the volume derivative of mass enthalpy of the phase at constant pressure.

$$\left(\frac{\partial H_{\text{mass}}}{\partial V}\right)_P$$

Returns

dH_mass_dV_P [float] The volume derivative of mass enthalpy of the phase at constant pressure, [J/mol/m^3/mol]

`dH_mass_dV_T()`

Method to calculate and return the volume derivative of mass enthalpy of the phase at constant temperature.

$$\left(\frac{\partial H_{\text{mass}}}{\partial V}\right)_T$$

Returns

dH_mass_dV_T [float] The volume derivative of mass enthalpy of the phase at constant temperature, [J/mol/m^3/mol]

`dP_dP_A()`

Method to calculate and return the pressure derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial P}\right)_A$$

Returns

dP_dP_A [float] The pressure derivative of pressure of the phase at constant Helmholtz energy, [Pa/Pa]

`dP_dP_G()`

Method to calculate and return the pressure derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial P}\right)_G$$

Returns

dP_dP_G [float] The pressure derivative of pressure of the phase at constant Gibbs energy, [Pa/Pa]

`dP_dP_H()`

Method to calculate and return the pressure derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial P}\right)_H$$

Returns

dP_dP_H [float] The pressure derivative of pressure of the phase at constant enthalpy, [Pa/Pa]

`dP_dP_S()`

Method to calculate and return the pressure derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial P}\right)_S$$

Returns

dP_dP_S [float] The pressure derivative of pressure of the phase at constant entropy, [Pa/Pa]

`dP_dP_U()`

Method to calculate and return the pressure derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial P}\right)_U$$

Returns

dP_dP_U [float] The pressure derivative of pressure of the phase at constant internal energy, [Pa/Pa]

`dP_dT()`

Method to calculate and return the first temperature derivative of pressure of the bulk according to the selected calculation methodology.

Returns

dP_dT [float] First temperature derivative of pressure, [Pa/K]

`dP_dT_A()`

Method to calculate and return the temperature derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial T}\right)_A$$

Returns

dP_dT_A [float] The temperature derivative of pressure of the phase at constant Helmholtz energy, [Pa/K]

`dP_dT_G()`

Method to calculate and return the temperature derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial T}\right)_G$$

Returns

dP_dT_G [float] The temperature derivative of pressure of the phase at constant Gibbs energy, [Pa/K]

`dP_dT_H()`
Method to calculate and return the temperature derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial T}\right)_H$$

Returns

dP_dT_H [float] The temperature derivative of pressure of the phase at constant enthalpy, [Pa/K]

`dP_dT_S()`
Method to calculate and return the temperature derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial T}\right)_S$$

Returns

dP_dT_S [float] The temperature derivative of pressure of the phase at constant entropy, [Pa/K]

`dP_dT_U()`
Method to calculate and return the temperature derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial T}\right)_U$$

Returns

dP_dT_U [float] The temperature derivative of pressure of the phase at constant internal energy, [Pa/K]

`dP_dT_frozen()`
Method to calculate and return the constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial P}{\partial T}\right)_{V,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial P}{\partial T}\right)_{i,V_i,\beta_i,zs_i}$$

Returns

dP_dT_frozen [float] Frozen constant-volume derivative of pressure with respect to temperature of the bulk phase, [Pa/K]

`dP_dV()`
Method to calculate and return the first volume derivative of pressure of the bulk according to the selected calculation methodology.

Returns

dP_dV [float] First volume derivative of pressure, [Pa*mol/m^3]

dP_dV_A()

Method to calculate and return the volume derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial V}\right)_A$$

Returns

dP_dV_A [float] The volume derivative of pressure of the phase at constant Helmholtz energy, [Pa/m^3/mol]

dP_dV_G()

Method to calculate and return the volume derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial V}\right)_G$$

Returns

dP_dV_G [float] The volume derivative of pressure of the phase at constant Gibbs energy, [Pa/m^3/mol]

dP_dV_H()

Method to calculate and return the volume derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial V}\right)_H$$

Returns

dP_dV_H [float] The volume derivative of pressure of the phase at constant enthalpy, [Pa/m^3/mol]

dP_dV_S()

Method to calculate and return the volume derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial V}\right)_S$$

Returns

dP_dV_S [float] The volume derivative of pressure of the phase at constant entropy, [Pa/m^3/mol]

dP_dV_U()

Method to calculate and return the volume derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial V}\right)_U$$

Returns

dP_dV_U [float] The volume derivative of pressure of the phase at constant internal energy, [Pa/m^3/mol]

dP_dV_frozen()

Method to calculate and return the constant-temperature derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions. This is a molar phase-fraction weighted calculation.

$$\left(\frac{\partial P}{\partial V}\right)_{T,\beta,zs} = \sum_i^{\text{phases}} \beta_i \left(\frac{\partial P}{\partial V}\right)_{i,T,\beta_i,zs_i}$$

Returns

dP_dV_frozen [float] Frozen constant-temperature derivative of pressure with respect to volume of the bulk phase, [Pa*mol/m^3]

dP_drho_A()

Method to calculate and return the density derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial \rho}\right)_A$$

Returns

dP_drho_A [float] The density derivative of pressure of the phase at constant Helmholtz energy, [Pa/mol/m^3]

dP_drho_G()

Method to calculate and return the density derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial \rho}\right)_G$$

Returns

dP_drho_G [float] The density derivative of pressure of the phase at constant Gibbs energy, [Pa/mol/m^3]

dP_drho_H()

Method to calculate and return the density derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial \rho}\right)_H$$

Returns

dP_drho_H [float] The density derivative of pressure of the phase at constant enthalpy, [Pa/mol/m^3]

dP_drho_S()

Method to calculate and return the density derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial \rho}\right)_S$$

Returns

dP_drho_S [float] The density derivative of pressure of the phase at constant entropy, [Pa/mol/m^3]

`dP_drho_U()`
Method to calculate and return the density derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial \rho}\right)_U$$

Returns

dP_drho_U [float] The density derivative of pressure of the phase at constant internal energy, [Pa/mol/m^3]

`dS_dP()`
Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.

Returns

dS_dP_T [float] Pressure derivative of entropy, [J/(mol*K*Pa)]

`dS_dP_T()`
Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.

Returns

dS_dP_T [float] Pressure derivative of entropy, [J/(mol*K*Pa)]

`dS_dV_P()`
Method to calculate and return the volume derivative of entropy of the phase at constant pressure.

Returns

dS_dV_P [float] Volume derivative of entropy, [J/(K*m^3)]

`dS_dV_T()`
Method to calculate and return the volume derivative of entropy of the phase at constant temperature.

Returns

dS_dV_T [float] Volume derivative of entropy, [J/(K*m^3)]

`dS_mass_dP()`
Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.

$$\left(\frac{\partial S_{\text{mass}}}{\partial P}\right)_T$$

Returns

dS_mass_dP [float] The pressure derivative of mass entropy of the phase at constant temperature, [J/(mol*K)/Pa]

`dS_mass_dP_T()`
Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.

$$\left(\frac{\partial S_{\text{mass}}}{\partial P}\right)_T$$

Returns

dS_mass_dP_T [float] The pressure derivative of mass entropy of the phase at constant temperature, [J/(mol*K)/Pa]

`dS_mass_dP_V()`
Method to calculate and return the pressure derivative of mass entropy of the phase at constant volume.

$$\left(\frac{\partial S_{\text{mass}}}{\partial P}\right)_V$$

Returns

dS_mass_dP_V [float] The pressure derivative of mass entropy of the phase at constant volume, [J/(mol*K)/Pa]

`dS_mass_dT()`
Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.

$$\left(\frac{\partial S_{\text{mass}}}{\partial T}\right)_P$$

Returns

dS_mass_dT [float] The temperature derivative of mass entropy of the phase at constant pressure, [J/(mol*K)/K]

`dS_mass_dT_P()`
Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.

$$\left(\frac{\partial S_{\text{mass}}}{\partial T}\right)_P$$

Returns

dS_mass_dT_P [float] The temperature derivative of mass entropy of the phase at constant pressure, [J/(mol*K)/K]

`dS_mass_dT_V()`
Method to calculate and return the temperature derivative of mass entropy of the phase at constant volume.

$$\left(\frac{\partial S_{\text{mass}}}{\partial T}\right)_V$$

Returns

dS_mass_dT_V [float] The temperature derivative of mass entropy of the phase at constant volume, [J/(mol*K)/K]

`dS_mass_dV_P()`

Method to calculate and return the volume derivative of mass entropy of the phase at constant pressure.

$$\left(\frac{\partial S_{\text{mass}}}{\partial V}\right)_P$$

Returns

dS_mass_dV_P [float] The volume derivative of mass entropy of the phase at constant pressure, [J/(mol*K)/m^3/mol]

`dS_mass_dV_T()`

Method to calculate and return the volume derivative of mass entropy of the phase at constant temperature.

$$\left(\frac{\partial S_{\text{mass}}}{\partial V}\right)_T$$

Returns

dS_mass_dV_T [float] The volume derivative of mass entropy of the phase at constant temperature, [J/(mol*K)/m^3/mol]

`dT_dP_A()`

Method to calculate and return the pressure derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial P}\right)_A$$

Returns

dT_dP_A [float] The pressure derivative of temperature of the phase at constant Helmholtz energy, [K/Pa]

`dT_dP_G()`

Method to calculate and return the pressure derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial P}\right)_G$$

Returns

dT_dP_G [float] The pressure derivative of temperature of the phase at constant Gibbs energy, [K/Pa]

`dT_dP_H()`

Method to calculate and return the pressure derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial P}\right)_H$$

Returns

dT_dP_H [float] The pressure derivative of temperature of the phase at constant enthalpy, [K/Pa]

dT_dP_S()

Method to calculate and return the pressure derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial P}\right)_S$$

Returns

dT_dP_S [float] The pressure derivative of temperature of the phase at constant entropy, [K/Pa]

dT_dP_U()

Method to calculate and return the pressure derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial P}\right)_U$$

Returns

dT_dP_U [float] The pressure derivative of temperature of the phase at constant internal energy, [K/Pa]

dT_dT_A()

Method to calculate and return the temperature derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial T}\right)_A$$

Returns

dT_dT_A [float] The temperature derivative of temperature of the phase at constant Helmholtz energy, [K/K]

dT_dT_G()

Method to calculate and return the temperature derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial T}\right)_G$$

Returns

dT_dT_G [float] The temperature derivative of temperature of the phase at constant Gibbs energy, [K/K]

`dT_dT_H()`

Method to calculate and return the temperature derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial T}\right)_{H}$$

Returns

dT_dT_H [float] The temperature derivative of temperature of the phase at constant enthalpy, [K/K]

`dT_dT_S()`

Method to calculate and return the temperature derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial T}\right)_{S}$$

Returns

dT_dT_S [float] The temperature derivative of temperature of the phase at constant entropy, [K/K]

`dT_dT_U()`

Method to calculate and return the temperature derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial T}\right)_{U}$$

Returns

dT_dT_U [float] The temperature derivative of temperature of the phase at constant internal energy, [K/K]

`dT_dV_A()`

Method to calculate and return the volume derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial V}\right)_{A}$$

Returns

dT_dV_A [float] The volume derivative of temperature of the phase at constant Helmholtz energy, [K/m^3/mol]

`dT_dV_G()`

Method to calculate and return the volume derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial V}\right)_G$$

Returns

dT_dV_G [float] The volume derivative of temperature of the phase at constant Gibbs energy, [K/m^3/mol]

dT_dV_H()

Method to calculate and return the volume derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial V}\right)_H$$

Returns

dT_dV_H [float] The volume derivative of temperature of the phase at constant enthalpy, [K/m^3/mol]

dT_dV_S()

Method to calculate and return the volume derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial V}\right)_S$$

Returns

dT_dV_S [float] The volume derivative of temperature of the phase at constant entropy, [K/m^3/mol]

dT_dV_U()

Method to calculate and return the volume derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial V}\right)_U$$

Returns

dT_dV_U [float] The volume derivative of temperature of the phase at constant internal energy, [K/m^3/mol]

dT_drho_A()

Method to calculate and return the density derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial \rho}\right)_A$$

Returns

dT_drho_A [float] The density derivative of temperature of the phase at constant Helmholtz energy, [K/mol/m^3]

`dT_drho_G()`

Method to calculate and return the density derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial \rho}\right)_G$$

Returns

dT_drho_G [float] The density derivative of temperature of the phase at constant Gibbs energy, [K/mol/m^3]

`dT_drho_H()`

Method to calculate and return the density derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial \rho}\right)_H$$

Returns

dT_drho_H [float] The density derivative of temperature of the phase at constant enthalpy, [K/mol/m^3]

`dT_drho_S()`

Method to calculate and return the density derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial \rho}\right)_S$$

Returns

dT_drho_S [float] The density derivative of temperature of the phase at constant entropy, [K/mol/m^3]

`dT_drho_U()`

Method to calculate and return the density derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial \rho}\right)_U$$

Returns

dT_drho_U [float] The density derivative of temperature of the phase at constant internal energy, [K/mol/m^3]

`dU_dP()`

Method to calculate and return the constant-temperature pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_T = -P\left(\frac{\partial V}{\partial P}\right)_T - V + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dU_dP [float] Constant-temperature pressure derivative of internal energy, [J/(mol*Pa)]

dU_dP_T()

Method to calculate and return the constant-temperature pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_T = -P\left(\frac{\partial V}{\partial P}\right)_T - V + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dU_dP [float] Constant-temperature pressure derivative of internal energy, [J/(mol*Pa)]

dU_dP_V()

Method to calculate and return the constant-volume pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_V = \left(\frac{\partial H}{\partial P}\right)_V - V$$

Returns

dU_dP_V [float] Constant-volume pressure derivative of internal energy, [J/(mol*Pa)]

dU_dT()

Method to calculate and return the constant-pressure temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_P = -P\left(\frac{\partial V}{\partial T}\right)_P + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dU_dT [float] Constant-pressure temperature derivative of internal energy, [J/(mol*K)]

dU_dT_P()

Method to calculate and return the constant-pressure temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_P = -P\left(\frac{\partial V}{\partial T}\right)_P + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dU_dT [float] Constant-pressure temperature derivative of internal energy, [J/(mol*K)]

dU_dT_V()

Method to calculate and return the constant-volume temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_V = \left(\frac{\partial H}{\partial T}\right)_V - V\left(\frac{\partial P}{\partial T}\right)_V$$

Returns

dU_dT_V [float] Constant-volume temperature derivative of internal energy, [J/(mol*K)]

dU_dV_P()

Method to calculate and return the constant-pressure volume derivative of internal energy.

$$\left(\frac{\partial U}{\partial V}\right)_P = \left(\frac{\partial U}{\partial T}\right)_P\left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dU_dV_P [float] Constant-pressure volume derivative of internal energy, [J/(m^3)]

dU_dV_T()
Method to calculate and return the constant-temperature volume derivative of internal energy.

$$\left(\frac{\partial U}{\partial V}\right)_T = \left(\frac{\partial U}{\partial P}\right)_T \left(\frac{\partial P}{\partial V}\right)_T$$

Returns

dU_dV_T [float] Constant-temperature volume derivative of internal energy, [J/(m^3)]

dU_mass_dP()
Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.

$$\left(\frac{\partial U_{\text{mass}}}{\partial P}\right)_T$$

Returns

dU_mass_dP [float] The pressure derivative of mass internal energy of the phase at constant temperature, [J/mol/Pa]

dU_mass_dP_T()
Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.

$$\left(\frac{\partial U_{\text{mass}}}{\partial P}\right)_T$$

Returns

dU_mass_dP_T [float] The pressure derivative of mass internal energy of the phase at constant temperature, [J/mol/Pa]

dU_mass_dP_V()
Method to calculate and return the pressure derivative of mass internal energy of the phase at constant volume.

$$\left(\frac{\partial U_{\text{mass}}}{\partial P}\right)_V$$

Returns

dU_mass_dP_V [float] The pressure derivative of mass internal energy of the phase at constant volume, [J/mol/Pa]

dU_mass_dT()
Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.

$$\left(\frac{\partial U_{\text{mass}}}{\partial T}\right)_P$$

Returns

dU_mass_dT [float] The temperature derivative of mass internal energy of the phase at constant pressure, [J/mol/K]

`dU_mass_dT_P()`

Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.

$$\left(\frac{\partial U_{\text{mass}}}{\partial T}\right)_P$$

Returns

dU_mass_dT_P [float] The temperature derivative of mass internal energy of the phase at constant pressure, [J/mol/K]

`dU_mass_dT_V()`

Method to calculate and return the temperature derivative of mass internal energy of the phase at constant volume.

$$\left(\frac{\partial U_{\text{mass}}}{\partial T}\right)_V$$

Returns

dU_mass_dT_V [float] The temperature derivative of mass internal energy of the phase at constant volume, [J/mol/K]

`dU_mass_dV_P()`

Method to calculate and return the volume derivative of mass internal energy of the phase at constant pressure.

$$\left(\frac{\partial U_{\text{mass}}}{\partial V}\right)_P$$

Returns

dU_mass_dV_P [float] The volume derivative of mass internal energy of the phase at constant pressure, [J/mol/m^3/mol]

`dU_mass_dV_T()`

Method to calculate and return the volume derivative of mass internal energy of the phase at constant temperature.

$$\left(\frac{\partial U_{\text{mass}}}{\partial V}\right)_T$$

Returns

dU_mass_dV_T [float] The volume derivative of mass internal energy of the phase at constant temperature, [J/mol/m^3/mol] `dV_dP_A()`

Method to calculate and return the pressure derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial P}\right)_{A}$$

Returns

dV_dP_A [float] The pressure derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/Pa]

`dV_dP_G()`

Method to calculate and return the pressure derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial P}\right)_{G}$$

Returns

dV_dP_G [float] The pressure derivative of volume of the phase at constant Gibbs energy, [m^3/mol/Pa]

`dV_dP_H()`

Method to calculate and return the pressure derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial P}\right)_{H}$$

Returns

dV_dP_H [float] The pressure derivative of volume of the phase at constant enthalpy, [m^3/mol/Pa]

`dV_dP_S()`

Method to calculate and return the pressure derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial P}\right)_{S}$$

Returns

dV_dP_S [float] The pressure derivative of volume of the phase at constant entropy, [m^3/mol/Pa]

`dV_dP_U()`

Method to calculate and return the pressure derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial P}\right)_{U}$$

Returns

dV_dP_U [float] The pressure derivative of volume of the phase at constant internal energy, [m^3/mol/Pa]

`dV_dT_A()`

Method to calculate and return the temperature derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial T}\right)_A$$

Returns

dV_dT_A [float] The temperature derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/K]

`dV_dT_G()`

Method to calculate and return the temperature derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial T}\right)_G$$

Returns

dV_dT_G [float] The temperature derivative of volume of the phase at constant Gibbs energy, [m^3/mol/K]

`dV_dT_H()`

Method to calculate and return the temperature derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial T}\right)_H$$

Returns

dV_dT_H [float] The temperature derivative of volume of the phase at constant enthalpy, [m^3/mol/K]

`dV_dT_S()`

Method to calculate and return the temperature derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial T}\right)_S$$

Returns

dV_dT_S [float] The temperature derivative of volume of the phase at constant entropy, [m^3/mol/K]

`dV_dT_U()`

Method to calculate and return the temperature derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial T}\right)_U$$

Returns

dV_dT_U [float] The temperature derivative of volume of the phase at constant internal energy, [m^3/mol/K]

`dV_dV_A()`

Method to calculate and return the volume derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial V}\right)_A$$

Returns

dV_dV_A [float] The volume derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/m^3/mol]

`dV_dV_G()`

Method to calculate and return the volume derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial V}\right)_G$$

Returns

dV_dV_G [float] The volume derivative of volume of the phase at constant Gibbs energy, [m^3/mol/m^3/mol]

`dV_dV_H()`

Method to calculate and return the volume derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial V}\right)_H$$

Returns

dV_dV_H [float] The volume derivative of volume of the phase at constant enthalpy, [m^3/mol/m^3/mol]

`dV_dV_S()`

Method to calculate and return the volume derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial V}\right)_S$$

Returns

dV_dV_S [float] The volume derivative of volume of the phase at constant entropy, [m^3/mol/m^3/mol]

dV_dV_U()

Method to calculate and return the volume derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial V}\right)_{U}$$

Returns

dV_dV_U [float] The volume derivative of volume of the phase at constant internal energy, [m^3/mol/m^3/mol]

dV_drho_A()

Method to calculate and return the density derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{A}$$

Returns

dV_drho_A [float] The density derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/mol/m^3]

dV_drho_G()

Method to calculate and return the density derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{G}$$

Returns

dV_drho_G [float] The density derivative of volume of the phase at constant Gibbs energy, [m^3/mol/mol/m^3]

dV_drho_H()

Method to calculate and return the density derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{H}$$

Returns

dV_drho_H [float] The density derivative of volume of the phase at constant enthalpy, [m^3/mol/mol/m^3]

dV_drho_S()

Method to calculate and return the density derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{S}$$

Returns

dV_drho_S [float] The density derivative of volume of the phase at constant entropy, [m^3/mol/mol/m^3]

`dV_drho_U()`
Method to calculate and return the density derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{U}$$

Returns

dV_drho_U [float] The density derivative of volume of the phase at constant internal energy, [m^3/mol/mol/m^3]

`property dipoles`
Dipole moments for each component, [debye].

Returns

dipoles [list[float]] Dipole moments for each component, [debye].

`drho_dP_A()`
Method to calculate and return the pressure derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial P}\right)_{A}$$

Returns

drho_dP_A [float] The pressure derivative of density of the phase at constant Helmholtz energy, [mol/m^3/Pa]

`drho_dP_G()`
Method to calculate and return the pressure derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial P}\right)_{G}$$

Returns

drho_dP_G [float] The pressure derivative of density of the phase at constant Gibbs energy, [mol/m^3/Pa]

`drho_dP_H()`
Method to calculate and return the pressure derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial P}\right)_{H}$$

Returns

drho_dP_H [float] The pressure derivative of density of the phase at constant enthalpy, [mol/m^3/Pa]

drho_dP_S()

Method to calculate and return the pressure derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial P}\right)_S$$

Returns

drho_dP_S [float] The pressure derivative of density of the phase at constant entropy, [mol/m^3/Pa]

drho_dP_U()

Method to calculate and return the pressure derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial P}\right)_U$$

Returns

drho_dP_U [float] The pressure derivative of density of the phase at constant internal energy, [mol/m^3/Pa]

drho_dT_A()

Method to calculate and return the temperature derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial T}\right)_A$$

Returns

drho_dT_A [float] The temperature derivative of density of the phase at constant Helmholtz energy, [mol/m^3/K]

drho_dT_G()

Method to calculate and return the temperature derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial T}\right)_G$$

Returns

drho_dT_G [float] The temperature derivative of density of the phase at constant Gibbs energy, [mol/m^3/K]

drho_dT_H()

Method to calculate and return the temperature derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial T}\right)_H$$

Returns

drho_dT_H [float] The temperature derivative of density of the phase at constant enthalpy, [mol/m^3/K]

`drho_dT_S()`

Method to calculate and return the temperature derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial T}\right)_S$$

Returns

drho_dT_S [float] The temperature derivative of density of the phase at constant entropy, [mol/m^3/K]

`drho_dT_U()`

Method to calculate and return the temperature derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial T}\right)_U$$

Returns

drho_dT_U [float] The temperature derivative of density of the phase at constant internal energy, [mol/m^3/K]

`drho_dV_A()`

Method to calculate and return the volume derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial V}\right)_A$$

Returns

drho_dV_A [float] The volume derivative of density of the phase at constant Helmholtz energy, [mol/m^3/m^3/mol]

`drho_dV_G()`

Method to calculate and return the volume derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial V}\right)_G$$

Returns

drho_dV_G [float] The volume derivative of density of the phase at constant Gibbs energy, [mol/m^3/m^3/mol]

`drho_dV_H()`

Method to calculate and return the volume derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{H}$$

Returns

drho_dV_H [float] The volume derivative of density of the phase at constant enthalpy, [mol/m^3/m^3/mol]

`drho_dV_S()`
Method to calculate and return the volume derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{S}$$

Returns

drho_dV_S [float] The volume derivative of density of the phase at constant entropy, [mol/m^3/m^3/mol]

`drho_dV_U()`
Method to calculate and return the volume derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{U}$$

Returns

drho_dV_U [float] The volume derivative of density of the phase at constant internal energy, [mol/m^3/m^3/mol]

`drho_drho_A()`
Method to calculate and return the density derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{A}$$

Returns

drho_drho_A [float] The density derivative of density of the phase at constant Helmholtz energy, [mol/m^3/mol/m^3]

`drho_drho_G()`
Method to calculate and return the density derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{G}$$

Returns

drho_drho_G [float] The density derivative of density of the phase at constant Gibbs energy, [mol/m^3/mol/m^3]

drho_drho_H()
Method to calculate and return the density derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_H$$

Returns

drho_drho_H [float] The density derivative of density of the phase at constant enthalpy, [mol/m^3/mol/m^3]

drho_drho_S()
Method to calculate and return the density derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_S$$

Returns

drho_drho_S [float] The density derivative of density of the phase at constant entropy, [mol/m^3/mol/m^3]

drho_drho_U()
Method to calculate and return the density derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_U$$

Returns

drho_drho_U [float] The density derivative of density of the phase at constant internal energy, [mol/m^3/mol/m^3]

property economic_statuses
Status of each component in in relation to import and export from various regions, [-].

Returns

economic_statuses [list[dict]] Status of each component in in relation to import and export from various regions, [-].

flashed = True

property formulas
Formulas of each component, [-].

Returns

formulas [list[str]] Formulas of each component, [-].

property heaviest_liquid
The liquid-like phase with the highest mass density, [-]

Returns

heaviest_liquid [Phase or None] Phase with the highest mass density or None if there are no liquid like phases, [-] `isentropic_exponent()`

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.

$$k = -\frac{V}{P}\frac{C_p}{C_v}\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

k_PV [float] Isentropic exponent of a real fluid, [-]

`isentropic_exponent_PT()`

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $P^{(1-k)}T^k = \text{const}$.

$$k = \frac{1}{1 - \frac{P}{C_p}\left(\frac{\partial V}{\partial T}\right)_P}$$

Returns

k_PT [float] Isentropic exponent of a real fluid, [-]

`isentropic_exponent_PV()`

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.

$$k = -\frac{V}{P}\frac{C_p}{C_v}\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

k_PV [float] Isentropic exponent of a real fluid, [-]

`isentropic_exponent_TV()`

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $TV^{k-1} = \text{const}$.

$$k = 1 + \frac{V}{C_v}\left(\frac{\partial P}{\partial T}\right)_V$$

Returns

k_TV [float] Isentropic exponent of a real fluid, [-]

`isobaric_expansion()`

Method to calculate and return the isobatic expansion coefficient of the bulk according to the selected calculation methodology.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Returns

beta [float] Isobaric coefficient of a thermal expansion, [1/K]

`isothermal_bulk_modulus()`

Method to calculate and return the isothermal bulk modulus of the phase.

$$K_T = -V\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

isothermal_bulk_modulus [float] Isothermal bulk modulus, [Pa]

k()

Calculate and return the thermal conductivity of the bulk according to the selected thermal conductivity settings in `BulkSettings`, the settings in *ThermalConductivityGasMixture* and *ThermalConductivityLiquidMixture*, and the configured pure-component settings in *ThermalConductivityGas* and *ThermalConductivityLiquid*.

Returns

k [float] Thermal Conductivity of bulk phase calculated with mixing rules, [Pa*s]

kappa()

Method to calculate and return the isothermal compressibility of the bulk according to the selected calculation methodology.

$$\kappa = -\frac{1}{V}\left(\frac{\partial V}{\partial P}\right)_T$$

Returns

kappa [float] Isothermal coefficient of compressibility, [1/Pa]

property legal_statuses

Status of each component in in relation to import and export rules from various regions, [-].

Returns

legal_statuses [list[dict]] Status of each component in in relation to import and export rules from various regions, [-].

property lightest_liquid

The liquid-like phase with the lowest mass density, [-]

Returns

lightest_liquid [Phase or None] Phase with the lowest mass density or None if there are no liquid like phases, [-]

liquid_bulk = None

property logPs

Octanol-water partition coefficients for each component, [-].

Returns

logPs [list[float]] Octanol-water partition coefficients for each component, [-].

log_zs()

Method to calculate and return the log of mole fractions specified. These are used in calculating entropy and in many other formulas.

$$\ln z_i$$

Returns

log_zs [list[float]] Log of mole fractions, [-]

max_liquid_phases = 1

molar_water_content(*phase=None*)

Method to calculate and return the molar water content; this is the g/mol of the fluid which is coming from water, [g/mol].

$$\text{water content} = \text{MW}_{H2O} w_{H2O}$$

Returns

molar_water_content [float] Molar water content, [g/mol]

`property molecular_diameters`

Lennard-Jones molecular diameters for each component, [angstrom].

Returns

molecular_diameters [list[float]] Lennard-Jones molecular diameters for each component, [angstrom].

`mu()`

Calculate and return the viscosity of the bulk according to the selected viscosity settings in `BulkSettings`, the settings in *ViscosityGasMixture* and *ViscosityLiquidMixture*, and the configured pure-component settings in *ViscosityGas* and *ViscosityLiquid*.

Returns

mu [float] Viscosity of bulk phase calculated with mixing rules, [Pa*s]

`property names`

Names for each component, [-].

Returns

names [list[str]] Names for each component, [-].

`nu(phase=None)`

Method to calculate and return the kinematic viscosity of the equilibrium state.

$$\nu = \frac{\mu}{\rho}$$

Returns

nu [float] Kinematic viscosity, [m^2/s]

`property omegas`

Acentric factors for each component, [-].

Returns

omegas [list[float]] Acentric factors for each component, [-].

`property phase`

Method to calculate and return a string representing the phase of the mixture. The return string uses 'V' to represent the gas phase, 'L' to represent a liquid phase, and 'S' to represent a solid phase (always in that order).

A state with three liquids, two solids, and a gas would return 'VLLLSS'.

Returns

phase [str] Phase string, [-]

`property phase_STPs`

Standard states ('g', 'l', or 's') for each component, [-].

Returns

phase_STPs [list[str]] Standard states ('g', 'l', or 's') for each component, [-].

pseudo_Pc(*phase=None*)

Method to calculate and return the pseudocritical pressure calculated using Kay's rule (linear mole fractions):

$$P_{c,pseudo} = \sum_i z_i P_{c,i}$$

Returns

pseudo_Pc [float] Pseudocritical pressure of the phase, [Pa]

pseudo_Tc(*phase=None*)

Method to calculate and return the pseudocritical temperature calculated using Kay's rule (linear mole fractions):

$$T_{c,pseudo} = \sum_i z_i T_{c,i}$$

Returns

pseudo_Tc [float] Pseudocritical temperature of the phase, [K]

pseudo_Vc(*phase=None*)

Method to calculate and return the pseudocritical volume calculated using Kay's rule (linear mole fractions):

$$V_{c,pseudo} = \sum_i z_i V_{c,i}$$

Returns

pseudo_Vc [float] Pseudocritical volume of the phase, [m^3/mol]

pseudo_Zc(*phase=None*)

Method to calculate and return the pseudocritical compressibility calculated using Kay's rule (linear mole fractions):

$$Z_{c,pseudo} = \sum_i z_i Z_{c,i}$$

Returns

pseudo_Zc [float] Pseudocritical compressibility of the phase, [-]

property quality

Method to return the mass vapor fraction of the equilibrium state. If no vapor/gas is present, 0 is always returned. This is normally called the quality.

Returns

quality [float] Vapor mass fraction, [-]

reacted = False

rho()

Method to calculate and return the molar density of the phase.

$$\rho = frac1V$$

Returns

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

`rho_mass`(*phase=None*)
Method to calculate and return mass density of the phase.

$$\rho = \frac{MW}{1000 \cdot VM}$$

Returns

rho_mass [float] Mass density, [kg/m^3]

`rho_mass_liquid_ref`(*phase=None*)
Method to calculate and return the liquid reference mass density according to the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings` and the composition of the phase.

Returns

rho_mass_liquid_ref [float] Liquid mass density at the reference condition, [kg/m^3]

`property rhocs`
Molar densities at the critical point for each component, [mol/m^3].

Returns

rhocs [list[float]] Molar densities at the critical point for each component, [mol/m^3].

`property rhocs_mass`
Densities at the critical point for each component, [kg/m^3].

Returns

rhocs_mass [list[float]] Densities at the critical point for each component, [kg/m^3].

`property rhog_STPs`
Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

Returns

rhog_STPs [list[float]] Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

`property rhog_STPs_mass`
Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

Returns

rhog_STPs_mass [list[float]] Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

`property rhol_60Fs`
Liquid molar densities for each component at 60 °F, [mol/m^3].

Returns

rhol_60Fs [list[float]] Liquid molar densities for each component at 60 °F, [mol/m^3].

`property rhol_60Fs_mass`
Liquid mass densities for each component at 60 °F, [kg/m^3].

Returns

rhol_60Fs_mass [list[float]] Liquid mass densities for each component at 60 °F, [kg/m^3].

`property rhol_STPs`
Molar liquid densities at STP for each component, [mol/m^3].

Returns

rhol_STPs [list[float]] Molar liquid densities at STP for each component, [mol/m^3].

property rhol_STPs_mass

Liquid densities at STP for each component, [kg/m^3].

Returns

rhol_STPs_mass [list[float]] Liquid densities at STP for each component, [kg/m^3].

property rhos_Tms

Solid molar densities for each component at their respective melting points, [mol/m^3].

Returns

rhos_Tms [list[float]] Solid molar densities for each component at their respective melting points, [mol/m^3].

property rhos_Tms_mass

Solid mass densities for each component at their melting point, [kg/m^3].

Returns

rhos_Tms_mass [list[float]] Solid mass densities for each component at their melting point, [kg/m^3].

sigma()

Calculate and return the surface tension of the bulk according to the selected surface tension settings in `BulkSettings`, the settings in *SurfaceTensionMixture* and the configured pure-component settings in *SurfaceTension*.

Returns

sigma [float] Surface tension of bulk phase calculated with mixing rules, [N/m]

Notes

A value is only returned if all phases in the bulk are liquids; this property is for a liquid-ideal gas calculation, not the interfacial tension between two liquid phases.

property sigma_STPs

Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

Returns

sigma_STPs [list[float]] Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

property sigma_Tbs

Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

Returns

sigma_Tbs [list[float]] Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

property sigma_Tms

Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

Returns

sigma_Tms [list[float]] Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

property similarity_variables

Similarity variables for each component, [mol/g].

Returns

similarity_variables [list[float]] Similarity variables for each component, [mol/g].

property smiless

SMILES identifiers for each component, [-].

Returns

smiless [list[str]] SMILES identifiers for each component, [-].

solid_bulk = None

property solubility_parameters

Solubility parameters for each component at 298.15 K, [Pa^0.5].

Returns

solubility_parameters [list[float]] Solubility parameters for each component at 298.15 K, [Pa^0.5].

speed_of_sound()

Method to calculate and return the molar speed of sound of the bulk according to the selected calculation methodology.

$$w = \left[-V^2 \left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

A similar expression based on molar density is:

$$w = \left[\left(\frac{\partial P}{\partial \rho}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

Returns

w [float] Speed of sound for a real gas, [m*kg^0.5/(s*mol^0.5)]

speed_of_sound_mass()

Method to calculate and return the speed of sound of the phase.

$$w = \left[-V^2 \frac{1000}{MW} \left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

Returns

w [float] Speed of sound for a real gas, [m/s]

value(*name*, *phase=None*)

Method to retrieve a property from a string. This more or less wraps *getattr*, but also allows for the property to be returned for a specific phase if *phase* is provided.

name could be a python property like 'Tms' or a callable method like 'H'; and if the property is on a per-phase basis like 'betas_mass', a phase object can be provided as the second argument and only the value for that phase will be returned.

Parameters

name [str] String representing the property, [-]

phase [`thermo.phase.Phase`, optional] Phase to retrieve the property for only (if specified), [-]

Returns

value [various] Value specified, [various]

`property water_index`

The index of the component water in the components. None if water is not present. Water is recognized by its CAS number.

Returns

water_index [int] The index of the component water, [-]

`property water_phase`

The liquid-like phase with the highest water mole fraction, [-]

Returns

water_phase [Phase or None] Phase with the highest water mole fraction or None if there are no liquid like phases with water, [-]

`property water_phase_index`

The liquid-like phase with the highest mole fraction of water, [-]

Returns

water_phase_index [int] Index into the attribute `EquilibriumState.liquids` which refers to the liquid-like phase with the highest water mole fraction, [-]

`ws`(*phase=None*)

Method to calculate and return the mass fractions of the phase, [-]

Returns

ws [list[float]] Mass fractions, [-]

`ws_no_water`(*phase=None*)

Method to calculate and return the mass fractions of all species in the phase, normalized to a water-free basis (the mass fraction of water returned is zero).

Returns

ws_no_water [list[float]] Mass fractions on a water free basis, [-]

`zs_no_water`(*phase=None*)

Method to calculate and return the mole fractions of all species in the phase, normalized to a water-free basis (the mole fraction of water returned is zero).

Returns

zs_no_water [list[float]] Mole fractions on a water free basis, [-]

7.13 Flash Calculations (thermo.flash)

This module contains classes and functions for performing flash calculations.

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

7.13.1 Main Interfaces

Pure Components

class thermo.flash.**FlashPureVLS**(*constants*, *correlations*, *gas*, *liquids*, *solids*, *settings=<thermo.bulk.BulkSettings object>*)

Bases: `thermo.flash.flash_base.Flash`

Class for performing flash calculations on pure-component systems. This class is subtantially more robust than using multicomponent algorithms on pure species. It is also faster. All parameters are also attributes.

The minimum information that is needed in addition to the `Phase` objects is:

- MW
- Vapor pressure curve if including liquids
- Sublimation pressure curve if including solids
- Functioning enthalpy models for each phase

Parameters

constants [`ChemicalConstantsPackage` object] Package of chemical constants; these are used as boundaries at times, initial guesses other times, and in all cases these properties are accessible as attributes of the resulting `EquilibriumState` object, [-]

correlations [`PropertyCorrelationsPackage`] Package of chemical T-dependent properties; these are used as boundaries at times, for initial guesses other times, and in all cases these properties are accessible as attributes of the resulting `EquilibriumState` object, [-]

gas [`Phase` object] A single phase which can represent the gas phase, [-]

liquids [list[`Phase`]] A list of phases for representing the liquid phase; normally only one liquid phase is present for a pure-component system, but multiple liquids are allowed for the really weird cases like having both parahydrogen and orthohydrogen. The liquid phase which calculates a lower Gibbs free energy is always used. [-]

solids [list[`Phase`]] A list of phases for representing the solid phase; it is very common for multiple solid forms of a compound to exist. For water ice, the list is very long - normally ice is in phase Ih but other phases are Ic, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, Square ice, and Amorphous ice. It is less common for there to be published, reliable, thermodynamic models for these different phases; for water there is the IAPWS-06 model for Ih, and another model here for phases Ih, Ic, II, III, IV, V, VI, IX, XI, XII. [-]

settings [`BulkSettings` object] Object containing settings for calculating bulk and transport properties, [-]

Notes

The algorithms in this object are mostly from [1] and [2]. They all boil down to newton methods with analytical derivatives. The phase with the lowest Gibbs energy is the most stable if there are multiple solutions.

Phase input combinations which have specific simplifying assumptions (and thus more speed) are:

- a *CEOSLiquid* and a *CEOSGas* with the same (consistent) parameters
- a *CEOSGas* with the *IGMIX* eos and a *GibbsExcessLiquid*
- a *IAPWS95Liquid* and a *IAPWS95Gas*
- a *CoolPropLiquid* and a *CoolPropGas*

Additional information that can be provided in the *ChemicalConstantsPackage* object and *PropertyCorrelationsPackage* object that may help convergence is:

- *Tc*, *Pc*, *omega*, *Tb*, and *atoms*
- Gas heat capacity correlations
- Liquid molar volume correlations
- Heat of vaporization correlations

References

[1], [2]

Examples

Create all the necessary objects using all of the default parameters for decane and do a flash at 300 K and 1 bar:

```
>>> from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CEOSGas,␣
↪FlashPureVLS
>>> constants, correlations = ChemicalConstantsPackage.from_IDs(['decane'])
>>> eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
>>> liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,␣
↪eos_kwargs=eos_kwargs)
>>> gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
>>> flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid],␣
↪solids=[])
>>> print(flasher.flash(T=300, P=1e5))
<EquilibriumState, T=300.0000, P=100000.0000, zs=[1.0], betas=[1.0], phases=[
↪<CEOSLiquid, T=300 K, P=100000 Pa>]>
```

Working with steam:

```
>>> from thermo import FlashPureVLS, IAPWS95Liquid, IAPWS95Gas, iapws_constants,␣
↪iapws_correlations
>>> liquid = IAPWS95Liquid(T=300, P=1e5, zs=[1])
>>> gas = IAPWS95Gas(T=300, P=1e5, zs=[1])
>>> flasher = FlashPureVLS(iapws_constants, iapws_correlations, gas, [liquid], [])
>>> PT = flasher.flash(T=800.0, P=1e7)
>>> PT.rho_mass()
```

(continues on next page)

(continued from previous page)

```
29.1071839176
>>> print(flasher.flash(T=600, VF=.5))
<EquilibriumState, T=600.0000, P=12344824.3572, zs=[1.0], betas=[0.5, 0.5], phases=[
↪<IAPWS95Gas, T=600 K, P=1.23448e+07 Pa>, <IAPWS95Liquid, T=600 K, P=1.23448e+07␣
↪Pa>]>
>>> print(flasher.flash(T=600.0, H=50802))
<EquilibriumState, T=600.0000, P=10000469.1288, zs=[1.0], betas=[1.0], phases=[
↪<IAPWS95Gas, T=600 K, P=1.00005e+07 Pa>]>
>>> print(flasher.flash(P=1e7, S=104.))
<EquilibriumState, T=599.6790, P=10000000.0000, zs=[1.0], betas=[1.0], phases=[
↪<IAPWS95Gas, T=599.679 K, P=1e+07 Pa>]>
>>> print(flasher.flash(V=.00061, U=55850))
<EquilibriumState, T=800.5922, P=10144789.0899, zs=[1.0], betas=[1.0], phases=[
↪<IAPWS95Gas, T=800.592 K, P=1.01448e+07 Pa>]>
```

Attributes

VL_IG_hack [bool] Whether or not to trust the saturation curve of the liquid phase; applied automatically to the `GibbsExcessLiquid` phase if there is a single liquid only, [-]

VL_EOS_hacks [bool] Whether or not to trust the saturation curve of the EOS liquid phase; applied automatically to the `CEOSLiquid` phase if there is a single liquid only, [-]

TPV_HSGUA_guess_maxiter [int] Maximum number of iterations to try when converging a shortcut model for flashes with one (T, P, V) spec and one (H, S, G, U, A) spec, [-]

TPV_HSGUA_guess_xtol [float] Convergence tolerance in the iteration variable when converging a shortcut model for flashes with one (T, P, V) spec and one (H, S, G, U, A) spec, [-]

TPV_HSGUA_maxiter [int] Maximum number of iterations to try when converging a flashes with one (T, P, V) spec and one (H, S, G, U, A) spec; this is on a per-phase basis, so if there is a liquid and a gas phase, the maximum number of iterations that could end up being tried would be twice this, [-]

TPV_HSGUA_xtol [float] Convergence tolerance in the iteration variable dimension when converging a flash with one (T, P, V) spec and one (H, S, G, U, A) spec, [-]

TVF_maxiter [int] Maximum number of iterations to try when converging a flashes with a temperature and vapor fraction specification, [-]

TVF_xtol [float] Convergence tolerance in the temperature dimension when converging a flashes with a temperature and vapor fraction specification, [-]

PVF_maxiter [int] Maximum number of iterations to try when converging a flashes with a pressure and vapor fraction specification, [-]

PVF_xtol [float] Convergence tolerance in the pressure dimension when converging a flashes with a pressure and vapor fraction specification, [-]

TSF_maxiter [int] Maximum number of iterations to try when converging a flashes with a temperature and solid fraction specification, [-]

TSF_xtol [float] Convergence tolerance in the temperature dimension when converging a flashes with a temperature and solid fraction specification, [-]

PSF_maxiter [int] Maximum number of iterations to try when converging a flashes with a pressure and solid fraction specification, [-]

PSF_xtol [float] Convergence tolerance in the pressure dimension when converging a flashes with a pressure and solid fraction specification, [-]

Vapor-Liquid Systems

class thermo.flash.**FlashVL**(*constants*, *correlations*, *gas*, *liquid*, *settings=<thermo.bulk.BulkSettings object>*)

Bases: *thermo.flash.flash_base.Flash*

Class for performing flash calculations on one and two phase vapor and liquid multicomponent systems. Use *FlashVLN* for systems which can have multiple liquid phases.

The minimum information that is needed in addition to the Phase objects is:

- MWs
- Vapor pressure curve
- Functioning enthalpy models for each phase

Parameters

constants [*ChemicalConstantsPackage* object] Package of chemical constants; these are used as boundaries at times, initial guesses other times, and in all cases these properties are accessible as attributes of the resulting *EquilibriumState* object, [-]

correlations [*PropertyCorrelationsPackage*] Package of chemical T-dependent properties; these are used as boundaries at times, for initial guesses other times, and in all cases these properties are accessible as attributes of the resulting *EquilibriumState* object, [-]

gas [*Phase* object] A single phase which can represent the gas phase, [-]

liquid [*Phase*] A single phase which can represent the liquid phase, [-]

settings [*BulkSettings* object] Object containing settings for calculating bulk and transport properties, [-]

Notes

The algorithms in this object are mostly from [1], [2] and [3]. Sequential substitution without acceleration is used by default to converge two-phase systems.

Quasi-newton methods are used by default to converge bubble and dew point calculations.

Flashes with one (T, P, V) spec and one (H, S, G, U, A) spec are solved by a 1D search over PT flashes.

Additional information that can be provided in the *ChemicalConstantsPackage* object and *PropertyCorrelationsPackage* object that may help convergence is:

- *Tc*, *Pc*, *omega*, *Tb*, and *atoms*
- Gas heat capacity correlations
- Liquid molar volume correlations
- Heat of vaporization correlations

> **Warning:** If this flasher is used on systems that can form two or more liquid phases, and the flash specs are in that region, there is no guarantee which solution is returned. Sometimes it is almost random, jumping back and forth and providing nasty discontinuities.

References

[1], [2], [3]

Examples

For the system methane-ethane-nitrogen with a composition [0.965, 0.018, 0.017], calculate the vapor fraction of the system and equilibrium phase compositions at 110 K and 1 bar. Use the Peng-Robinson equation of state and the chemsep sample interaction parameter database.

```
>>> from thermo import ChemicalConstantsPackage, CEOSGas, CEOSLiquid, PRMIX, FlashVL
>>> from thermo.interaction_parameters import IPDB
>>> constants, properties = ChemicalConstantsPackage.from_IDs(['methane', 'ethane',
↪'nitrogen'])
>>> kijs = IPDB.get_ip_asymmetric_matrix('ChemSep PR', constants.CASs, 'kij')
>>> kijs
[[0.0, -0.0059, 0.0289], [-0.0059, 0.0, 0.0533], [0.0289, 0.0533, 0.0]]
>>> eos_kwargs = {'Pcs': constants.Pcs, 'Tcs': constants.Tcs, 'omegas': constants.
↪omegas, 'kijs': kijs}
>>> gas = CEOSGas(PRMIX, eos_kwargs=eos_kwargs, HeatCapacityGases=properties.
↪HeatCapacityGases)
>>> liquid = CEOSLiquid(PRMIX, eos_kwargs=eos_kwargs, HeatCapacityGases=properties.
↪HeatCapacityGases)
>>> flasher = FlashVL(constants, properties, liquid=liquid, gas=gas)
>>> zs = [0.965, 0.018, 0.017]
>>> PT = flasher.flash(T=110.0, P=1e5, zs=zs)
>>> PT.VF, PT.gas.zs, PT.liquid0.zs
(0.10365, [0.881788, 2.6758e-05, 0.11818], [0.97462, 0.02007, 0.005298])
```

A few more flashes with the same system to showcase the functionality of the *flash* interface:

```
>>> flasher.flash(P=1e5, VF=1, zs=zs).T
133.6
>>> flasher.flash(T=133, VF=0, zs=zs).P
518367.4
>>> flasher.flash(P=PT.P, H=PT.H(), zs=zs).T
110.0
>>> flasher.flash(P=PT.P, S=PT.S(), zs=zs).T
110.0
>>> flasher.flash(T=PT.T, H=PT.H(), zs=zs).T
110.0
>>> flasher.flash(T=PT.T, S=PT.S(), zs=zs).T
110.0
```

Attributes

PT_SS_MAXITER [int] Maximum number of sequential substitution iterations to try when converging a two-phase solution, [-]

PT_SS_TOL [float] Convergence tolerance in sequential substitution [-]

PT_SS_POLISH [bool] When set to True, flashes which are very near a vapor fraction of 0 or 1 are converged to a higher tolerance to ensure the solution is correct; without this, a flash might converge to a vapor fraction of -1e-7 and be called single phase, but with this the correct solution may be found to be 1e-8 and will be correctly returned as two phase.[-]

PT_SS_POLISH_VF [float] What tolerance to a vapor fraction of 0 or 1; this is an absolute vapor fraction value, [-]

PT_SS_POLISH_MAXITER [int] Maximum number of sequential substitution iterations to try when converging a two-phase solution that has been detected to be very sensitive, with a vapor fraction near 0 or 1 [-]

PT_SS_POLISH_TOL [float] Convergence tolerance in sequential substitution when converging a two-phase solution that has been detected to be very sensitive, with a vapor fraction near 0 or 1 [-]

PT_STABILITY_MAXITER [int] Maximum number of iterations to try when converging a stability test, [-]

PT_STABILITY_XTOL [float] Convergence tolerance in the stability test [-]

DEW_BUBBLE_VF_K_COMPOSITION_INDEPENDENT_XTOL [float] Convergence tolerance in Newton solver for bubble, dew, and vapor fraction spec flashes when both the liquid and gas model's K values do not dependent on composition, [-]

DEW_BUBBLE_QUASI_NEWTON_XTOL [float] Convergence tolerance in quasi-Newton bubble and dew point flashes, [-]

DEW_BUBBLE_QUASI_NEWTON_MAXITER [int] Maximum number of iterations to use in quasi-Newton bubble and dew point flashes, [-]

DEW_BUBBLE_NEWTON_XTOL [float] Convergence tolerance in Newton bubble and dew point flashes, [-]

DEW_BUBBLE_NEWTON_MAXITER [int] Maximum number of iterations to use in Newton bubble and dew point flashes, [-]

TPV_HSGUA_BISECT_XTOL [float] Tolerance in the iteration variable when converging a flash with one (T, P, V) spec and one (H, S, G, U, A) spec using a bisection-type solver, [-]

TPV_HSGUA_BISECT_YTOL [float] Absolute tolerance in the (H, S, G, U, A) spec when converging a flash with one (T, P, V) spec and one (H, S, G, U, A) spec using a bisection-type solver, [-]

TPV_HSGUA_BISECT_YTOL_ONLY [bool] When True, the *TPV_HSGUA_BISECT_XTOL* setting is ignored and the flash is considered converged once *TPV_HSGUA_BISECT_YTOL* is satisfied, [-]

TPV_HSGUA_NEWTON_XTOL [float] Tolerance in the iteration variable when converging a flash with one (T, P, V) spec and one (H, S, G, U, A) spec using a full newton solver, [-]

TPV_HSGUA_NEWTON_MAXITER [float] Maximum number of iterations when converging a flash with one (T, P, V) spec and one (H, S, G, U, A) spec using full newton solver, [-]

TPV_HSGUA_SECANT_MAXITER [float] Maximum number of iterations when converging a flash with one (T, P, V) spec and one (H, S, G, U, A) spec using a secant solver, [-]

HSGUA_NEWTON_ANALYTICAL_JAC [bool] Whether or not to calculate the full newton jacobian analytically or numerically; this would need to be set to False if the phase objects used in the flash do not have complete analytical derivatives implemented, [-]

Vapor and Multiple Liquid Systems

class thermo.flash.**FlashVLN**(*constants*, *correlations*, *liquids*, *gas*, *solids=None*, *settings=<thermo.bulk.BulkSettings object>*)

Bases: *thermo.flash.flash_vl.FlashVL*

Class for performing flash calculations on multiphase vapor-liquid systems. This rigorous class does not make any assumptions and will search for up to the maximum amount of liquid phases specified by the user. Vapor and each liquid phase do not need to use a consistent thermodynamic model.

The minimum information that is needed in addition to the Phase objects is:

- MWs
- Vapor pressure curve
- Functioning enthalpy models for each phase

Parameters

constants [*ChemicalConstantsPackage* object] Package of chemical constants; these are used as boundaries at times, initial guesses other times, and in all cases these properties are accessible as attributes of the resulting *EquilibriumState* object, [-]

correlations [*PropertyCorrelationsPackage*] Package of chemical T-dependent properties; these are used as boundaries at times, for initial guesses other times, and in all cases these properties are accessible as attributes of the resulting *EquilibriumState* object, [-]

gas [*Phase* object] A single phase which can represent the gas phase, [-]

liquids [list[*Phase*]] A list of phase objects that can represent the liquid phases; if working with a VLL system with a consistent model, specify the same liquid phase twice; the length of this list is the maximum number of liquid phases that will be searched for, [-]

solids [list[*Phase*]] Not used, [-]

settings [*BulkSettings* object] Object containing settings for calculating bulk and transport properties, [-]

Notes

The algorithms in this object are mostly from [1], [2] and [3]. Sequential substitution without acceleration is used by default to converge multiphase systems.

Additional information that can be provided in the *ChemicalConstantsPackage* object and *PropertyCorrelationsPackage* object that may help convergence is:

- *Tc*, *Pc*, *omega*, *Tb*, and *atoms*
- Gas heat capacity correlations
- Liquid molar volume correlations
- Heat of vaporization correlations

References

[1], [2], [3]

Examples

A three-phase flash of butanol, water, and ethanol with the SRK EOS without BIPs:

```
>>> from thermo import ChemicalConstantsPackage, CEOSGas, CEOSLiquid, SRKMIX,␣
↪FlashVLN, PropertyCorrelationsPackage, HeatCapacityGas
>>> constants = ChemicalConstantsPackage(Tcs=[563.0, 647.14, 514.0], Pcs=[4414000.0,
↪ 22048320.0, 6137000.0], omegas=[0.59, 0.344, 0.635], MWs=[74.1216, 18.01528, 46.
↪06844], CASs=['71-36-3', '7732-18-5', '64-17-5'])
>>> properties = PropertyCorrelationsPackage(constants=constants,
...                                      HeatCapacityGases=[HeatCapacityGas(poly_
↪fit=(50.0, 1000.0, [-3.787200194613107e-20, 1.7692887427654656e-16, -3.
↪445247207129205e-13, 3.612771874320634e-10, -2.1953250181084466e-07, 7.
↪707135849197655e-05, -0.014658388538054169, 1.5642629364740657, -7.
↪614560475001724])),
...                                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [5.
↪543665000518528e-22, -2.403756749600872e-18, 4.2166477594350336e-15, -3.
↪7965208514613565e-12, 1.823547122838406e-09, -4.3747690853614695e-07, 5.
↪437938301211039e-05, -0.003220061088723078, 33.32731489750759])),
...                                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [-1.
↪162767978165682e-20, 5.4975285700787494e-17, -1.0861242757337942e-13, 1.
↪1582703354362728e-10, -7.160627710867427e-08, 2.5392014654765875e-05, -0.
↪004732593693568646, 0.5072291035198603, 20.037826650765965])),], )
>>> eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
>>> gas = CEOSGas(SRKMIX, eos_kwargs, HeatCapacityGases=properties.
↪HeatCapacityGases)
>>> liq = CEOSLiquid(SRKMIX, eos_kwargs, HeatCapacityGases=properties.
↪HeatCapacityGases)
>>> flashN = FlashVLN(constants, properties, liquids=[liq, liq], gas=gas)
>>> res = flashN.flash(T=361, P=1e5, zs=[.25, 0.7, .05])
>>> res.phase_count
3
```

Attributes

SS_NP_MAXITER [int] Maximum number of sequential substitution iterations to try when converging a three or more phase solution, [-]

SS_NP_TOL [float] Convergence tolerance in sequential substitution for a three or more phase solution [-]

SS_NP_TRIVIAL_TOL [float] Tolerance at which to quick a three-phase flash because it is converging to the trivial solution, [-]

SS_STAB_AQUEOUS_CHECK [bool] If True, the first three-phase stability check will be on water (if it is present) as it forms a three-phase solution more than any other component, [-]

DOUBLE_CHECK_2P [bool] This parameter should be set to True if any issues in the solution are noticed. It can slow down two-phase solution. It ensures that all potential vapor-liquid and liquid-liquid phase pairs are searched for stability, instead of testing first for a vapor-liquid solution and then moving on to a three phase flash if an instability is detected, [-]

Base Flash Class

class thermo.flash.**Flash**

Bases: object

Base class for performing flash calculations. All Flash objects need to inherit from this, and common methods can be added to it.

Methods

flash([zs, T, P, VF, SF, V, H, S, G, U, A, ...])	Method to perform a flash calculation and return the result as an *EquilibriumState* object.
plot_TP(zs[, Tmin, Tmax, pts, branches, ...])	Method to create a plot of the phase envelope as can be calculated from a series of temperature & vapor fraction spec flashes.

flash(*zs=None*, *T=None*, *P=None*, *VF=None*, *SF=None*, *V=None*, *H=None*, *S=None*, *G=None*, *U=None*, *A=None*, *solution=None*, *hot_start=None*, *retry=False*, *dest=None*)

Method to perform a flash calculation and return the result as an *EquilibriumState* object. This generic interface allows flashes with any combination of valid specifications; if a flash is unimplemented and error will be raised.

Parameters

zs [list[float], optional] Mole fractions of each component, required unless there is only one component, [-]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

VF [float, optional] Vapor fraction, [-]

SF [float, optional] Solid fraction, [-]

V [float, optional] Molar volume of the overall bulk, [m^3/mol]

H [float, optional] Molar enthalpy of the overall bulk, [J/mol]

S [float, optional] Molar entropy of the overall bulk, [J/(mol*K)]

G [float, optional] Molar Gibbs free energy of the overall bulk, [J/mol]

U [float, optional] Molar internal energy of the overall bulk, [J/mol]

A [float, optional] Molar Helmholtz energy of the overall bulk, [J/mol]

solution [str or int, optional] When multiple solutions exist, if more than one is found they will be sorted by T (and then P) increasingly; this number will index into the multiple solution array. Negative indexing is supported. 'high' is an alias for 0, and 'low' an alias for -1. Setting this parameter may make a flash slower because in some cases more checks are performed. [-]

hot_start [*EquilibriumState*] A previously converged flash or initial guessed state from which the flash can begin; this parameter can save time in some cases, [-]

retry [bool] Usually for flashes like UV or PH, there are multiple sets of possible iteration variables. For the UV case, the prefered iteration variable is P, so each iteration a PV solve is done on the phase; but equally the flash can be done iterating on *T*, where a TV solve is done on the phase each iteration. Depending on the tolerances, the flash type, the thermodynamic consistency of the phase, and other factors, it is possible the flash can fail. If *retry* is set to True, the alternate variable set will be iterated as a backup if the first flash fails. [-]

dest [None or `EquilibriumState` or `EquilibriumStream`] What type of object the flash result is set into; leave as None to obtain the normal *EquilibriumState* results, [-]

Returns

results [`EquilibriumState`] Equilibrium object containing the state of the phases after the flash calculation [-]

Notes

Warning: Not all flash specifications have a unique solution. Not all flash specifications will converge, whether from a bad model, bad inputs, or simply a lack of convergence by the implemented algorithms. You are welcome to submit these cases to the author but the library is provided AS IS, with NO SUPPORT.

Warning: Convergence of a flash may be impaired by providing *hot_start*. If reliability is desired, do not use this parameter.

Warning: The most likely thermodynamic methods to converge are thermodynamically consistent ones. This means e.g. an ideal liquid and an ideal gas; or an equation of state for both phases. Mixing thermodynamic models increases the possibility of multiple solutions, discontinuities, and other not-fun issues for the algorithms.

`plot_TP`(*zs*, *Tmin=None*, *Tmax=None*, *pts=50*, *branches=None*, *ignore_errors=True*, *values=False*, *show=True*, *hot=True*)

Method to create a plot of the phase envelope as can be calculated from a series of temperature & vapor fraction spec flashes. By default vapor fractions of 0 and 1 are plotted; additional vapor fraction specifications can be specified in the *branches* argument as a list.

Parameters

zs [list[float]] Mole fractions of the feed, [-]

Tmin [float, optional] Minimum temperature to begin the plot, [K]

Tmax [float, optional] Maximum temperature to end the plot, [K]

pts [int, optional] The number of points to calculated for each vapor fraction value, [-]

branches [list[float], optional] Extra vapor fraction values to plot, [-]

ignore_errors [bool, optional] Whether to fail on a calculation failure or to ignore the bad point, [-]

values [bool, optional] If True, the calculated values will be returned instead of plotted, [-]

show [bool, optional] If False, the plot will be returned instead of shown, [-]

hot [bool, optional] Whether to restart the next flash from the previous flash or not (intended to speed the call when True), [-]

Returns

Ts [list[float]] Temperatures, [K]

P_dews, P_bubbles, branch_Ps

7.13.2 Specific Flash Algorithms

It is recommended to use the Flash classes, which are designed to have generic interfaces. The implemented specific flash algorithms may be changed in the future, but reading their source code may be helpful for instructive purposes.

7.14 Functional Group Identification (thermo.functional_groups)

This module contains various methods for identifying functional groups in molecules. This functionality requires the RDKit library to work.

For submitting pull requests, please use the GitHub issue tracker.

7.14.1 Specific molecule matching functions

thermo.functional_groups.**is_organic**(*mol*, *restrict_atoms=None*, *organic_smiles=frozenset({'C', 'CO', 'NC(N)=O', 'O=C(OC(=O)C(F)(F)F)C(F)(F)F'})*, *inorganic_smiles=frozenset({'BrC(Br)(Br)Br', 'C#N', 'ClC(Cl)(Cl)Cl', 'FC(F)(F)F', 'IC(I)(I)I', 'O=C(Cl)Cl', 'O=C(F)F', 'O=C(O)O', 'O=C=O', 'O=C=S', 'S=C=S', '[C-]#[O+]'})*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is organic. The definition of organic vs. inorganic compounds is arabitrary. The rules implemented here are fairly complex.

- If a compound has an C-C bond, a C=C bond, a carbon triple bond, a carbon attatched however to a hydrogen, a carbon in a ring, or an amide group.

- If a compound is in the list of canonical smiles *organic_smiles*, either the defaults in the library or those provided as an input to the function, the molecule is considered organic.
- If a compound is in the list of canonical smiles *inorganic_smiles*, either the defaults in the library or those provided as an input to the function, the molecule is considered inorganic.
- If *restrict_atoms* is provided and atoms are present in the molecule that are restricted, the compound is considered restricted.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

restrict_atoms [Iterable[str]] Atoms that cannot be found in an organic molecule, [-]

organic_smiles [Iterable[str]] Smiles that are hardcoded to be organic, [-]

inorganic_smiles [Iterable[str]] Smiles that are hardcoded to be inorganic, [-]

Returns

is_organic [bool] Whether or not the compound is a organic or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_organic(MolFromSmiles("CC(C)C(C)C(C)C"))
True
```

thermo.functional_groups.**is_inorganic**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is inorganic.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_inorganic [bool] Whether or not the compound is inorganic or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_inorganic(MolFromSmiles("O=[Zr].Cl.Cl"))
True
```

7.14.2 Hydrocarbon Groups

thermo.functional_groups.**is_alkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an alkane, also refered to as a paraffin. All bonds in the molecule must be single carbon-carbon or carbon-hydrogen.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alkane [bool] Whether or not the compound is an alkane or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alkane(MolFromSmiles("CCC"))
True
```

thermo.functional_groups.**is_cycloalkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a cycloalkane, also refered to as a naphthenes.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_cycloalkane [bool] Whether or not the compound is a cycloalkane or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_cycloalkane(MolFromSmiles('C1CCCCCCCCC1'))
True
```

thermo.functional_groups.**is_branched_alkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a branched alkane, also refered to as an isoparaffin. All bonds in the molecule must be single carbon-carbon or carbon-hydrogen.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_branched_alkane [bool] Whether or not the compound is a branched alkane or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_branched_alkane(MolFromSmiles("CC(C)C(C)C(C)C"))
True
```

thermo.functional_groups.**is_alkene**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an alkene. Alkenes are also refered to as olefins.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alkene [bool] Whether or not the compound is a alkene or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alkene(MolFromSmiles('C=C'))
True
```

thermo.functional_groups.**is_alkyne**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an alkyne.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alkyne [bool] Whether or not the compound is a alkyne or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alkyne(MolFromSmiles('CC#C'))
True
```

thermo.functional_groups.**is_aromatic**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is aromatic.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_aromatic [bool] Whether or not the compound is aromatic or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_aromatic(MolFromSmiles('CC1=CC=CC=C1C'))
True
```

7.14.3 Oxygen Groups

thermo.functional_groups.**is_alcohol**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule any alcohol functional groups.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alcohol [bool] Whether or not the compound is an alcohol, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alcohol(MolFromSmiles('CCO'))
True
```

`thermo.functional_groups.is_polyol(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a polyol (more than 1 alcohol functional groups).

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_polyol [bool] Whether or not the compound is a polyol, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_polyol(MolFromSmiles('C(C(CO)O)O'))
True
```

`thermo.functional_groups.is_ketone(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a ketone.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_ketone [bool] Whether or not the compound is a ketone, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_ketone(MolFromSmiles('C1CCC(=O)CC1'))
True
```

`thermo.functional_groups.is_aldehyde(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an aldehyde.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_aldehyde [bool] Whether or not the compound is an aldehyde, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_aldehyde(MolFromSmiles('C=O'))
True
```

thermo.functional_groups.**is_carboxylic_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carboxylic acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carboxylic_acid [bool] Whether or not the compound is a carboxylic acid, [-].

Examples

Butyric acid (butter)

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carboxylic_acid(MolFromSmiles('CCCC(=O)O'))
True
```

thermo.functional_groups.**is_ether**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an ether.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_ether [bool] Whether or not the compound is an ether, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_ether(MolFromSmiles('CC(C)OC(C)C'))
True
```

thermo.functional_groups.**is_phenol**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a phenol.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_phenol [bool] Whether or not the compound is a phenol, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_phenol(MolFromSmiles('CC(=O)NC1=CC=C(C=C1)O'))
True
```

thermo.functional_groups.**is_ester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an ester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_ester [bool] Whether or not the compound is an ester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_ester(MolFromSmiles('CCOC(=O)C'))
True
```

thermo.functional_groups.**is_anhydride**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an anhydride.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_anhydride [bool] Whether or not the compound is an anhydride, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_anhydride(MolFromSmiles('C1=CC(=O)OC1=O'))
True
```

thermo.functional_groups.**is_acyl_halide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a acyl halide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_acyl_halide [bool] Whether or not the compound is a acyl halide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_acyl_halide(MolFromSmiles('C(CCC(=O)Cl)CC(=O)Cl'))
True
```

thermo.functional_groups.**is_carbonate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carbonate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carbonate [bool] Whether or not the compound is a carbonate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carbonate(MolFromSmiles('C(=O)(OC(Cl)(Cl)Cl)OC(Cl)(Cl)Cl'))
True
```

thermo.functional_groups.**is_carboxylate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carboxylate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carboxylate [bool] Whether or not the compound is a carboxylate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carboxylate(MolFromSmiles('CC(=O)[O-].[Na+]'))
True
```

thermo.functional_groups.**is_hydroperoxide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a hydroperoxide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_hydroperoxide [bool] Whether or not the compound is a hydroperoxide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_hydroperoxide(MolFromSmiles('CC(C)(C)OO'))
True
```

thermo.functional_groups.**is_peroxide**(*mol*)
Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a peroxide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_peroxide [bool] Whether or not the compound is a peroxide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_peroxide(MolFromSmiles('CC(C)(C)OOC(C)(C)C'))
True
```

thermo.functional_groups.**is_orthoester**(*mol*)
Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a orthoester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_orthoester [bool] Whether or not the compound is a orthoester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_orthoester(MolFromSmiles('CCOC(C)(OCC)OCC'))
True
```

thermo.functional_groups.**is_methylenedioxy**(*mol*)
Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a methylenedioxy.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_methylenedioxy [bool] Whether or not the compound is a methylenedioxy, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_methylenedioxy(MolFromSmiles('C1OC2=CC=CC=C2O1'))
True
```

thermo.functional_groups.**is_orthocarbonate_ester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a orthocarbonate ester .

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_orthocarbonate_ester [bool] Whether or not the compound is a orthocarbonate ester , [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_orthocarbonate_ester (MolFromSmiles('COC(OC)(OC)OC')
True
```

thermo.functional_groups.**is_carboxylic_anhydride**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carboxylic anhydride .

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carboxylic_anhydride [bool] Whether or not the compound is a carboxylic anhydride, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carboxylic_anhydride (MolFromSmiles('CCCC(=O)OC(=O)CCC')
True
```

7.14.4 Nitrogen Groups

thermo.functional_groups.**is_amide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule has a amide RC(=O)NRR group.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_amide [bool] Whether or not the compound is a amide or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_amide(MolFromSmiles('CN(C)C=O'))
True
```

thermo.functional_groups.**is_amidine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule has a amidine RC(NR)NR2 group.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_amidine [bool] Whether or not the compound is a amidine or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_amidine(MolFromSmiles('C1=CC(=CC=C1C(=N)N)OCCCCCOC2=CC=C(C=C2)C(=N)N'))
True
```

thermo.functional_groups.**is_amine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a amine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_amine [bool] Whether or not the compound is a amine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_amine(MolFromSmiles('CN'))
True
```

thermo.functional_groups.**is_primary_amine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a primary amine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_primary_amine [bool] Whether or not the compound is a primary amine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_primary_amine(MolFromSmiles('CN'))
True
```

thermo.functional_groups.**is_secondary_amine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a secondary amine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_secondary_amine [bool] Whether or not the compound is a secondary amine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_secondary_amine(MolFromSmiles('CNC'))
True
```

thermo.functional_groups.**is_tertiary_amine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a tertiary amine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_tertiary_amine [bool] Whether or not the compound is a tertiary amine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_tertiary_amine(MolFromSmiles('CN(C)C'))
True
```

thermo.functional_groups.**is_quat**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a quat.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_quat [bool] Whether or not the compound is a quat, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_quat(MolFromSmiles('CCCCCCCCCCCCCCCCCC[N+](C)(C)CCCCCCCCCCCCCCCCCC.[Cl-]'))
True
```

thermo.functional_groups.**is_imine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a imine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_imine [bool] Whether or not the compound is a imine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_imine(MolFromSmiles('C1=CC=C(C=C1)C(=N)C2=CC=CC=C2'))
True
```

thermo.functional_groups.**is_primary_ketimine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a primary ketimine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_primary_ketimine [bool] Whether or not the compound is a primary ketimine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_primary_ketimine(MolFromSmiles('C1=CC=C(C=C1)C(=N)C2=CC=CC=C2'))
True
```

thermo.functional_groups.**is_secondary_ketimine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a secondary ketimine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_secondary_ketimine [bool] Whether or not the compound is a secondary ketimine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_secondary_ketimine(MolFromSmiles(
↪'CC(C)CC(=NC1=CC=C(C=C1)CC2=CC=C(C=C2)N=C(C)CC(C)C)C'))
True
```

thermo.functional_groups.**is_primary_aldimine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a primary aldimine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_primary_aldimine [bool] Whether or not the compound is a primary aldimine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_primary_aldimine(MolFromSmiles('CC=N'))
True
```

thermo.functional_groups.**is_secondary_aldimine**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a secondary aldimine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_secondary_aldimine [bool] Whether or not the compound is a secondary aldimine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_secondary_aldimine(MolFromSmiles( 'C1=CC=C(C=C1)/C=N\\O'))
True
```

thermo.functional_groups.**is_imide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a imide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_imide [bool] Whether or not the compound is a imide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_imide(MolFromSmiles('C1=CC=C2C(=C1)C(=O)NC2=O'))
True
```

thermo.functional_groups.**is_azide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a azide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_azide [bool] Whether or not the compound is a azide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_azide(MolFromSmiles('C1=CC=C(C=C1)N=[N+]=[N-]'))
True
```

thermo.functional_groups.**is_azo**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a azo.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_azo [bool] Whether or not the compound is a azo, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_azo(MolFromSmiles('C1=CC=C(C=C1)N=NC2=CC=CC=C2'))
True
```

thermo.functional_groups.**is_cyanate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a cyanate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_cyanate [bool] Whether or not the compound is a cyanate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_cyanate(MolFromSmiles('COC#N'))
True
```

thermo.functional_groups.**is_isocyanate**(*mol*)
Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a isocyanate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_isocyanate [bool] Whether or not the compound is a isocyanate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_isocyanate(MolFromSmiles('CN=C=O'))
True
```

thermo.functional_groups.**is_nitrate**(*mol*)
Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a nitrate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_nitrate [bool] Whether or not the compound is a nitrate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_nitrate(MolFromSmiles('CCCCCO[N+](=O)[O-]'))
True
```

thermo.functional_groups.**is_nitrile**(*mol*)
Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a nitrile.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_nitrile [bool] Whether or not the compound is a nitrile, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_nitrile(MolFromSmiles('CC#N'))
True
```

`thermo.functional_groups.is_isonitrile(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a isonitrile.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_isonitrile [bool] Whether or not the compound is a isonitrile, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_isonitrile(MolFromSmiles('C[N+]#[C-]'))
True
```

`thermo.functional_groups.is_nitrite(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a nitrite.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_nitrite [bool] Whether or not the compound is a nitrite, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_nitrite(MolFromSmiles('CC(C)CCON=O'))
True
```

`thermo.functional_groups.is_nitro(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a nitro.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_nitro [bool] Whether or not the compound is a nitro, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_nitro(MolFromSmiles('C[N+](=O)[O-]'))
True
```

thermo.functional_groups.**is_nitroso**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a nitroso.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_nitroso [bool] Whether or not the compound is a nitroso, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_nitroso(MolFromSmiles('C1=CC=C(C=C1)N=O'))
True
```

thermo.functional_groups.**is_oxime**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a oxime.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_oxime [bool] Whether or not the compound is a oxime, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_oxime(MolFromSmiles('CC(=NO)C'))
True
```

thermo.functional_groups.**is_pyridyl**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a pyridyl.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_pyridyl [bool] Whether or not the compound is a pyridyl, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_pyridyl(MolFromSmiles('CN1CCC[C@H]1C1=CC=CN=C1'))
True
```

thermo.functional_groups.**is_carbamate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carbamate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carbamate [bool] Whether or not the compound is a carbamate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carbamate(MolFromSmiles('CC(C)OC(=O)NC1=CC(=CC=C1)Cl'))
True
```

7.14.5 Sulfur Groups

thermo.functional_groups.**is_mercaptan**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule has a mercaptan R-SH group. This is also called a thiol.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_mercaptan [bool] Whether or not the compound is a mercaptan or not, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_mercaptan(MolFromSmiles("CS"))
True
```

thermo.functional_groups.**is_sulfide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a sulfide. This group excludes disulfides.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_sulfide [bool] Whether or not the compound is a sulfide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_sulfide(MolFromSmiles('CSC'))
True
```

thermo.functional_groups.**is_disulfide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a disulfide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_disulfide [bool] Whether or not the compound is a disulfide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_disulfide(MolFromSmiles('CSSC'))
True
```

thermo.functional_groups.**is_sulfoxide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a sulfoxide.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_sulfoxide [bool] Whether or not the compound is a sulfoxide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_sulfoxide(MolFromSmiles('CS(=O)C'))
True
```

thermo.functional_groups.**is_sulfone**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a sulfone.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_sulfone [bool] Whether or not the compound is a sulfone, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_sulfone(MolFromSmiles('CS(=O)(=O)C'))
True
```

thermo.functional_groups.**is_sulfinic_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a sulfinic acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_sulfinic_acid [bool] Whether or not the compound is a sulfinic acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_sulfinic_acid(MolFromSmiles('O=S(O)CCN'))
True
```

thermo.functional_groups.**is_sulfonic_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a sulfonic acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_sulfonic_acid [bool] Whether or not the compound is a sulfonic acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_sulfonic_acid(MolFromSmiles('OS(=O)(=O)c1ccccc1'))
True
```

thermo.functional_groups.**is_sulfonate_ester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a sulfonate ester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_sulfonate_ester [bool] Whether or not the compound is a sulfonate ester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_sulfonate_ester(MolFromSmiles('COS(=O)(=O)C(F)(F)F'))
True
```

thermo.functional_groups.**is_thiocyanate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a thiocyanate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_thiocyanate [bool] Whether or not the compound is a thiocyanate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_thiocyanate(MolFromSmiles('C1=CC=C(C=C1)SC#N'))
True
```

thermo.functional_groups.**is_isothiocyanate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a isothiocyanate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_isothiocyanate [bool] Whether or not the compound is a isothiocyanate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_isothiocyanate(MolFromSmiles('C=CCN=C=S'))
True
```

thermo.functional_groups.**is_thioketone**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a thioketone.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_thioketone [bool] Whether or not the compound is a thioketone, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_thioketone(MolFromSmiles('C1=CC=C(C=C1)C(=S)C2=CC=CC=C2'))
True
```

thermo.functional_groups.**is_thial**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a thial.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_thial [bool] Whether or not the compound is a thial, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_thial(MolFromSmiles('CC=S'))
True
```

thermo.functional_groups.**is_carbothioic_s_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a Carbothioic S-acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carbothioic_s_acid [bool] Whether or not the compound is a Carbothioic S-acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carbothioic_s_acid(MolFromSmiles('C1=CC=C(C=C1)C(=O)S'))
True
```

thermo.functional_groups.**is_carbothioic_o_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a Carbothioic S-acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carbothioic_o_acid [bool] Whether or not the compound is a Carbothioic S-acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carbothioic_o_acid(MolFromSmiles('OC(=S)c1ccccc1O'))
True
```

thermo.functional_groups.**is_thiolester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a thiolester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_thiolester [bool] Whether or not the compound is a thiolester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_thiolester(MolFromSmiles('CSC(=O)C=C'))
True
```

thermo.functional_groups.**is_thionoester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a thionoester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_thionoester [bool] Whether or not the compound is a thionoester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_thionoester(MolFromSmiles('CCOC(=S)S'))
True
```

thermo.functional_groups.**is_carbodithioic_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carbodithioic acid .

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carbodithioic_acid [bool] Whether or not the compound is a carbodithioic acid , [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carbodithioic_acid(MolFromSmiles('C1=CC=C(C=C1)C(=S)S'))
True
```

thermo.functional_groups.**is_carbodithio**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a carbodithio.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_carbodithio [bool] Whether or not the compound is a carbodithio, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_carbodithio(MolFromSmiles('C(=S)(N)SSC(=S)N'))
True
```

7.14.6 Silicon Groups

thermo.functional_groups.**is_siloxane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a siloxane.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_siloxane [bool] Whether or not the compound is a siloxane, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_siloxane(MolFromSmiles('C[Si]1(O[Si](O[Si](O[Si](O1)(C)C)(C)C)(C)C)C'))
True
```

thermo.functional_groups.**is_silyl_ether**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule any silyl ether functional groups.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_silyl_ether [bool] Whether or not the compound is an silyl ether, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_silyl_ether(MolFromSmiles('C[Si](C)(C)OS(=O)(=O)C(F)(F)F'))
True
```

7.14.7 Boron Groups

thermo.functional_groups.**is_boronic_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule has any boronic acid functional groups.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_boronic_acid [bool] Whether or not the compound is an boronic acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_boronic_acid(MolFromSmiles('B(C)(O)O'))
True
```

thermo.functional_groups.**is_boronic_ester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a boronic ester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_boronic_ester [bool] Whether or not the compound is a boronic ester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_boronic_ester(MolFromSmiles('B(C)(OC(C)C)OC(C)C'))
True
```

thermo.functional_groups.**is_borinic_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a borinic acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_borinic_acid [bool] Whether or not the compound is a borinic acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_borinic_acid(MolFromSmiles('BO'))
True
```

`thermo.functional_groups.is_borinic_ester(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a borinic ester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_borinic_ester [bool] Whether or not the compound is a borinic ester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_borinic_ester(MolFromSmiles('B(C1=CC=CC=C1)(C2=CC=CC=C2)OCCN'))
True
```

7.14.8 Phosphorus Groups

`thermo.functional_groups.is_phosphine(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a phosphine.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_phosphine [bool] Whether or not the compound is a phosphine, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_phosphine(MolFromSmiles('CCCPC'))
True
```

`thermo.functional_groups.is_phosphonic_acid(mol)`

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a phosphonic_acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_phosphonic_acid [bool] Whether or not the compound is a phosphonic_acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_phosphonic_acid(MolFromSmiles('C1=CC=C(C=C1)CP(=O)(O)O'))
True
```

thermo.functional_groups.**is_phosphodiester**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a phosphodiester.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_phosphodiester [bool] Whether or not the compound is a phosphodiester, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_phosphodiester(MolFromSmiles('C(COP(=O)(O)OCC(C(=O)O)N)N=C(N)N'))
True
```

thermo.functional_groups.**is_phosphate**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a phosphate.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_phosphate [bool] Whether or not the compound is a phosphate, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_phosphate(MolFromSmiles(
↪'C1=CN(C(=O)N=C1N)[C@H]2[C@@H]([C@@H]([C@H](O2)COP(=O)(O)OP(=O)(O)OP(=O)(O)O)O)O
↪'))
True
```

7.14.9 Halogen Groups

thermo.functional_groups.**is_haloalkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a haloalkane.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_haloalkane [bool] Whether or not the compound is a haloalkane, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_haloalkane(MolFromSmiles('CCCl'))
True
```

thermo.functional_groups.**is_fluoroalkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a fluoroalkane.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_fluoroalkane [bool] Whether or not the compound is a fluoroalkane, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_fluoroalkane(MolFromSmiles('CF'))
True
```

thermo.functional_groups.**is_chloroalkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a chloroalkane.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_chloroalkane [bool] Whether or not the compound is a chloroalkane, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_chloroalkane(MolFromSmiles('CCl'))
True
```

thermo.functional_groups.**is_bromoalkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a bromoalkane.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_bromoalkane [bool] Whether or not the compound is a bromoalkane, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_bromoalkane(MolFromSmiles('CBr'))
True
```

thermo.functional_groups.**is_iodoalkane**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is a iodoalkane.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_iodoalkane [bool] Whether or not the compound is a iodoalkane, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_iodoalkane(MolFromSmiles('CI'))
True
```

7.14.10 Organometalic Groups

thermo.functional_groups.**is_alkyllithium**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule any alkyllithium functional groups.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alkyllithium [bool] Whether or not the compound is an alkyllithium, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alkyllithium(MolFromSmiles('[Li+].[CH3-]'))
True
```

thermo.functional_groups.**is_alkylaluminium**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule any alkylaluminium functional groups.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alkylaluminium [bool] Whether or not the compound is an alkylaluminium, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alkylaluminium(MolFromSmiles('CC[Al](CC)CC'))
True
```

thermo.functional_groups.**is_alkylmagnesium_halide**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule any alkylmagnesium_halide functional groups.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_alkylmagnesium_halide [bool] Whether or not the compound is an alkylmagnesium_halide, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_alkylmagnesium_halide(MolFromSmiles('C1=CC=[C-]C=C1.[Mg+2].[Br-]'))
True
```

7.14.11 Other Groups

thermo.functional_groups.**is_acid**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, returns whether or not the molecule is an acid.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

is_acid [bool] Whether or not the compound is a acid, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> is_acid(MolFromSmiles('CC(=O)O'))
True
```

7.14.12 Utility functions

thermo.functional_groups.**count_ring_ring_attatchments**(*mol*)

Given a *rdkit.Chem.rdchem.Mol* object, count the number of times a ring in the molecule is bonded with another ring in the molecule.

An easy explanation is cubane - each edge of the cube is a ring uniquely bonding with another ring; so this function returns twelve.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

Returns

ring_ring_attatchments [bool] The number of ring-ring bonds, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> count_ring_ring_attatchments(MolFromSmiles('C12C3C4C1C5C2C3C45'))
12
```

thermo.functional_groups.**count_rings_attatched_to_rings**(*mol*, *allow_neighbors=True*, *atom_rings=None*)

Given a *rdkit.Chem.rdchem.Mol* object, count the number of rings in the molecule that are attatched to another ring. if *allow_neighbors* is True, any bond to another atom that is part of a ring is allowed; if it is False, the rings have to share a wall.

Parameters

mol [rdkit.Chem.rdchem.Mol] Molecule [-]

allow_neighbors [bool] Whether or not to count neighboring rings or just ones sharing a wall, [-]

atom_rings [rdkit.Chem.rdchem.RingInfo, optional] Internal parameter, used for performance only

Returns

rings_attatched_to_rings [bool] The number of rings bonded to other rings, [-].

Examples

```
>>> from rdkit.Chem import MolFromSmiles
>>> count_rings_attatched_to_rings(MolFromSmiles('C12C3C4C1C5C2C3C45'))
6
```

7.14.13 Functions using group identification

`thermo.functional_groups.BVirial_Tsonopoulos_extended_ab(Tc, Pc, dipole, smiles)`

Calculates the of *a* and *b* parameters of the Tsonopoulos (extended) second virial coefficient prediction method. These parameters account for polarity. This function uses *rdkit* to identify the component type of the molecule.

Parameters

Tc [float] Critical temperature of fluid [K]

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

dipole [float] dipole moment, optional, [Debye]

Returns

a [float] Fit parameter matched to one of the supported chemical classes.

b [float] Fit parameter matched to one of the supported chemical classes.

Notes

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:

$$\mu_r = 100000\frac{\mu^2(Pc/101325.0)}{Tc^2}$$

References

[1], [2]

7.15 Heat Capacity (thermo.heat_capacity)

This module contains implementations of *TDependentProperty* representing liquid, vapor, and solid heat capacity. A variety of estimation and data methods are available as included in the *chemicals* library. Additionally liquid, vapor, and solid mixture heat capacity predictor objects are implemented subclassing *MixtureProperty*.

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

7.15.1 Pure Liquid Heat Capacity

class thermo.heat_capacity.**HeatCapacityLiquid**(*CASRN=''*, *MW=None*, *similarity_variable=None*, *Tc=None*, *omega=None*, *Cpgm=None*, *extrapolation='linear'*, ***kwargs*)

Bases: `thermo.utils.t_dependent_property.TDependentProperty`

Class for dealing with liquid heat capacity as a function of temperature. Consists of seven coefficient-based methods, two constant methods, one tabular source, two CSP methods based on gas heat capacity, one simple estimator, and the external library CoolProp.

Parameters

CASRN [str, optional] The CAS number of the chemical

MW [float, optional] Molecular weight, [g/mol]

similarity_variable [float, optional] similarity variable, n_atoms/MW, [mol/g]

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

omega [float, optional] Acentric factor, [-]

Cpgm [float or callable, optional] Idea-gas molar heat capacity at T or callable for the same, [J/mol/K]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

chemicals.heat_capacity.Zabransky_quasi_polynomial

chemicals.heat_capacity.Zabransky_cubic

chemicals.heat_capacity.Rowlinson_Poling

chemicals.heat_capacity.Rowlinson_Bondi

chemicals.heat_capacity.Dadgostar_Shaw

chemicals.heat_capacity.Shomate

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the list stored in *heat_capacity_liquid_methods*.

ZABRANSKY_SPLINE, ZABRANSKY_QUASIPOLYNOMIAL, ZABRANSKY_SPLINE_C, and ZABRANSKY_QUASIPOLYNOMIAL_C:

Rigorous expressions developed in [1] following critical evaluation of the available data. The spline methods use the form described in Zabransky_cubic over short ranges with varying coefficients to obtain a wider range. The quasi-polynomial methods use the form described in Zabransky_quasi_polynomial, more suitable for extrapolation, and over then entire range. Respectively, there is data available for 588, 146, 51, and 26 chemicals. 'C' denotes constant- pressure data available from more precise experiments. The others are heat capacity values averaged over a temperature changed.

ZABRANSKY_SPLINE_SAT and ZABRANSKY_QUASIPOLYNOMIAL_SAT:

Rigorous expressions developed in [1] following critical evaluation of the available data. The spline method use the form described in Zabransky_cubic over short ranges with varying coefficients to obtain a wider range. The quasi-polynomial method use the form described in Zabransky_quasi_polynomial, more suitable for extrapolation, and over their entire range. Respectively, there is data available for 203, and 16 chemicals. Note that these methods are for the saturation curve!

VDI_TABULAR:

Tabular data up to the critical point available in [5]. Note that this data is along the saturation curve.

ROWLINSON_POLING:

CSP method described in Rowlinson_Poling. Requires a ideal gas heat capacity value at the same temperature as it is to be calculated.

ROWLINSON_BONDI:

CSP method described in Rowlinson_Bondi. Requires a ideal gas heat capacity value at the same temperature as it is to be calculated.

COOLPROP:

CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [3]. Very slow.

DADGOSTAR_SHAW:

A basic estimation method using the *similarity variable* concept; requires only molecular structure, so is very convenient. See Dadgostar_Shaw for details.

POLING_CONST:

Constant values in [2] at 298.15 K; available for 245 liquids.

CRCSTD:

Constant values tabulated in [4] at 298.15 K; data is available for 433 liquids.

WEBBOOK_SHOMATE: Shomate form coefficients from [6] for ~200 compounds.

References

[1], [2], [3], [4], [5], [6]

Examples

```
>>> CpLiquid = HeatCapacityLiquid(CASRN='142-82-5', MW=100.2, similarity_variable=0.
↪2295, Tc=540.2, omega=0.3457, Cpgm=165.2)
```

Methods

`calculate`(T, method)	Method to calculate heat capacity of a liquid at temperature T with a given method.
`test_method_validity`(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)

Method to calculate heat capacity of a liquid at temperature T with a given method.

This method has no exception handling; see `T_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate heat capacity, [K]

method [str] Name of the method to use

Returns

Cp [float] Heat capacity of the liquid at T, [J/mol/K]

name = 'Liquid heat capacity'

property_max = 10000.0

Maximum valid of Heat capacity; arbitrarily set. For fluids very near the critical point, this value can be obscenely high.

property_min = 1

Allow very low heat capacities; arbitrarily set; liquid heat capacity should always be somewhat substantial.

ranked_methods = ['ZABRANSKY_SPLINE', 'ZABRANSKY_QUASIPOLYNOMIAL', 'ZABRANSKY_SPLINE_C', 'ZABRANSKY_QUASIPOLYNOMIAL_C', 'ZABRANSKY_SPLINE_SAT', 'ZABRANSKY_QUASIPOLYNOMIAL_SAT', 'WEBBOOK_SHOMATE', 'VDI_TABULAR', 'COOLPROP', 'DADGOSTAR_SHAW', 'ROWLINSON_POLING', 'ROWLINSON_BONDI', 'POLING_CONST', 'CRCSTD']

Default rankings of the available methods.

test_method_validity(*T*, *method*)

Method to check the validity of a method. Follows the given ranges for all coefficient-based methods.

For the CSP method `Rowlinson_Poling`, the model is considered valid for all temperatures. The simple method `Dadgostar_Shaw` is considered valid for all temperatures. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'J/mol/K'

thermo.heat_capacity.**heat_capacity_liquid_methods = ['ZABRANSKY_SPLINE', 'ZABRANSKY_QUASIPOLYNOMIAL', 'ZABRANSKY_SPLINE_C', 'ZABRANSKY_QUASIPOLYNOMIAL_C', 'ZABRANSKY_SPLINE_SAT', 'ZABRANSKY_QUASIPOLYNOMIAL_SAT', 'WEBBOOK_SHOMATE', 'VDI_TABULAR', 'ROWLINSON_POLING', 'ROWLINSON_BONDI', 'COOLPROP', 'DADGOSTAR_SHAW', 'POLING_CONST', 'CRCSTD']**

Holds all methods available for the `HeatCapacityLiquid class`, for use in iterating over them.

7.15.2 Pure Gas Heat Capacity

class thermo.heat_capacity.**HeatCapacityGas**(*CASRN=''*, *MW=None*, *similarity_variable=None*, *extrapolation='linear'*, *iscyclic_aliphatic=False*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with gas heat capacity as a function of temperature. Consists of three coefficient-based methods, two constant methods, one tabular source, one simple estimator, one group-contribution estimator, one component specific method, and the external library CoolProp.

Parameters

CASRN [str, optional] The CAS number of the chemical

MW [float, optional] Molecular weight, [g/mol]

similarity_variable [float, optional] similarity variable, n_atoms/MW, [mol/g]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.heat_capacity.TRCCp`

`chemicals.heat_capacity.Shomate`

`chemicals.heat_capacity.Lastovka_Shaw`

`chemicals.heat_capacity.Rowlinson_Poling` `chemicals.heat_capacity.Rowlinson_Bondi`

`thermo.joback.Joback`

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the list stored in *heat_capacity_gas_methods*.

TRCIG: A rigorous expression derived in [1] for modeling gas heat capacity. Coefficients for 1961 chemicals are available.

POLING_POLY: Simple polynomials in [2] not suitable for extrapolation. Data is available for 308 chemicals.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [3]. The heat capacity and enthalpy are implemented analytically and fairly fast; the entropy integral has no analytical integral and so is numerical. CoolProp's amazing coefficient collection is used directly in Python.

LASTOVKA_SHAW: A basic estimation method using the *similarity variable* concept; requires only molecular structure, so is very convenient. See `Lastovka_Shaw` for details.

CRCSTD: Constant values tabulated in [4] at 298.15 K; data is available for 533 gases.

POLING_CONST: Constant values in [2] at 298.15 K; available for 348 gases.

VDI_TABULAR: Tabular data up to the critical point available in [5]. Note that this data is along the saturation curve.

WEBBOOK_SHOMATE: Shomate form coefficients from [6] for ~700 compounds.

JOBACK: An estimation method for organic substances in [7]

References

[1], [2], [3], [4], [5], [6], [7]

Examples

```
>>> CpGas = HeatCapacityGas(CASRN='142-82-5', MW=100.2, similarity_variable=0.2295)
>>> CpGas(700)
317.244
```

Methods

calculate(T, method)	Method to calculate surface tension of a liquid at temperature T with a given method.
test_method_validity(T, method)	Method to test the validity of a specified method for a given temperature.

calculate(*T*, *method*)

Method to calculate surface tension of a liquid at temperature T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate heat capacity, [K]

method [str] Method name to use

Returns

Cp [float] Calculated heat capacity, [J/mol/K]

name = 'gas heat capacity'

property_max = 10000.0
Maximum valid of Heat capacity; arbitrarily set. For fluids very near the critical point, this value can be obscenely high.

property_min = 0
Heat capacities have a minimum value of 0 at 0 K.

ranked_methods = ['TRCIG', 'WEBBOOK_SHOMATE', 'POLING_POLY', 'COOLPROP', 'JOBACK', 'LASTOVKA_SHAW', 'CRCSTD', 'POLING_CONST', 'VDI_TABULAR']
Default rankings of the available methods.

test_method_validity(*T*, *method*)
Method to test the validity of a specified method for a given temperature.

'TRC' and 'Poling' both have minimum and maimum temperatures. The constant temperatures in POLING_CONST and CRCSTD are considered valid for 50 degrees around their specified temperatures. `Lastovka_Shaw` is considered valid for the whole range of temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to determine the validity of the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a specifid method is valid

units = 'J/mol/K'

thermo.heat_capacity.**heat_capacity_gas_methods = ['COOLPROP', 'TRCIG', 'WEBBOOK_SHOMATE', 'POLING_POLY', 'LASTOVKA_SHAW', 'CRCSTD', 'POLING_CONST', 'JOBACK', 'VDI_TABULAR']**
Holds all methods available for the *HeatCapacityGas* class, for use in iterating over them.

7.15.3 Pure Solid Heat Capacity

class thermo.heat_capacity.**HeatCapacitySolid**(*CASRN=''*, *similarity_variable=None*, *MW=None*, *extrapolation='linear'*, ***kwargs*)
Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with solid heat capacity as a function of temperature. Consists of two temperature-dependent expressions, one constant value source, and one simple estimator.

Parameters

similarity_variable [float, optional] similarity variable, n_atoms/MW, [mol/g]

MW [float, optional] Molecular weight, [g/mol]

CASRN [str, optional] The CAS number of the chemical

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.heat_capacity.Lastovka_solid`

`chemicals.heat_capacity.Shomate`

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the list stored in *heat_capacity_solid_methods*.

PERRY151: Simple polynomials with vaious exponents selected for each expression. Coefficients are in units of calories/mol/K. The full expression is:

$$Cp = a + bT + c/T^2 + dT^2$$

Data is available for 284 solids, from [2].

CRCSTD: Values tabulated in [1] at 298.15 K; data is available for 529 solids.

LASTOVKA_S: A basic estimation method using the *similarity variable* concept; requires only molecular structure, so is very convenient. See `Lastovka_solid` for details.

WEBBOOK_SHOMATE: Shomate form coefficients from [3] for ~300 compounds.

References

[1], [2], [3]

Examples

```
>>> CpSolid = HeatCapacitySolid(CASRN='142-82-5', MW=100.2, similarity_variable=0.
↪2295)
>>> CpSolid(200)
131.205824
```

Methods

`calculate`(T, method)	Method to calculate heat capacity of a solid at temperature T with a given method.
`test_method_validity`(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)

Method to calculate heat capacity of a solid at temperature T with a given method.

This method has no exception handling; see `T_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate heat capacity, [K]

method [str] Name of the method to use

Returns

Cp [float] Heat capacity of the solid at T, [J/mol/K]

name = 'solid heat capacity'

property_max = 10000.0

Maximum value of Heat capacity; arbitrarily set.

property_min = 0

Heat capacities have a minimum value of 0 at 0 K.

ranked_methods = ['WEBBOOK_SHOMATE', 'PERRY151', 'CRCSTD', 'LASTOVKA_S']

Default rankings of the available methods.

test_method_validity(*T*, *method*)

Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures. For the `Lastovka_solid` method, it is considered valid under 10000K.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'J/mol/K'

thermo.heat_capacity.**heat_capacity_solid_methods = ['WEBBOOK_SHOMATE', 'PERRY151', 'CRCSTD', 'LASTOVKA_S']**

Holds all methods available for the `HeatCapacitySolid` class, for use in iterating over them.

7.15.4 Mixture Liquid Heat Capacity

class thermo.heat_capacity.**HeatCapacityLiquidMixture**(*MWs=[]*, *CASs=[]*, *HeatCapacityLiquids=[]*)

Bases: *thermo.utils.mixture_property.MixtureProperty*

Class for dealing with liquid heat capacity of a mixture as a function of temperature, pressure, and composition. Consists only of mole weighted averaging, and the Laliberte method for aqueous electrolyte solutions.

Parameters

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

CASs [str, optional] The CAS numbers of all species in the mixture

HeatCapacityLiquids [list[HeatCapacityLiquid], optional] HeatCapacityLiquid objects created for all species in the mixture [-]

Notes

To iterate over all methods, use the list stored in *heat_capacity_liquid_mixture_methods*.

LALIBERTE: Electrolyte model equation with coefficients; see *thermo.electrochem.Laliberte_heat_capacity* for more details.

LINEAR: Mixing rule described in mixing_simple.

Methods

calculate(T, P, zs, ws, method)	Method to calculate heat capacity of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
test_method_validity(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax

Maximum temperature at which no method can calculate the heat capacity above.

Tmin

Minimum temperature at which no method can calculate the heat capacity under.

calculate(*T*, *P*, *zs*, *ws*, *method*)

Method to calculate heat capacity of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see *mixture_property* for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

Cplm [float] Molar heat capacity of the liquid mixture at the given conditions, [J/mol]

name = 'Liquid heat capacity'

property_max = 10000.0

Maximum valid of Heat capacity; arbitrarily set. For fluids very near the critical point, this value can be obscenely high.

property_min = 1

Allow very low heat capacities; arbitrarily set; liquid heat capacity should always be somewhat substantial.

ranked_methods = ['LALIBERTE', 'LINEAR']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)

Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'J/mol'

thermo.heat_capacity.**heat_capacity_liquid_mixture_methods = ['LALIBERTE', 'LINEAR']**

Holds all methods available for the `HeatCapacityLiquidMixture` class, for use in iterating over them.

7.15.5 Mixture Gas Heat Capacity

class thermo.heat_capacity.**HeatCapacityGasMixture**(*CASs=[]*, *HeatCapacityGases=[]*, *MWs=[]*)

Bases: `thermo.utils.mixture_property.MixtureProperty`

Class for dealing with the gas heat capacity of a mixture as a function of temperature, pressure, and composition. Consists only of mole weighted averaging.

Parameters

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

HeatCapacityGases [list[HeatCapacityGas], optional] HeatCapacityGas objects created for all species in the mixture [-]

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

Notes

To iterate over all methods, use the list stored in *heat_capacity_gas_mixture_methods*.

LINEAR: Mixing rule described in mixing_simple.

Methods

calculate(T, P, zs, ws, method)	Method to calculate heat capacity of a gas mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
test_method_validity(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the heat capacity above.

Tmin
Minimum temperature at which no method can calculate the heat capacity under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate heat capacity of a gas mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see *mixture_property* for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

Cpgm [float] Molar heat capacity of the gas mixture at the given conditions, [J/mol]

name = 'Gas heat capacity'

property_max = 10000.0
Maximum valid of Heat capacity; arbitrarily set. For fluids very near the critical point, this value can be obscenely high.

property_min = 0
Heat capacities have a minimum value of 0 at 0 K.

ranked_methods = ['LINEAR']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'J/mol'

thermo.heat_capacity.**heat_capacity_gas_mixture_methods = ['LINEAR']**

Holds all methods available for the *HeatCapacityGasMixture* class, for use in iterating over them.

7.15.6 Mixture Solid Heat Capacity

class thermo.heat_capacity.**HeatCapacitySolidMixture**(*CASs=[]*, *HeatCapacitySolids=[]*, *MWs=[]*)

Bases: *thermo.utils.mixture_property.MixtureProperty*

Class for dealing with solid heat capacity of a mixture as a function of temperature, pressure, and composition. Consists only of mole weighted averaging.

Parameters

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

HeatCapacitySolids [list[HeatCapacitySolid], optional] HeatCapacitySolid objects created for all species in the mixture [-]

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

Notes

To iterate over all methods, use the list stored in *heat_capacity_solid_mixture_methods*.

LINEAR: Mixing rule described in `mixing_simple`.

Methods

calculate(T, P, zs, ws, method)	Method to calculate heat capacity of a solid mixture at temperature T, pressure P, mole fractions zs and weight fractions ws with a given method.
test_method_validity(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax

Maximum temperature at which no method can calculate the heat capacity above.

Tmin

Minimum temperature at which no method can calculate the heat capacity under.

calculate(*T*, *P*, *zs*, *ws*, *method*)

Method to calculate heat capacity of a solid mixture at temperature T, pressure P, mole fractions zs and weight fractions ws with a given method.

This method has no exception handling; see *mixture_property* for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

Cpsm [float] Molar heat capacity of the solid mixture at the given conditions, [J/mol]

`name = 'Solid heat capacity'`

`property_max = 10000.0`
Maximum value of Heat capacity; arbitrarily set.

`property_min = 0`
Heat capacities have a minimum value of 0 at 0 K.

`ranked_methods = ['LINEAR']`

`test_method_validity`(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

`units = 'J/mol'`

`thermo.heat_capacity.heat_capacity_solid_mixture_methods = ['LINEAR']`
Holds all methods available for the `HeatCapacitySolidMixture` class, for use in iterating over them.

7.16 Interfacial/Surface Tension (thermo.interface)

This module contains implementations of `TDependentProperty` representing liquid-air surface tension. A variety of estimation and data methods are available as included in the *chemicals* library. Additionally a liquid mixture surface tension predictor objects are implemented subclassing `MixtureProperty`.

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

7.16.1 Pure Liquid Surface Tension

class thermo.interface.**SurfaceTension**(*MW=None*, *Tb=None*, *Tc=None*, *Pc=None*, *Vc=None*, *Zc=None*, *omega=None*, *StielPolar=None*, *Hvap_Tb=None*, *CASRN=''*, *Vml=None*, *Cpl=None*, *extrapolation='DIPPR106_AB'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with surface tension as a function of temperature. Consists of three coefficient-based methods and four data sources, one source of tabular information, five corresponding-states estimators, and one substance-specific method.

Parameters

Tb [float, optional] Boiling point, [K]

MW [float, optional] Molecular weight, [g/mol]

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, [-]

StielPolar [float, optional] Stiel polar factor

Hvap_Tb [float] Mass enthalpy of vaporization at the normal boiling point [kg/m^3]

CASRN [str, optional] The CAS number of the chemical

Vml [float or callable, optional] Liquid molar volume at a given temperature and pressure or callable for the same, [m^3/mol]

Cpl [float or callable, optional] Molar heat capacity of the fluid at a pressure and temperature or or callable for the same, [J/mol/K]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

chemicals.interface.REFPROP_sigma

chemicals.interface.Somayajulu

chemicals.interface.Jasper

chemicals.interface.Brock_Bird

chemicals.interface.Sastri_Rao

chemicals.interface.Pitzer

chemicals.interface.Zuo_Stenby

chemicals.interface.Miqueu

chemicals.interface.Aleem

chemicals.interface.sigma_IAPWS

Notes

To iterate over all methods, use the list stored in *surface_tension_methods*.

***IAPWS:** The IAPWS formulation for water, REFPROP_sigma

STREFPROP: The REFPROP coefficient-based method, documented in the function REFPROP_sigma for 115 fluids from [5].

SOMAYAJULU and SOMAYAJULU2: The Somayajulu coefficient-based method, documented in the function Somayajulu. Both methods have data for 64 fluids. The first data set if from [1], and the second from [2]. The later, revised coefficients should be used prefered.

JASPER: Fit with a single temperature coefficient from Jaspen (1972) as documented in the function Jasper. Data for 522 fluids is available, as shown in [4] but originally in [3].

BROCK_BIRD: CSP method documented in Brock_Bird. Most popular estimation method; from 1955.

SASTRI_RAO: CSP method documented in Sastri_Rao. Second most popular estimation method; from 1995.

PITZER: CSP method documented in Pitzer_sigma; from 1958.

ZUO_STENBY: CSP method documented in Zuo_Stenby; from 1997.

MIQUEU: CSP method documented in Miqueu.

ALEEM: CSP method documented in Aleem.

VDI_TABULAR: Tabular data in [6] along the saturation curve; interpolation is as set by the user or the default.

References

[1], [2], [3], [4], [5], [6]

Methods

calculate(T, method)	Method to calculate surface tension of a liquid at temperature T with a given method.
test_method_validity(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)

Method to calculate surface tension of a liquid at temperature T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate surface tension, [K]

method [str] Name of the method to use

Returns

sigma [float] Surface tension of the liquid at T, [N/m]

name = 'Surface tension'

property_max = 4.0
Maximum valid value of surface tension. Set to roughly twice that of cobalt at its melting point.

property_min = 0
Mimimum valid value of surface tension. This occurs at the critical point exactly.

ranked_methods = ['IAPWS', 'REFPROP', 'SOMAYAJULU2', 'SOMAYAJULU', 'VDI_PPDS', 'VDI_TABULAR', 'JASPER', 'MIQUEU', 'BROCK_BIRD', 'SASTRI_RAO', 'PITZER', 'ZUO_STENBY', 'Aleem']
Default rankings of the available methods.

test_method_validity(*T*, *method*)
Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For CSP methods, the models are considered valid from 0 K to the critical point. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'N/m'

thermo.interface.**surface_tension_methods = ['IAPWS', 'REFPROP', 'SOMAYAJULU2', 'SOMAYAJULU', 'VDI_PPDS', 'VDI_TABULAR', 'JASPER', 'MIQUEU', 'BROCK_BIRD', 'SASTRI_RAO', 'PITZER', 'ZUO_STENBY', 'Aleem']**
Holds all methods available for the `SurfaceTension` class, for use in iterating over them.

7.16.2 Mixture Liquid Heat Capacity

class thermo.interface.**SurfaceTensionMixture**(*MWs=[]*, *Tbs=[]*, *Tcs=[]*, *CASs=[]*, *SurfaceTensions=[]*, *VolumeLiquids=[]*, ***kwargs*)
Bases: `thermo.utils.mixture_property.MixtureProperty`

Class for dealing with surface tension of a mixture as a function of temperature, pressure, and composition. Consists of two mixing rules specific to surface tension, and mole weighted averaging.

Prefered method is `Winterfeld_Scriven_Davis` which requires mole fractions, pure component surface tensions, and the molar density of each pure component. `Diguilio_Teja` is of similar accuracy, but requires the surface tensions of pure components at their boiling points, as well as boiling points and critical points and mole fractions. An ideal mixing rule based on mole fractions, **LINEAR**, is also available and is still relatively accurate.

Parameters

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

Tbs [list[float], optional] Boiling points of all species in the mixture, [K]

Tcs [list[float], optional] Critical temperatures of all species in the mixture, [K]

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

SurfaceTensions [list[SurfaceTension], optional] SurfaceTension objects created for all species in the mixture [-]

VolumeLiquids [list[VolumeLiquid], optional] VolumeLiquid objects created for all species in the mixture [-]

correct_pressure_pure [bool, optional] Whether to try to use the better pressure-corrected pure component models or to use only the T-only dependent pure species models, [-]

See also:

`chemicals.interface.Winterfeld_Scriven_Davis`

`chemicals.interface.Diguilio_Teja`

Notes

To iterate over all methods, use the list stored in *surface_tension_mixture_methods*.

WINTERFELDSCRIVENDAVIS: Mixing rule described in `Winterfeld_Scriven_Davis`.

DIGUILIOTEJA: Mixing rule described in `Diguilio_Teja`.

LINEAR: Mixing rule described in `mixing_simple`.

References

[1]

Methods

calculate(T, P, zs, ws, method)	Method to calculate surface tension of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
test_method_validity(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate surface tension of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see *mixture_property* for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

sigma [float] Surface tension of the liquid at given conditions, [N/m]

name = 'Surface tension'

property_max = 4.0

Maximum valid value of surface tension. Set to roughly twice that of cobalt at its melting point.

property_min = 0

Mimimum valid value of surface tension. This occurs at the critical point exactly.

ranked_methods = ['Winterfeld, Scriven, and Davis (1978)', 'Diguilio and Teja (1988)', 'LINEAR']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)

Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'N/m'

thermo.interface.**surface_tension_mixture_methods = ['Winterfeld, Scriven, and Davis (1978)', 'Diguilio and Teja (1988)', 'LINEAR']**

Holds all methods available for the *SurfaceTensionMixture* class, for use in iterating over them.

7.17 Interaction Parameters (thermo.interaction_parameters)

This module contains a small database of interaction parameters. Only two data sets are currently included, both from ChemSep. If you would like to add parameters to this project please make a referenced compilation of values and submit them on GitHub.

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

class thermo.interaction_parameters.**InteractionParameterDB**

Basic database framework for interaction parameters.

Methods

get_ip_asymmetric_matrix(name, CASs, ip[, T])	Get a table of interaction parameters from a specified source for the specified parameters.
get_ip_automatic(CASs, ip_type, ip)	Get an interaction parameter for the first table containing the value.
get_ip_specific(name, CASs, ip)	Get an interaction parameter from a table.

continues on next page

Table 75 – continued from previous page

get_ip_symmetric_matrix(name, CASs, ip[, T])	Get a table of interaction parameters from a specified source for the specified parameters.
get_tables_with_type(ip_type)	Get a list of tables which have a type of a parameter.
has_ip_specific(name, CASs, ip)	Check if a bip exists in a table.
load_json(file, name)	Load a json file from disk containing interaction coefficients.
validate_table(name)	Basic method which checks that all CAS numbers are valid, and that all elements of the data have non-nan values.

get_ip_asymmetric_matrix(*name*, *CASs*, *ip*, *T=298.15*)

Get a table of interaction parameters from a specified source for the specified parameters.

Parameters

name [str] Name of the data table, [-]

CASs [Iterable[str]] CAS numbers; they do not need to be sorted, [-]

ip [str] Name of the parameter to retrieve, [-]

T [float, optional] Temperature of the system, [-]

Returns

values [list[list[float]]] Interaction parameters specified by *ip*, [-]

Examples

```
>>> from thermo.interaction_parameters import IPDB
>>> IPDB.get_ip_symmetric_matrix(name='ChemSep NRTL', CASs=['64-17-5', '7732-18-
↪5', '67-56-1'], ip='alphaij')
[[0.0, 0.2937, 0.3009], [0.2937, 0.0, 0.2999], [0.3009, 0.2999, 0.0]]
```

get_ip_automatic(*CASs*, *ip_type*, *ip*)

Get an interaction parameter for the first table containing the value.

Parameters

CASs [Iterable[str]] CAS numbers; they do not need to be sorted, [-]

ip_type [str] Name of the parameter type, [-]

ip [str] Name of the parameter to retrieve, [-]

Returns

value [float] Interaction parameter specified by *ip*, [-]

Examples

```
>>> from thermo.interaction_parameters import IPDB
>>> IPDB.get_ip_automatic(CASs=['7727-37-9', '74-84-0'], ip_type='PR kij', ip=
↪'kij')
0.0533
```

get_ip_specific(*name*, *CASs*, *ip*)

Get an interaction parameter from a table. If the specified parameter is missing, the default *missing* value as defined in the data file is returned instead.

Parameters

name [str] Name of the data table, [-]

CASs [Iterable[str]] CAS numbers; they do not need to be sorted, [-]

ip [str] Name of the parameter to retrieve, [-]

Returns

value [float] Interaction parameter specified by *ip*, [-]

Examples

Check if nitrogen-ethane as a PR BIP:

```
>>> from thermo.interaction_parameters import IPDB
>>> IPDB.get_ip_specific('ChemSep PR', ['7727-37-9', '74-84-0'], 'kij')
0.0533
```

get_ip_symmetric_matrix(*name*, *CASs*, *ip*, *T=298.15*)

Get a table of interaction parameters from a specified source for the specified parameters. This method assumes symmetric parameters for speed.

Parameters

name [str] Name of the data table, [-]

CASs [Iterable[str]] CAS numbers; they do not need to be sorted, [-]

ip [str] Name of the parameter to retrieve, [-]

T [float, optional] Temperature of the system, [-]

Returns

values [list[list[float]]] Interaction parameters specified by *ip*, [-]

Examples

```
>>> from thermo.interaction_parameters import IPDB
>>> IPDB.get_ip_symmetric_matrix(name='ChemSep PR', CASs=['7727-37-9', '74-84-0
↪', '74-98-6'], ip='kij')
[[0.0, 0.0533, 0.0878], [0.0533, 0.0, 0.0011], [0.0878, 0.0011, 0.0]]
```

get_tables_with_type(*ip_type*)

Get a list of tables which have a type of a parameter.

Parameters

ip_type [str] Name of the parameter type, [-]

Returns

table_names [list[str]] Interaction parameter tables including *ip*, [-]

Examples

```
>>> from thermo.interaction_parameters import IPDB
>>> IPDB.get_tables_with_type('PR kij')
['ChemSep PR']
```

has_ip_specific(*name*, *CASs*, *ip*)

Check if a bip exists in a table.

Parameters

name [str] Name of the data table, [-]

CASs [Iterable[str]] CAS numbers; they do not need to be sorted, [-]

ip [str] Name of the parameter to retrieve, [-]

Returns

present [bool] Whether or not the data is included in the table, [-]

Examples

Check if nitrogen-ethane as a PR BIP:

```
>>> from thermo.interaction_parameters import IPDB
>>> IPDB.has_ip_specific('ChemSep PR', ['7727-37-9', '74-84-0'], 'kij')
True
```

load_json(*file*, *name*)

Load a json file from disk containing interaction coefficients.

The format for the file is as follows:

A *data* key containing a dictionary with a key:

- ***CAS1 CAS2*** [str] The CAS numbers of both components, sorted from small to high as integers; they should have the '-' symbols still in them and have a single space between them; if these are ternary or higher parameters, follow the same format for the other CAS numbers, [-]
- **values** [dict[str][various]] All of the values listed in the metadata element *necessary keys*; they are None if missing.

A *metadata* key containing:

- ***symmetric*** [bool] Whether or not the interaction coefficients are missing.
- ***source*** [str] Where the data came from.
- ***components*** [int] The number of components each interaction parameter is for; 2 for binary, 3 for ternary, etc.
- ***necessary keys*** [list[str]] Which elements are required in the data.

- ***P dependent*** [bool] Whether or not the interaction parameters are pressure dependent.
- ***missing*** [dict[str][float]] Values which are missing are returned with these values
- ***type*** [One of 'PR kij', 'SRK kij', etc; used to group data but not] tied into anything else.
- ***T dependent*** [bool] Whether or not the data is T-dependent.

Parameters

file [str] Path to json file on disk which contains interaction coefficients, [-]

name [str] Name that the data read should be referred to by, [-]

`validate_table`(*name*)

Basic method which checks that all CAS numbers are valid, and that all elements of the data have non-nan values. Raises an exception if any of the data is missing or is a nan value.

`thermo.interaction_parameters.IPDB = <thermo.interaction_parameters.InteractionParameterDB object>`

Basic database framework for interaction parameters.

Exmple database with NRTL and PR values from ChemSep. This is lazy-loaded, access it as *thermo.interaction_parameters.IPDB*.

`class thermo.interaction_parameters.ScalarParameterDB`

Basic database framework for scalar parameters of various thermodynamic models. The following keys are used:

Peng-Robinson

`Twu Volume-translated Peng-Robinson`: *TwuPRL*, *TwuPRM*, *TwuPRN*, *TwuPRc*

`Volume-translated Peng-Robinson`: *PRc*

`Peng-Robinson-Stryjek-Vera`: *PRSVkappa1*

`Peng-Robinson-Stryjek-Vera 2`: *PRSV2kappa1*, *PRSV2kappa2*, *PRSV2kappa3*

SRK

`Twu Volume-translated Peng-Robinson`: *TwuSRKL*, *TwuSRKM*, *TwuSRKN*, *TwuSRKc*

`Volume-translated Peng-Robinson`: *SRKc*

`Refinery Soave-Redlich-Kwong`: *APISRKS1*, *APISRKS2*

`MSRK`: *MSRKM*, *MSRKN*, *MSRKc*

`Predictive Soave-Redlich-Kwong`: *MCSRKC1*, *MCSRKC2*, *MCSRKC3*

Excess Gibbs Energy Models

`Regular Solution`: *RegularSolutionV*, *RegularSolutionSP*

Methods

get_parameter_automatic(CAS, parameter)	Get an interaction parameter for the first table containing the value.
get_parameter_specific(name, CAS, parameter)	Get a parameter from a table.
get_parameter_vector(name, CASs, parameter)	Get a list of parameters from a specified source for the specified parameter.
get_tables_with_type(parameter)	Get a list of tables which have a parameter.
has_parameter_specific(name, CAS, parameter)	Check if a parameter exists in a table.

load_json	

SPDB

Example scalar parameters for models. This is lazy-loaded, access it as *thermo.interaction_parameters.SPDB*.

7.18 Legal and Economic Chemical Data (thermo.law)

thermo.law.**economic_status**(*CASRN*, *method=None*, *get_methods=False*)

Look up the economic status of a chemical.

This API is considered experimental, and is expected to be removed in a future release in favor of a more complete object-oriented interface.

```
>>> economic_status(CASRN='98-00-0')
["US public: {'Manufactured': 0.0, 'Imported': 10272.711, 'Exported': 184.127}",
↪'10,000 - 100,000 tonnes per annum', 'OECD HPV Chemicals']
```

```
>>> economic_status(CASRN='13775-50-3')  # SODIUM SESQUISULPHATE
[]
>>> economic_status(CASRN='98-00-0', method='OECD high production volume chemicals')
'OECD HPV Chemicals'
>>> economic_status(CASRN='98-01-1', method='European Chemicals Agency Total␣
↪Tonnage Bands')
['10,000 - 100,000 tonnes per annum']
```

thermo.law.**legal_status**(*CASRN*, *method=None*, *get_methods=False*, *CASi=None*)

Looks up the legal status of a chemical 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 [string] CASRN [-]

Returns

status [str or dict] Legal status information [-]

methods [list, only returned if get_methods == True] List of methods which can be used to obtain legal status with the given inputs

Other Parameters

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

get_methods [bool, optional] If True, function will determine which methods can be used to obtain the legal status for the desired chemical, and will return methods instead of the status

CASi [int, optional] CASRN as an integer, used internally [-]

Notes

Supported methods are:

- **DSL**: Canada Domestic Substance List, [1]. As extracted on Feb 11, 2015 from a html list. This list is updated continuously, so this version will always be somewhat old. Strictly speaking, there are multiple lists but they are all bundled together here. A chemical may be 'Listed', or be on the 'Non-Domestic Substances List (NDSL)', or be on the list of substances with 'Significant New Activity (SNAc)', or be on the DSL but with a 'Ministerial Condition pertaining to this substance', or have been removed from the DSL, or have had a Ministerial prohibition for the substance.
- **TSCA**: USA EPA Toxic Substances Control Act Chemical Inventory, [2]. This list is as extracted on 2016-01. It is believed this list is updated on a periodic basis (> 6 month). A chemical may simply be 'Listed', or may have certain flags attached to it. All these flags are described in the dict TSCA_flags.
- **EINECS**: European INventory of Existing Commercial chemical Substances, [3]. As extracted from a spreadsheet dynamically generated at [1]. This list was obtained March 2015; a more recent revision already exists.
- **NLP**: No Longer Polymers, a list of chemicals with special regulatory exemptions in EINECS. Also described at [3].
- **SPIN**: Substances Prepared in Nordic Countries. Also a boolean data type. Retrieved 2015-03 from [4].

Other methods which could be added are:

- Australia: AICS Australian Inventory of Chemical Substances
- China: Inventory of Existing Chemical Substances Produced or Imported in China (IECSC)
- Europe: REACH List of Registered Substances
- India: List of Hazardous Chemicals
- Japan: ENCS: Inventory of existing and new chemical substances
- Korea: Existing Chemicals Inventory (KECI)
- Mexico: INSQ National Inventory of Chemical Substances in Mexico
- New Zealand: Inventory of Chemicals (NZIoC)
- Philippines: PICCS Philippines Inventory of Chemicals and Chemical Substances

References

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

Examples

```
>>> legal_status('64-17-5')
{'DSL': 'LISTED', 'TSCA': 'LISTED', 'EINECS': 'LISTED', 'NLP': 'UNLISTED', 'SPIN':
↪'LISTED'}
```

thermo.law.**load_economic_data**()

thermo.law.**load_law_data**()

7.19 NRTL Gibbs Excess Model (thermo.nrtl)

This module contains a class *NRTL* for performing activity coefficient calculations with the NRTL model. An older, functional calculation for activity coefficients only is also present, *NRTL_gammas*.

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

7.19.1 NRTL Class

class thermo.nrtl.**NRTL**(*T*, *xs*, *tau_coeffs=None*, *alpha_coeffs=None*, *ABEFGHCD=None*, *tau_as=None*, *tau_bs=None*, *tau_es=None*, *tau_fs=None*, *tau_gs=None*, *tau_hs=None*, *alpha_cs=None*, *alpha_ds=None*)

Bases: *thermo.activity.GibbsExcess*

Class for representing an a liquid with excess gibbs energy represented by the NRTL equation. This model is capable of representing VL and LL behavior. [1] and [2] are good references on this model.

$$g^E = RT\sum_i x_i \frac{\sum_j \tau_{ji} G_{ji} x_j}{\sum_j G_{ji} x_j}$$

$$G_{ij} = \exp(-\alpha_{ij}\tau_{ij})$$

$$\alpha_{ij} = c_{ij} + d_{ij}T$$

$$\tau_{ij} = A_{ij} + \frac{B_{ij}}{T} + E_{ij}\ln T + F_{ij}T + \frac{G_{ij}}{T^2} + H_{ij}T^2$$

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

tau_coeffs [list[list[list[float]]], optional] NRTL parameters, indexed by [i][j] and then each value is a 6 element list with parameters [a, b, e, f, g, h]; either (*tau_coeffs* and *alpha_coeffs*) or *ABEFGHCD* are required, [-]

alpha_coeffs [list[list[float]], optional] NRTL alpha parameters, []

ABEFGHCD [tuple[list[list[float]], 8], optional] Contains the following. One of (*tau_coeffs* and *alpha_coeffs*) or *ABEFGHCD* or some of the *tau* or *alpha* parameters are required, [-]

tau_as [list[list[float]], optional] a parameters used in calculating `NRTL.taus`, [-]

tau_bs [list[list[float]], optional] b parameters used in calculating `NRTL.taus`, [K]

tau_es [list[list[float]], optional] e parameters used in calculating `NRTL.taus`, [-]

tau_fs [list[list[float]], optional] f paraemeters used in calculating `NRTL.taus`, [1/K]

tau_gs [list[list[float]], optional] e parameters used in calculating `NRTL.taus`, [K^2]

tau_hs [list[list[float]], optional] f parameters used in calculating `NRTL.taus`, [1/K^2]

alpha_cs [list[list[float]], optional] c parameters used in calculating `NRTL.alphas`, [-]

alpha_ds [list[list[float]], optional] d paraemeters used in calculating `NRTL.alphas`, [1/K]

Notes

In addition to the methods presented here, the methods of its base class `thermo.activity.GibbsExcess` are available as well.

References

[1], [2]

Examples

The DDBST has published numerous problems showing this model a simple binary system, Example P05.01b in [2], shows how to use parameters from the DDBST which are in units of calorie and need the gas constant as a multiplier:

```
>>> from scipy.constants import calorie, R
>>> N = 2
>>> T = 70.0 + 273.15
>>> xs = [0.252, 0.748]
>>> tausA = tausE = tausF = tausG = tausH = alphaD = [[0.0]*N for i in range(N)]
>>> tausB = [[0, -121.2691/R*calorie], [1337.8574/R*calorie, 0]]
>>> alphaC =  [[0, 0.2974],[.2974, 0]]
>>> ABEFGHCD = (tausA, tausB, tausE, tausF, tausG, tausH, alphaC, alphaD)
>>> GE = NRTL(T=T, xs=xs, ABEFGHCD=ABEFGHCD)
>>> GE.gammas()
[1.93605165145, 1.15366304520]
>>> GE
NRTL(T=343.15, xs=[0.252, 0.748], tau_bs=[[0, -61.0249799309399], [673.
↪2359767282798, 0]], alpha_cs=[[0, 0.2974], [0.2974, 0]])
>>> GE.GE(), GE.dGE_dT(), GE.d2GE_dT2()
(780.053057219, 0.5743500022, -0.003584843605528)
```

(continues on next page)

(continued from previous page)

```
>>> GE.HE(), GE.SE(), GE.dHE_dT(), GE.dSE_dT()
(582.964853938, -0.57435000227, 1.230139083237, 0.0035848436055)
```

The solution given by the DDBST has the same values [1.936, 1.154], and can be found here: http://chemthermo.ddbst.com/Problems_Solutions/Mathcad_Files/P05.01b%20VLE%20Behavior%20of%20Ethanol%20-%20Water%20Using%20NRTL.xps

Attributes

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Methods

GE()	Calculate and return the excess Gibbs energy of a liquid phase represented by the NRTL model.
Gs()	Calculates and return the *G* terms in the NRTL model for a specified temperature.
alphas()	Calculates and return the *alpha* terms in the NRTL model for a specified temperature.
d2GE_dT2()	Calculate and return the second tempreature derivative of excess Gibbs energy of a liquid phase represented by the NRTL model.
d2GE_dTdxs()	Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy of a liquid represented by the NRTL model.
d2GE_dxixjs()	Calculate and return the second mole fraction derivatives of excess Gibbs energy of a liquid represented by the NRTL model.
d2Gs_dT2()	Calculates and return the second temperature derivative of *G* terms in the NRTL model for a specified temperature.
d2taus_dT2()	Calculate and return the second temperature derivative of the *tau* terms for the NRTL model for a specified temperature.
d3Gs_dT3()	Calculates and return the third temperature derivative of *G* terms in the NRTL model for a specified temperature.
d3taus_dT3()	Calculate and return the third temperature derivative of the *tau* terms for the NRTL model for a specified temperature.
dGE_dT()	Calculate and return the first tempreature derivative of excess Gibbs energy of a liquid phase represented by the NRTL model.
dGE_dxs()	Calculate and return the mole fraction derivatives of excess Gibbs energy of a liquid represented by the NRTL model.
dGs_dT()	Calculates and return the first temperature derivative of *G* terms in the NRTL model for a specified temperature.

continues on next page

Table 77 – continued from previous page

dtaus_dT()	Calculate and return the temperature derivative of the *tau* terms for the NRTL model for a specified temperature.
taus()	Calculate and return the *tau* terms for the NRTL model for a specified temperature.
to_T_xs(T, xs)	Method to construct a new NRTL instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

to_T_xs(*T*, *xs*)

Method to construct a new NRTL instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions of each component, [-]

Returns

obj [NRTL] New NRTL object at the specified conditions [-]

Notes

If the new temperature is the same temperature as the existing temperature, if the *tau*, *Gs*, or *alphas* terms or their derivatives have been calculated, they will be set to the new object as well.

taus()

Calculate and return the *tau* terms for the NRTL model for a specified temperature.

$$\tau_{ij} = A_{ij} + \frac{B_{ij}}{T} + E_{ij}\ln T + F_{ij}T + \frac{G_{ij}}{T^2} + H_{ij}T^2$$

Returns

taus [list[list[float]]] tau terms, asymmetric matrix [-]

Notes

These *tau ij* values (and the coefficients) are NOT symmetric.

dtaus_dT()

Calculate and return the temperature derivative of the *tau* terms for the NRTL model for a specified temperature.

$$\frac{\partial \tau_{ij}}{\partial T}_{P,x_i} = -\frac{B_{ij}}{T^2} + \frac{E_{ij}}{T} + F_{ij} - \frac{2G_{ij}}{T^3} + 2H_{ij}T$$

Returns

dtaus_dT [list[list[float]]] First temperature derivative of tau terms, asymmetric matrix [1/K]

d2taus_dT2()

Calculate and return the second temperature derivative of the *tau* terms for the NRTL model for a specified temperature.

$$\frac{\partial^2 \tau_{ij}}{\partial T^2}_{P,x_i} = \frac{2B_{ij}}{T^3} - \frac{E_{ij}}{T^2} + \frac{6G_{ij}}{T^4} + 2H_{ij}$$

Returns

d2taus_dT2 [list[list[float]]] Second temperature derivative of tau terms, asymmetric matrix [1/K^2]

`d3taus_dT3()`

Calculate and return the third temperature derivative of the *tau* terms for the NRTL model for a specified temperature.

$$\frac{\partial^3 \tau_{ij}}{\partial T^3}_{P,x_i} = -\frac{6B_{ij}}{T^4} + \frac{2E_{ij}}{T^3} - \frac{24G_{ij}}{T^5}$$

Returns

d3taus_dT3 [list[list[float]]] Third temperature derivative of tau terms, asymmetric matrix [1/K^3]

`alphas()`

Calculates and return the *alpha* terms in the NRTL model for a specified temperature.

$$\alpha_{ij} = c_{ij} + d_{ij}T$$

Returns

alphas [list[list[float]]] alpha terms, possibly asymmetric matrix [-]

Notes

alpha values (and therefore *cij* and *dij* are normally symmetrical; but this is not strictly required.

Some sources suggest the c term should be fit to a given system; but the *d* term should be fit for an entire chemical family to avoid overfitting.

Recommended values for *cij* according to one source are:

0.30 Nonpolar substances with nonpolar substances; low deviation from ideality. 0.20 Hydrocarbons that are saturated interacting with polar liquids that do not associate, or systems that for multiple liquid phases which are immiscible 0.47 Strongly self associative systems, interacting with non-polar substances

alpha_coeffs should be a list[list[cij, dij]] so a 3d array

`Gs()`

Calculates and return the *G* terms in the NRTL model for a specified temperature.

$$G_{ij} = \exp(-\alpha_{ij}\tau_{ij})$$

Returns

Gs [list[list[float]]] G terms, asymmetric matrix [-]

`dGs_dT()`

Calculates and return the first temperature derivative of *G* terms in the NRTL model for a specified temperature.

$$\frac{\partial G_{ij}}{\partial T} = \left(-\alpha(T)\frac{d}{dT}\tau(T) - \tau(T)\frac{d}{dT}\alpha(T)\right)e^{-\alpha(T)\tau(T)}$$

Returns

dGs_dT [list[list[float]]] Temperature derivative of G terms, asymmetric matrix [1/K]

Notes

Derived with SymPy:

```
>>> from sympy import *
>>> T = symbols('T')
>>> alpha, tau = symbols('alpha, tau', cls=Function)
>>> diff(exp(-alpha(T)*tau(T)), T)
```

d2Gs_dT2()

Calculates and return the second temperature derivative of G terms in the NRTL model for a specified temperature.

$$\frac{\partial^2 G_{ij}}{\partial T^2} = \left(\left(\alpha(T)\frac{d}{dT}\tau(T) + \tau(T)\frac{d}{dT}\alpha(T)\right)^2 - \alpha(T)\frac{d^2}{dT^2}\tau(T) - 2\frac{d}{dT}\alpha(T)\frac{d}{dT}\tau(T)\right)e^{-\alpha(T)\tau(T)}$$

Returns

d2Gs_dT2 [list[list[float]]] Second temperature derivative of G terms, asymmetric matrix [1/K^2]

Notes

Derived with SymPy:

```
>>> from sympy import *
>>> T = symbols('T')
>>> alpha, tau = symbols('alpha, tau', cls=Function)
>>> diff(exp(-alpha(T)*tau(T)), T, 2)
```

d3Gs_dT3()

Calculates and return the third temperature derivative of G terms in the NRTL model for a specified temperature.

$$\frac{\partial^3 G_{ij}}{\partial T^3} = \left(\alpha(T)\frac{d}{dT}\tau(T) + \tau(T)\frac{d}{dT}\alpha(T)\right)^3 + \left(3\alpha(T)\frac{d}{dT}\tau(T) + 3\tau(T)\frac{d}{dT}\alpha(T)\right)\left(\alpha(T)\frac{d^2}{dT^2}\tau(T) + 2\frac{d}{dT}\alpha(T)\frac{d}{dT}\tau(\right.$$

Returns

d3Gs_dT3 [list[list[float]]] Third temperature derivative of G terms, asymmetric matrix [1/K^3]

Notes

Derived with SymPy:

```
>>> from sympy import *
>>> T = symbols('T')
>>> alpha, tau = symbols('alpha, tau', cls=Function)
>>> diff(exp(-alpha(T)*tau(T)), T, 3)
```

GE()

Calculate and return the excess Gibbs energy of a liquid phase represented by the NRTL model.

$$g^E = RT\sum_i x_i \frac{\sum_j \tau_{ji} G_{ji} x_j}{\sum_j G_{ji} x_j}$$

Returns

GE [float] Excess Gibbs energy, [J/mol]

`dGE_dT()`

Calculate and return the first tempreature derivative of excess Gibbs energy of a liquid phase represented by the NRTL model.

Returns

dGE_dT [float] First temperature derivative of excess Gibbs energy, [J/(mol*K)]

`d2GE_dT2()`

Calculate and return the second tempreature derivative of excess Gibbs energy of a liquid phase represented by the NRTL model.

Returns

d2GE_dT2 [float] Second temperature derivative of excess Gibbs energy, [J/(mol*K^2)]

`dGE_dxs()`

Calculate and return the mole fraction derivatives of excess Gibbs energy of a liquid represented by the NRTL model.

$$\frac{\partial g^E}{\partial x_i}$$

Returns

dGE_dxs [list[float]] Mole fraction derivatives of excess Gibbs energy, [J/mol]

`d2GE_dxixjs()`

Calculate and return the second mole fraction derivatives of excess Gibbs energy of a liquid represented by the NRTL model.

$$\frac{\partial^2 g^E}{\partial x_i \partial x_j} = RT\Bigg[+\frac{G_{ij}\tau_{ij}}{\sum_m x_m G_{mj}} + \frac{G_{ji}\tau_{jiij}}{\sum_m x_m G_{mi}} - \frac{(\sum_m x_m G_{mj}\tau_{mj})G_{ij}}{(\sum_m x_m G_{mj})^2} - \frac{(\sum_m x_m G_{mi}\tau_{mi})G_{ji}}{(\sum_m x_m G_{mi})^2}\sum_k \Bigg(\frac{2x_k(\sum_m x_m \tau_m}{(\sum_m x_m}$$

Returns

d2GE_dxixjs [list[list[float]]] Second mole fraction derivatives of excess Gibbs energy, [J/mol]

`d2GE_dTdxs()`

Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy of a liquid represented by the NRTL model.

$$\frac{\partial^2 g^E}{\partial x_i \partial T} = R\Bigg[-T\Bigg(\sum_j\Bigg(-\frac{x_j(G_{ij}\frac{\partial \tau_{ij}}{\partial T} + \tau_{ij}\frac{\partial G_{ij}}{\partial T})}{\sum_k x_k G_{kj}} + \frac{x_j G_{ij}\tau_{ij}(\sum_k x_k \frac{\partial G_{kj}}{\partial T})}{(\sum_k x_k G_{kj})^2} + \frac{x_j \frac{\partial G_{ij}}{\partial T}(\sum_k x_k G_{kj}\tau_{kj})}{(\sum_k x_k G_{kj})^2} + \frac{x_j G_{ij}(\sum_k x}{(}$$

Returns

d2GE_dTdxs [list[float]] Temperature derivative of mole fraction derivatives of excess Gibbs energy, [J/(mol*K)]

7.19.2 NRTL Functional Calculations

thermo.nrtl.**NRTL_gammas**(*xs*, *taus*, *alphas*)

Calculates the activity coefficients of each species in a mixture using the Non-Random Two-Liquid (NRTL) method, given their mole fractions, dimensionless interaction parameters, and nonrandomness constants. Those are normally correlated with temperature in some form, and need to be calculated separately.

$$\ln(\gamma_i) = \frac{\sum_{j=1}^{n} x_j\tau_{ji}G_{ji}}{\sum_{k=1}^{n} x_k G_{ki}} + \sum_{j=1}^{n}\frac{x_jG_{ij}}{\sum_{k=1}^{n} x_k G_{kj}}\left(\tau_{ij} - \frac{\sum_{m=1}^{n} x_m\tau_{mj}G_{mj}}{\sum_{k=1}^{n} x_k G_{kj}}\right)$$

$$G_{ij} = \exp\left(-\alpha_{ij}\tau_{ij}\right)$$

Parameters

xs [list[float]] Liquid mole fractions of each species, [-]

taus [list[list[float]]] Dimensionless interaction parameters of each compound with each other, [-]

alphas [list[list[float]]] Nonrandomness constants of each compound interacting with each other, [-]

Returns

gammas [list[float]] Activity coefficient for each species in the liquid mixture, [-]

Notes

This model needs N^2 parameters.

One common temperature dependence of the nonrandomness constants is:

$$\alpha_{ij} = c_{ij} + d_{ij}T$$

Most correlations for the interaction parameters include some of the terms shown in the following form:

$$\tau_{ij} = A_{ij} + \frac{B_{ij}}{T} + \frac{C_{ij}}{T^2} + D_{ij}\ln\left(T\right) + E_{ij}T^{F_{ij}}$$

The original form of this model used the temperature dependence of taus in the form (values can be found in the literature, often with units of calories/mol):

$$\tau_{ij} = \frac{b_{ij}}{RT}$$

For this model to produce ideal acitivty coefficients (gammas = 1), all interaction parameters should be 0; the value of alpha does not impact the calculation when that is the case.

References

[1], [2]

Examples

Ethanol-water example, at 343.15 K and 1 MPa:

```
>>> NRTL_gammas(xs=[0.252, 0.748], taus=[[0, -0.178], [1.963, 0]],
... alphas=[[0, 0.2974],[.2974, 0]])
[1.9363183763514304, 1.1537609663170014]
```

7.19.3 NRTL Regression Calculations

thermo.nrtl.**NRTL_gammas_binaries**(*xs*, *tau12*, *tau21*, *alpha12*, *alpha21*, *gammas=None*)

Calculates activity coefficients at fixed *tau* and *alpha* values for a binary system at a series of mole fractions. This is used for regression of *tau* and *alpha* parameters. This function is highly optimized, and operates on multiple points at a time.

$$\ln \gamma_1 = x_2^2 \left[\tau_{21}\left(\frac{G_{21}}{x_1 + x_2 G_{21}}\right)^2 + \frac{\tau_{12} G_{12}}{(x_2 + x_1 G_{12})^2}\right]$$

$$\ln \gamma_2 = x_1^2 \left[\tau_{12}\left(\frac{G_{12}}{x_2 + x_1 G_{12}}\right)^2 + \frac{\tau_{21} G_{21}}{(x_1 + x_2 G_{21})^2}\right]$$

$$G_{ij} = \exp(-\alpha_{ij}\tau_{ij})$$

Parameters

xs [list[float]] Liquid mole fractions of each species in the format x0_0, x1_0, (component 1 point1, component 2 point 1), x0_1, x1_1, (component 1 point2, component 2 point 2), … [-]

tau12 [float] *tau* parameter for 12, [-]

tau21 [float] *tau* parameter for 21, [-]

alpha12 [float] *alpha* parameter for 12, [-]

alpha21 [float] *alpha* parameter for 21, [-]

gammas [list[float], optional] Array to store the activity coefficient for each species in the liquid mixture, indexed the same as *xs*; can be omitted or provided for slightly better performance [-]

Returns

gammas [list[float]] Activity coefficient for each species in the liquid mixture, indexed the same as *xs*, [-]

Examples

```
>>> NRTL_gammas_binaries([.1, .9, 0.3, 0.7, .85, .15], 0.1759, 0.7991, .2, .3)
[2.121421, 1.011342, 1.52177, 1.09773, 1.016062, 1.841391]
```

7.20 Legacy Mixtures (thermo.mixture)

class thermo.mixture.**Mixture**(*IDs=None*, *zs=None*, *ws=None*, *Vfls=None*, *Vfgs=None*, *T=None*, *P=None*, *VF=None*, *H=None*, *Hm=None*, *S=None*, *Sm=None*, *pkg=None*, *Vf_TP=(None, None)*)

Bases: `object`

Creates a Mixture object which contains basic information such as molecular weight and the structure of the species, as well as thermodynamic and transport properties as a function of two of the variables temperature, pressure, vapor fraction, enthalpy, or entropy.

The components of the mixture must be specified by specifying the names of the chemicals; the composition can be specified by providing any one of the following parameters:

- Mass fractions *ws*
- Mole fractions *zs*
- Liquid volume fractions (based on pure component densities) *Vfls*
- Gas volume fractions (based on pure component densities) *Vfgs*

If volume fractions are provided, by default the pure component volumes are calculated at the specified *T* and *P*. To use another reference temperature and pressure specify it as a tuple for the argument *Vf_TP*.

If no thermodynamic conditions are specified, or if only one of T and P are specifed without another thermodynamic variable as well, the T and P 298.15 K and/or 101325 Pa will be set instead of the missing variables.

Parameters

IDs [list, optional] List of chemical identifiers - names, CAS numbers, SMILES or InChi strings can all be recognized and may be mixed [-]

zs [list or dict, optional] Mole fractions of all components in the mixture [-]

ws [list or dict, optional] Mass fractions of all components in the mixture [-]

Vfls [list or dict, optional] Volume fractions of all components as a hypothetical liquid phase based on pure component densities [-]

Vfgs [list, or dict optional] Volume fractions of all components as a hypothetical gas phase based on pure component densities [-]

T [float, optional] Temperature of the mixture (default 298.15 K), [K]

P [float, optional] Pressure of the mixture (default 101325 Pa) [Pa]

VF [float, optional] Vapor fraction (mole basis) of the mixture, [-]

Hm [float, optional] Molar enthalpy of the mixture, [J/mol]

H [float, optional] Mass enthalpy of the mixture, [J/kg]

Sm [float, optional] Molar entropy of the mixture, [J/mol/K]

S [float, optional] Mass entropy of the mixture, [J/kg/K]

pkg [object] The thermodynamic property package to use for flash calculations; one of the caloric packages in *thermo.property_package*; defaults to the ideal model [-]

Vf_TP [tuple(2, float), optional] The (T, P) at which the volume fractions are specified to be at, [K] and [Pa]

Notes

> **Warning:** The Mixture class is not designed for high-performance or the ability to use different thermodynamic models. It is especially limited in its multiphase support and the ability to solve with specifications other than temperature and pressure. It is impossible to change constant properties such as a compound's critical temperature in this interface.
>
> It is recommended to switch over to the *thermo.flash* interface which solves those problems and is better positioned to grow. That interface also requires users to be responsible for their chemical constants and pure component correlations; while default values can easily be loaded for most compounds, the user is ultimately responsible for them.

Examples

Creating Mixture objects:

```
>>> Mixture(['water', 'ethanol'], Vfls=[.6, .4], T=300, P=1E5)
<Mixture, components=['water', 'ethanol'], mole fractions=[0.8299, 0.1701], T=300.
↪00 K, P=100000 Pa>
```

For mixtures with large numbers of components, it may be confusing to enter the composition separate from the names of the chemicals. For that case, the syntax using dictionaries as follows is supported with any composition specification:

```
>>> comp = OrderedDict([('methane', 0.96522),
...                     ('nitrogen', 0.00259),
...                     ('carbon dioxide', 0.00596),
...                     ('ethane', 0.01819),
...                     ('propane', 0.0046),
...                     ('isobutane', 0.00098),
...                     ('butane', 0.00101),
...                     ('2-methylbutane', 0.00047),
...                     ('pentane', 0.00032),
...                     ('hexane', 0.00066)])
>>> m = Mixture(zs=comp)
```

Attributes

MW [float] Mole-weighted average molecular weight all chemicals in the mixture, [g/mol]

IDs [list of str] Names of all the species in the mixture as given in the input, [-]

names [list of str] Names of all the species in the mixture, [-]

CASs [list of str] CAS numbers of all species in the mixture, [-]

MWs [list of float] Molecular weights of all chemicals in the mixture, [g/mol]

Tms [list of float] Melting temperatures of all chemicals in the mixture, [K]

Tbs [list of float] Boiling temperatures of all chemicals in the mixture, [K]

Tcs [list of float] Critical temperatures of all chemicals in the mixture, [K]

Pcs [list of float] Critical pressures of all chemicals in the mixture, [Pa]

Vcs [list of float] Critical volumes of all chemicals in the mixture, [m^3/mol]

Zcs [list of float] Critical compressibilities of all chemicals in the mixture, [-]

rhocs [list of float] Critical densities of all chemicals in the mixture, [kg/m^3]

rhocms [list of float] Critical molar densities of all chemicals in the mixture, [mol/m^3]

omegas [list of float] Acentric factors of all chemicals in the mixture, [-]

StielPolars [list of float] Stiel Polar factors of all chemicals in the mixture, see `chemicals.acentric.Stiel_polar_factor` for the definition, [-]

Tts [list of float] Triple temperatures of all chemicals in the mixture, [K]

Pts [list of float] Triple pressures of all chemicals in the mixture, [Pa]

Hfuss [list of float] Enthalpy of fusions of all chemicals in the mixture, [J/kg]

Hfusms [list of float] Molar enthalpy of fusions of all chemicals in the mixture, [J/mol]

Hsubs [list of float] Enthalpy of sublimations of all chemicals in the mixture, [J/kg]

Hsubms [list of float] Molar enthalpy of sublimations of all chemicals in the mixture, [J/mol]

Hfms [list of float] Molar enthalpy of formations of all chemicals in the mixture, [J/mol]

Hfs [list of float] Enthalpy of formations of all chemicals in the mixture, [J/kg]

Gfms [list of float] Molar Gibbs free energies of formation of all chemicals in the mixture, [J/mol]

Gfs [list of float] Gibbs free energies of formation of all chemicals in the mixture, [J/kg]

Sfms [list of float] Molar entropy of formation of all chemicals in the mixture, [J/mol/K]

Sfs [list of float] Entropy of formation of all chemicals in the mixture, [J/kg/K]

S0ms [list of float] Standard absolute entropies of all chemicals in the mixture, [J/mol/K]

S0s [list of float] Standard absolute entropies of all chemicals in the mixture, [J/kg/K]

Hcms [list of float] Molar higher heats of combustions of all chemicals in the mixture, [J/mol]

Hcs [list of float] Higher heats of combustions of all chemicals in the mixture, [J/kg]

Hcms_lower [list of float] Molar lower heats of combustions of all chemicals in the mixture, [J/mol]

Hcs_lower [list of float] Higher lower of combustions of all chemicals in the mixture, [J/kg]

Tflashs [list of float] Flash points of all chemicals in the mixture, [K]

Tautoignitions [list of float] Autoignition points of all chemicals in the mixture, [K]

LFLs [list of float] Lower flammability limits of the gases in an atmosphere at STP, mole fractions, [-]

UFLs [list of float] Upper flammability limit of the gases in an atmosphere at STP, mole fractions, [-]

TWAs [list of list of tuple(quantity, unit)] Time-Weighted Average limits on worker exposure to dangerous chemicals.

STELs [list of tuple(quantity, unit)] Short-term Exposure limits on worker exposure to dangerous chemicals.

Ceilings [list of tuple(quantity, unit)] Ceiling limits on worker exposure to dangerous chemicals.

Skins [list of bool] Whether or not each of the chemicals can be absorbed through the skin.

Carcinogens [list of str or dict] Carcinogen status information for each chemical in the mixture.

Chemicals [list of Chemical instances] Chemical instances used in calculating mixture properties, [-]

dipoles [list of float] Dipole moments of all chemicals in the mixture in debye, [3.33564095198e-30 ampere*second^2]

Stockmayers [list of float] Lennard-Jones depth of potential-energy minimum over k for all chemicals in the mixture, [K]

molecular_diameters [list of float] Lennard-Jones molecular diameters of all chemicals in the mixture, [angstrom]

GWPs [list of float] Global warming potentials (default 100-year outlook) (impact/mass chemical)/(impact/mass CO2) of all chemicals in the mixture, [-]

ODPs [list of float] Ozone Depletion potentials (impact/mass chemical)/(impact/mass CFC-11), of all chemicals in the mixture, [-]

logPs [list of float] Octanol-water partition coefficients of all chemicals in the mixture, [-]

Psat_298s [list of float] Vapor pressure of the chemicals in the mixture at 298.15 K, [Pa]

phase_STPs [list of str] Phase of the chemicals in the mixture at 298.15 K and 101325 Pa; one of 's', 'l', 'g', or 'l/g'.

Vml_Tbs [list of float] Molar volumes of the chemicals in the mixture as liquids at their normal boiling points, [m^3/mol]

Vml_Tms [list of float] Molar volumes of the chemicals in the mixture as liquids at their melting points, [m^3/mol]

Vml_STPs [list of float] Molar volume of the chemicals in the mixture as liquids at 298.15 K and 101325 Pa, [m^3/mol]

rhoml_STPs [list of float] Molar densities of the chemicals in the mixture as liquids at 298.15 K and 101325 Pa, [mol/m^3]

Vmg_STPs [list of float] Molar volume of the chemicals in the mixture as gases at 298.15 K and 101325 Pa, [m^3/mol]

Vms_Tms [list of float] Molar volumes of solid phase at the melting point [m^3/mol]

rhos_Tms [list of float] Mass densities of solid phase at the melting point [kg/m^3]

Hvap_Tbms [list of float] Molar enthalpies of vaporization of the chemicals in the mixture at their normal boiling points, [J/mol]

Hvap_Tbs [list of float] Mass enthalpies of vaporization of the chemicals in the mixture at their normal boiling points, [J/kg]

`alpha` Thermal diffusivity of the mixture at its current temperature, pressure, and phase in units of [m^2/s].

`alphag` Thermal diffusivity of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

alphags Pure component thermal diffusivities of the chemicals in the mixture in the gas phase at the current temperature and pressure, in units of [m^2/s].

alphal Thermal diffusivity of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

alphals Pure component thermal diffusivities of the chemicals in the mixture in the liquid phase at the current temperature and pressure, in units of [m^2/s].

A Helmholtz energy of the mixture at its current state, in units of [J/kg].

Am Helmholtz energy of the mixture at its current state, in units of [J/mol].

atom_fractions Dictionary of atomic fractions for each atom in the mixture.

atom_fractionss List of dictionaries of atomic fractions for all chemicals in the mixture.

atomss List of dictionaries of atom counts for all chemicals in the mixture.

Bvirial Second virial coefficient of the gas phase of the mixture at its current temperature, pressure, and composition in units of [mol/m^3].

charges Charges for all chemicals in the mixture, [faraday].

Cp Mass heat capacity of the mixture at its current phase and temperature, in units of [J/kg/K].

Cpg Gas-phase heat capacity of the mixture at its current temperature , and composition in units of [J/kg/K].

Cpgm Gas-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cpgms Gas-phase ideal gas heat capacity of the chemicals at its current temperature, in units of [J/mol/K].

Cpgs Gas-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Cpl Liquid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/kg/K].

Cplm Liquid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cplms Liquid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Cpls Liquid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Cpm Molar heat capacity of the mixture at its current phase and temperature, in units of [J/mol/K].

Cps Solid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/kg/K].

Cpsm Solid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cpsms Solid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Cpss Solid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Cvg Gas-phase ideal-gas contant-volume heat capacity of the mixture at its current temperature, in units of [J/kg/K].

Cvgm Gas-phase ideal-gas contant-volume heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cvgms Gas-phase pure component ideal-gas contant-volume heat capacities of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Cvgs Gas-phase pure component ideal-gas contant-volume heat capacities of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

economic_statuses List of dictionaries of the economic status for all chemicals in the mixture.

eos Equation of state object held by the mixture.

formulas Chemical formulas for all chemicals in the mixture.

Hvapms Pure component enthalpies of vaporization of the chemicals in the mixture at its current temperature, in units of [J/mol].

Hvaps Enthalpy of vaporization of the chemicals in the mixture at its current temperature, in units of [J/kg].

InChI_Keys InChI keys for all chemicals in the mixture.

InChIs InChI strings for all chemicals in the mixture.

isentropic_exponent Gas-phase ideal-gas isentropic exponent of the mixture at its current temperature, [dimensionless].

isentropic_exponents Gas-phase pure component ideal-gas isentropic exponent of the chemicals in the mixture at its current temperature, [dimensionless].

isobaric_expansion Isobaric (constant-pressure) expansion of the mixture at its current phase, temperature, and pressure in units of [1/K].

isobaric_expansion_g Isobaric (constant-pressure) expansion of the gas phase of the mixture at its current temperature and pressure, in units of [1/K].

isobaric_expansion_gs Pure component isobaric (constant-pressure) expansions of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [1/K].

isobaric_expansion_l Isobaric (constant-pressure) expansion of the liquid phase of the mixture at its current temperature and pressure, in units of [1/K].

isobaric_expansion_ls Pure component isobaric (constant-pressure) expansions of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [1/K].

IUPAC_names IUPAC names for all chemicals in the mixture.

JT Joule Thomson coefficient of the mixture at its current phase, temperature, and pressure in units of [K/Pa].

JTg Joule Thomson coefficient of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [K/Pa].

JTgs Pure component Joule Thomson coefficients of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [K/Pa].

JTl Joule Thomson coefficient of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [K/Pa].

JTls Pure component Joule Thomson coefficients of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [K/Pa].

k Thermal conductivity of the mixture at its current phase, temperature, and pressure in units of [W/m/K].

kg Thermal conductivity of the mixture in the gas phase at its current temperature, pressure, and composition in units of [Pa*s].

kgs Pure component thermal conductivies of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [W/m/K].

kl Thermal conductivity of the mixture in the liquid phase at its current temperature, pressure, and composition in units of [Pa*s].

kls Pure component thermal conductivities of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [W/m/K].

legal_statuses List of dictionaries of the legal status for all chemicals in the mixture.

mass_fractions Dictionary of mass fractions for each atom in the mixture.

mass_fractionss List of dictionaries of mass fractions for all chemicals in the mixture.

mu Viscosity of the mixture at its current phase, temperature, and pressure in units of [Pa*s].

mug Viscosity of the mixture in the gas phase at its current temperature, pressure, and composition in units of [Pa*s].

mugs Pure component viscosities of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [Pa*s].

mul Viscosity of the mixture in the liquid phase at its current temperature, pressure, and composition in units of [Pa*s].

muls Pure component viscosities of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [Pa*s].

nu Kinematic viscosity of the the mixture at its current temperature, pressure, and phase in units of [m^2/s].

nug Kinematic viscosity of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

nugs Pure component kinematic viscosities of the gas phase of the chemicals in the mixture at its current temperature and pressure, in units of [m^2/s].

nul Kinematic viscosity of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

nuls Pure component kinematic viscosities of the liquid phase of the chemicals in the mixture at its current temperature and pressure, in units of [m^2/s].

permittivites Pure component relative permittivities of the chemicals in the mixture at its current temperature, [dimensionless].

Pr Prandtl number of the mixture at its current temperature, pressure, and phase; [dimensionless].

Prg Prandtl number of the gas phase of the mixture if one exists at its current temperature and pressure, [dimensionless].

Prgs Pure component Prandtl numbers of the gas phase of the chemicals in the mixture at its current temperature and pressure, [dimensionless].

Prl Prandtl number of the liquid phase of the mixture if one exists at its current temperature and pressure, [dimensionless].

Prls Pure component Prandtl numbers of the liquid phase of the chemicals in the mixture at its current temperature and pressure, [dimensionless].

Psats Pure component vapor pressures of the chemicals in the mixture at its current temperature, in units of [Pa].

PSRK_groups List of dictionaries of PSRK subgroup: count groups for each chemical in the mixture.

PubChems PubChem Component ID numbers for all chemicals in the mixture.

rho Mass density of the mixture at its current phase and temperature and pressure, in units of [kg/m^3].

rhog Gas-phase mass density of the mixture at its current temperature, pressure, and composition in units of [kg/m^3].

rhogm Molar density of the mixture in the gas phase at the current temperature, pressure, and composition in units of [mol/m^3].

rhogms Pure component molar densities of the chemicals in the gas phase at the current temperature and pressure, in units of [mol/m^3].

rhogm_STP Molar density of the mixture in the gas phase at 298.15 K and 101.325 kPa, and the current composition, in units of [mol/m^3].

rhogs Pure-component gas-phase mass densities of the chemicals in the mixture at its current temperature and pressure, in units of [kg/m^3].

rhog_STP Gas-phase mass density of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [kg/m^3].

rhol Liquid-phase mass density of the mixture at its current temperature, pressure, and composition in units of [kg/m^3].

rholm Molar density of the mixture in the liquid phase at the current temperature, pressure, and composition in units of [mol/m^3].

rholms Pure component molar densities of the chemicals in the mixture in the liquid phase at the current temperature and pressure, in units of [mol/m^3].

rholm_STP Molar density of the mixture in the liquid phase at 298.15 K and 101.325 kPa, and the current composition, in units of [mol/m^3].

rhols Pure-component liquid-phase mass density of the chemicals in the mixture at its current temperature and pressure, in units of [kg/m^3].

rhol_STP Liquid-phase mass density of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [kg/m^3].

rhom Molar density of the mixture at its current phase and temperature and pressure, in units of [mol/m^3].

rhosms Pure component molar densities of the chemicals in the solid phase at the current temperature and pressure, in units of [mol/m^3].

rhoss Pure component solid-phase mass density of the chemicals in the mixture at its current temperature, in units of [kg/m^3].

ringss List of ring counts for all chemicals in the mixture.

sigma Surface tension of the mixture at its current temperature and composition, in units of [N/m].

sigmas Pure component surface tensions of the chemicals in the mixture at its current temperature, in units of [N/m].

smiless SMILES strings for all chemicals in the mixture.

solubility_parameters Pure component solubility parameters of the chemicals in the mixture at its current temperature and pressure, in units of [Pa^0.5].

synonymss Lists of synonyms for all chemicals in the mixture.

U Internal energy of the mixture at its current state, in units of [J/kg].

Um Internal energy of the mixture at its current state, in units of [J/mol].

UNIFAC_Dortmund_groups List of dictionaries of Dortmund UNIFAC subgroup: count groups for each chemcial in the mixture.

UNIFAC_groups List of dictionaries of UNIFAC subgroup: count groups for each chemical in the mixture.

Vm Molar volume of the mixture at its current phase and temperature and pressure, in units of [m^3/mol].

Vmg Gas-phase molar volume of the mixture at its current temperature, pressure, and composition in units of [m^3/mol].

Vmgs Pure component gas-phase molar volumes of the chemicals in the mixture at its current temperature and pressure, in units of [m^3/mol].

Vmg_STP Gas-phase molar volume of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [m^3/mol].

Vml Liquid-phase molar volume of the mixture at its current temperature, pressure, and composition in units of [m^3/mol].

Vmls Pure component liquid-phase molar volumes of the chemicals in the mixture at its current temperature and pressure, in units of [m^3/mol].

Vml_STP Liquid-phase molar volume of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [m^3/mol].

Vmss Pure component solid-phase molar volumes of the chemicals in the mixture at its current temperature, in units of [m^3/mol].

Z Compressibility factor of the mixture at its current phase and temperature and pressure, [dimensionless].

Zg Compressibility factor of the mixture in the gas phase at the current temperature, pressure, and composition, [dimensionless].

Zgs Pure component compressibility factors of the chemicals in the mixture in the gas phase at the current temperature and pressure, [dimensionless].

Zg_STP Gas-phase compressibility factor of the mixture at 298.15 K and 101.325 kPa, and the current composition, [dimensionless].

Zl Compressibility factor of the mixture in the liquid phase at the current temperature, pressure, and composition, [dimensionless].

Zls Pure component compressibility factors of the chemicals in the liquid phase at the current temperature and pressure, [dimensionless].

Zl_STP Liquid-phase compressibility factor of the mixture at 298.15 K and 101.325 kPa, and the current composition, [dimensionless].

Zss Pure component compressibility factors of the chemicals in the mixture in the solid phase at the current temperature and pressure, [dimensionless].

Methods

Hc_volumetric_g([T, P])	Standard higher molar heat of combustion of the mixture, in units of [J/m^3] at the specified T and P in the gas phase.
Hc_volumetric_g_lower([T, P])	Standard lower molar heat of combustion of the mixture, in units of [J/m^3] at the specified T and P in the gas phase.
Vfgs([T, P])	Volume fractions of all species in a hypothetical pure-gas phase at the current or specified temperature and pressure.
Vfls([T, P])	Volume fractions of all species in a hypothetical pure-liquid phase at the current or specified temperature and pressure.
draw_2d([Hs])	Interface for drawing a 2D image of all the molecules in the mixture.
set_chemical_TP([T, P])	Basic method to change all chemical instances to be at the T and P specified.
set_chemical_constants()	Basic method which retrieves and sets constants of chemicals to be accessible as lists from a Mixture object.

Bond	
Capillary	
Grashof	
Jakob	
Peclet_heat	
Reynolds	
Weber	
compound_index	
eos_pures	
flash_caloric	
properties	
set_Chemical_property_objects	
set_TP_sources	
set_constant_sources	
set_constants	
set_eos	
set_property_package	

property A

Helmholtz energy of the mixture at its current state, in units of [J/kg].

This property requires that the property package of the mixture found a solution to the given state variables. It also depends on the molar volume of the mixture at its current conditions.

property API

API gravity of the hypothetical liquid phase of the mixture, [degrees]. The reference condition is water at 15.6 °C (60 °F) and 1 atm (rho=999.016 kg/m^3, standardized).

Examples

```
>>> Mixture(['hexane', 'decane'], ws=[0.5, 0.5]).API
71.34707841728181
```

property Am

Helmholtz energy of the mixture at its current state, in units of [J/mol].

This property requires that the property package of the mixture found a solution to the given state variables. It also depends on the molar volume of the mixture at its current conditions.

Bond(*L=None*)

property Bvirial

Second virial coefficient of the gas phase of the mixture at its current temperature, pressure, and composition in units of [mol/m^3].

This property uses the object-oriented interface *thermo.volume.VolumeGasMixture*, converting its result with `thermo.utils.B_from_Z`.

Examples

```
>>> Mixture(['hexane'], ws=[1], T=300, P=1E5).Bvirial
-0.001486976173801296
```

Capillary(*V=None*)

property Cp

Mass heat capacity of the mixture at its current phase and temperature, in units of [J/kg/K].

Examples

```
>>> w = Mixture(['water'], ws=[1])
>>> w.Cp, w.phase
(4180.597021827336, 'l')
>>> Pd = Mixture(['palladium'], ws=[1])
>>> Pd.Cp, Pd.phase
(234.26767209171211, 's')
```

property Cpg

Gas-phase heat capacity of the mixture at its current temperature , and composition in units of [J/kg/K]. For calculation of this property at other temperatures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityGasMixture*; each Mixture instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Mixture(['oxygen', 'nitrogen'], ws=[.4, .6], T=350, P=1E6).Cpg
995.8911053614883
```

property Cpgm

Gas-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K]. For calculation of this property at other temperatures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityGasMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['oxygen', 'nitrogen'], ws=[.4, .6], T=350, P=1E6).Cpgm
29.361044582498046
```

property Cpgms

Gas-phase ideal gas heat capacity of the chemicals at its current temperature, in units of [J/mol/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cpgms
[89.55804092586159, 111.70390334788907]
```

property Cpgs

Gas-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cpgs
[1146.5360555565146, 1212.3488046342566]
```

property Cpl

Liquid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/kg/K]. For calculation of this property at other temperatures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityLiquidMixture*; each Mixture instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Mixture(['water', 'sodium chloride'], ws=[.9, .1], T=301.5).Cpl
3735.4604049449786
```

property Cplm

Liquid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K]. For calculation of this property at other temperatures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacityLiquidMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['toluene', 'decane'], ws=[.9, .1], T=300).Cplm
168.29127923518843
```

property Cplms

Liquid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cplms
[140.9113971170526, 163.62584810669068]
```

property Cpls

Liquid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cpls
[1803.9697581961016, 1775.869915141704]
```

property Cpm

Molar heat capacity of the mixture at its current phase and temperature, in units of [J/mol/K]. Available only if single phase.

Examples

```
>>> Mixture(['ethylbenzene'], ws=[1], T=550, P=3E6).Cpm
294.18449553310046
```

property Cps

Solid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/kg/K]. For calculation of this property at other temperatures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacitySolidMixture*; each Mixture instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Mixture(['silver', 'platinum'], ws=[0.95, 0.05]).Cps
229.55166388430328
```

property Cpsm

Solid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K]. For calculation of this property at other temperatures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.heat_capacity.HeatCapacitySolidMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['silver', 'platinum'], ws=[0.95, 0.05]).Cpsm
25.32745796347474
```

property Cpsms
Solid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cpsms
[109.77384365511931, 135.22614707678474]
```

property Cpss
Solid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cpss
[1405.341925822248, 1467.6412627521154]
```

property Cvg
Gas-phase ideal-gas contant-volume heat capacity of the mixture at its current temperature, in units of [J/kg/K]. Subtracts R from the ideal-gas heat capacity; does not include pressure-compensation from an equation of state.

Examples

```
>>> Mixture(['water'], ws=[1], T=520).Cvg
1506.1471795798861
```

property Cvgm
Gas-phase ideal-gas contant-volume heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K]. Subtracts R from the ideal-gas heat capacity; does not include pressure-compensation from an equation of state.

Examples

```
>>> Mixture(['water'], ws=[1], T=520).Cvgm
27.13366316134193
```

property Cvgms
Gas-phase pure component ideal-gas contant-volume heat capacities of the chemicals in the mixture at its current temperature, in units of [J/mol/K]. Subtracts R from the ideal-gas heat capacities; does not include pressure-compensation from an equation of state.

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cvgms
[81.2435811258616, 103.38944354788907]
```

property Cvgs

Gas-phase pure component ideal-gas contant-volume heat capacities of the chemicals in the mixture at its current temperature, in units of [J/kg/K]. Subtracts R from the ideal-gas heat capacity; does not include pressure-compensation from an equation of state.

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Cvgs
[1040.093040003431, 1122.1100117398266]
```

Grashof(*Tw=None*, *L=None*)

H = None

property Hc

Standard higher heat of combustion of the mixture, in units of [J/kg].

This property depends on the bulk composition only.

property Hc_lower

Standard lower heat of combustion of the mixture, in units of [J/kg].

This property depends on the bulk composition only.

Hc_volumetric_g(*T=288.7055555555555*, *P=101325.0*)

Standard higher molar heat of combustion of the mixture, in units of [J/m^3] at the specified T and P in the gas phase.

This property depends on the bulk composition only.

Parameters

T [float, optional] Reference temperature, [K]

P [float, optional] Reference pressure, [Pa]

Returns

Hc_volumetric_g [float, optional] Higher heat of combustion on a volumetric basis, [J/m^3]

Hc_volumetric_g_lower(*T=288.7055555555555*, *P=101325.0*)

Standard lower molar heat of combustion of the mixture, in units of [J/m^3] at the specified T and P in the gas phase.

This property depends on the bulk composition only.

Parameters

T [float, optional] Reference temperature, [K]

P [float, optional] Reference pressure, [Pa]

Returns

Hc_volumetric_g [float, optional] Lower heat of combustion on a volumetric basis, [J/m^3]

property Hcm
Standard higher molar heat of combustion of the mixture, in units of [J/mol].

This property depends on the bulk composition only.

property Hcm_lower
Standard lower molar heat of combustion of the mixture, in units of [J/mol].

This property depends on the bulk composition only.

Hm = None

property Hvapms
Pure component enthalpies of vaporization of the chemicals in the mixture at its current temperature, in units of [J/mol].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Hvapms
[32639.806783391632, 36851.7902195611]
```

property Hvaps
Enthalpy of vaporization of the chemicals in the mixture at its current temperature, in units of [J/kg].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Hvaps
[417859.9144942896, 399961.16950519773]
```

property IUPAC_names
IUPAC names for all chemicals in the mixture.

Examples

```
>>> Mixture(['1-hexene', '1-nonene'], zs=[.7, .3]).IUPAC_names
['hex-1-ene', 'non-1-ene']
```

property InChI_Keys
InChI keys for all chemicals in the mixture.

Examples

```
>>> Mixture(['1-nonene'], zs=[1]).InChI_Keys
['JRZJOMJEPLMPRA-UHFFFAOYSA-N']
```

property InChIs
InChI strings for all chemicals in the mixture.

Examples

```
>>> Mixture(['methane', 'ethane', 'propane', 'butane'],
... zs=[0.25, 0.25, 0.25, 0.25]).InChIs
['CH4/h1H4', 'C2H6/c1-2/h1-2H3', 'C3H8/c1-3-2/h3H2,1-2H3', 'C4H10/c1-3-4-2/h3-
↪4H2,1-2H3']
```

property JT

Joule Thomson coefficient of the mixture at its current phase, temperature, and pressure in units of [K/Pa]. Available only if single phase.

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Examples

```
>>> Mixture(['water'], ws=[1]).JT
-2.2150394958666412e-07
```

property JTg

Joule Thomson coefficient of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Examples

```
>>> Mixture(['dodecane'], ws=[1], T=400, P=1000).JTg
5.4089897835384913e-05
```

property JTgs

Pure component Joule Thomson coefficients of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).JTgs
[6.0940046688790938e-05, 4.1290005523287549e-05]
```

property JTl

Joule Thomson coefficient of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Examples

```
>>> Mixture(['dodecane'], ws=[1], T=400).JTl
-3.193910574559279e-07
```

property JTls

Pure component Joule Thomson coefficients of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [K/Pa].

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).JTls
[-3.8633730709853161e-07, -3.464395792560331e-07]
```

Jakob(*Tw=None*)

property PSRK_groups

List of dictionaries of PSRK subgroup: count groups for each chemical in the mixture. Uses the PSRK subgroups, as determined by DDBST's online service.

Examples

```
>>> Mixture(['1-pentanol', 'decane'], ws=[0.5, 0.5]).PSRK_groups
[{1: 1, 2: 4, 14: 1}, {1: 2, 2: 8}]
```

P_default = 101325.0

property Parachor

Parachor of the mixture at its current temperature and pressure, in units of [N^0.25*m^2.75/mol].

$$P = \frac{\sigma^{0.25} MW}{\rho_L - \rho_V}$$

Calculated based on surface tension, density of the liquid and gas phase, and molecular weight. For uses of this property, see `thermo.utils.Parachor`.

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).Parachor
4.233407085050756e-05
```

property Parachors

Pure component Parachor parameters of the chemicals in the mixture at its current temperature and pressure, in units of [N^0.25*m^2.75/mol].

$$P = \frac{\sigma^{0.25} MW}{\rho_L - \rho_V}$$

Calculated based on surface tension, density of the liquid and gas phase, and molecular weight. For uses of this property, see `thermo.utils.Parachor`.

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).Parachors
[3.6795616000855504e-05, 4.82947303150274e-05]
```

property Pbubble

Bubble point pressure of the mixture at its current temperature and composition, in units of [Pa].

This property requires that the property package of the mixture found a solution to the given state variables.

property Pdew

Dew point pressure of the mixture at its current temperature and composition, in units of [Pa].

This property requires that the property package of the mixture found a solution to the given state variables.

Peclet_heat(*V=None*, *D=None*)

property Pr

Prandtl number of the mixture at its current temperature, pressure, and phase; [dimensionless]. Available only if single phase.

$$Pr = \frac{C_p \mu}{k}$$

Examples

```
>>> Mixture(['acetone'], ws=[1]).Pr
4.183039103542711
```

property Prg

Prandtl number of the gas phase of the mixture if one exists at its current temperature and pressure, [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Examples

```
>>> Mixture(['NH3'], ws=[1]).Prg
0.8472637319330079
```

property Prgs

Pure component Prandtl numbers of the gas phase of the chemicals in the mixture at its current temperature and pressure, [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).Prgs
[0.7810364900059606, 0.784358381123896]
```

property Prl

Prandtl number of the liquid phase of the mixture if one exists at its current temperature and pressure, [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Examples

```
>>> Mixture(['nitrogen'], ws=[1], T=70).Prl
2.782821450148889
```

property Prls

Pure component Prandtl numbers of the liquid phase of the chemicals in the mixture at its current temperature and pressure, [dimensionless].

$$Pr = \frac{C_p \mu}{k}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).Prls
[6.13542244155373, 5.034355147908088]
```

property Psats

Pure component vapor pressures of the chemicals in the mixture at its current temperature, in units of [Pa].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Psats
[32029.25774454549, 10724.419010511821]
```

property PubChems

PubChem Component ID numbers for all chemicals in the mixture.

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5]).PubChems
[241, 1140]
```

property R_specific

Specific gas constant of the mixture, in units of [J/kg/K].

Examples

```
>>> Mixture(['N2', 'O2'], zs=[0.79, .21]).R_specific
288.1928437986195
```

Reynolds(*V=None*, *D=None*)

property SG
Specific gravity of the mixture, [dimensionless].

For gas-phase conditions, this is calculated at 15.6 °C (60 °F) and 1 atm for the mixture and the reference fluid, air. For liquid and solid phase conditions, this is calculated based on a reference fluid of water at 4°C at 1 atm, but the with the liquid or solid mixture's density at the currently specified conditions.

Examples

```
>>> Mixture('MTBE').SG
0.7428160596603596
```

property SGg
Specific gravity of a hypothetical gas phase of the mixture, . [dimensionless]. The reference condition is air at 15.6 °C (60 °F) and 1 atm (rho=1.223 kg/m^3). The definition for gases uses the compressibility factor of the reference gas and the mixture both at the reference conditions, not the conditions of the mixture.

Examples

```
>>> Mixture('argon').SGg
1.3800407778218216
```

property SGl
Specific gravity of a hypothetical liquid phase of the mixture at the specified temperature and pressure, [dimensionless]. The reference condition is water at 4 °C and 1 atm (rho=999.017 kg/m^3). For liquids, SG is defined that the reference chemical's T and P are fixed, but the chemical itself varies with the specified T and P.

Examples

```
>>> Mixture('water', ws=[1], T=365).SGl
0.9650065522428539
```

property SGs
Specific gravity of a hypothetical solid phase of the mixture at the specified temperature and pressure, [dimensionless]. The reference condition is water at 4 °C and 1 atm (rho=999.017 kg/m^3). The SG varries with temperature and pressure but only very slightly.

T_default = 298.15

property Tbubble
Bubble point temperature of the mixture at its current pressure and composition, in units of [K].

This property requires that the property package of the mixture found a solution to the given state variables.

property Tdew

Dew point temperature of the mixture at its current pressure and composition, in units of [K].

This property requires that the property package of the mixture found a solution to the given state variables.

property U

Internal energy of the mixture at its current state, in units of [J/kg].

This property requires that the property package of the mixture found a solution to the given state variables. It also depends on the molar volume of the mixture at its current conditions.

property UNIFAC_Dortmund_groups

List of dictionaries of Dortmund UNIFAC subgroup: count groups for each chemcial in the mixture. Uses the Dortmund UNIFAC subgroups, as determined by DDBST's online service.

Examples

```
>>> Mixture(['1-pentanol', 'decane'], ws=[0.5, 0.5]).UNIFAC_Dortmund_groups
[{1: 1, 2: 4, 14: 1}, {1: 2, 2: 8}]
```

property UNIFAC_Qs

UNIFAC Q (normalized Van der Waals area) values, dimensionless. Used in the UNIFAC model.

Examples

```
>>> Mixture(['o-xylene', 'decane'], zs=[.5, .5]).UNIFAC_Qs
[3.536, 6.016]
```

property UNIFAC_Rs

UNIFAC R (normalized Van der Waals volume) values, dimensionless. Used in the UNIFAC model.

Examples

```
>>> Mixture(['o-xylene', 'm-xylene'], zs=[.5, .5]).UNIFAC_Rs
[4.6578, 4.6578]
```

property UNIFAC_groups

List of dictionaries of UNIFAC subgroup: count groups for each chemical in the mixture. Uses the original UNIFAC subgroups, as determined by DDBST's online service.

Examples

```
>>> Mixture(['1-pentanol', 'decane'], ws=[0.5, 0.5]).UNIFAC_groups
[{1: 1, 2: 4, 14: 1}, {1: 2, 2: 8}]
```

property Um

Internal energy of the mixture at its current state, in units of [J/mol].

This property requires that the property package of the mixture found a solution to the given state variables. It also depends on the molar volume of the mixture at its current conditions.

V_over_F = None

property Van_der_Waals_areas

List of unnormalized Van der Waals areas of all the chemicals in the mixture, in units of [m^2/mol].

Examples

```
>>> Mixture(['1-pentanol', 'decane'], ws=[0.5, 0.5]).Van_der_Waals_areas
[1052000.0, 1504000.0]
```

property Van_der_Waals_volumes

List of unnormalized Van der Waals volumes of all the chemicals in the mixture, in units of [m^3/mol].

Examples

```
>>> Mixture(['1-pentanol', 'decane'], ws=[0.5, 0.5]).Van_der_Waals_volumes
[6.9762279e-05, 0.00010918455800000001]
```

Vfgs(*T=None*, *P=None*)

Volume fractions of all species in a hypothetical pure-gas phase at the current or specified temperature and pressure. If temperature or pressure are specified, the non-specified property is assumed to be that of the mixture. Note this is a method, not a property. Volume fractions are calculated based on **pure species volumes only**.

Examples

```
>>> Mixture(['sulfur hexafluoride', 'methane'], zs=[.2, .9], T=315).Vfgs()
[0.18062059238682632, 0.8193794076131737]
```

```
>>> S = Mixture(['sulfur hexafluoride', 'methane'], zs=[.1, .9])
>>> S.Vfgs(P=1E2)
[0.0999987466608421, 0.9000012533391578]
```

Vfls(*T=None*, *P=None*)

Volume fractions of all species in a hypothetical pure-liquid phase at the current or specified temperature and pressure. If temperature or pressure are specified, the non-specified property is assumed to be that of the mixture. Note this is a method, not a property. Volume fractions are calculated based on **pure species volumes only**.

Examples

```
>>> Mixture(['hexane', 'pentane'], zs=[.5, .5], T=315).Vfls()
[0.5299671144566751, 0.47003288554332484]
```

```
>>> S = Mixture(['hexane', 'decane'], zs=[0.25, 0.75])
>>> S.Vfls(298.16, 101326)
[0.18301434895886864, 0.8169856510411313]
```

property Vm

Molar volume of the mixture at its current phase and temperature and pressure, in units of [m^3/mol]. Available only if single phase.

Examples

```
>>> Mixture(['ethylbenzene'], ws=[1], T=550, P=3E6).Vm
0.00017758024401627633
```

property Vmg

Gas-phase molar volume of the mixture at its current temperature, pressure, and composition in units of [m^3/mol]. For calculation of this property at other temperatures or pressures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeGasMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['hexane'], ws=[1], T=300, P=2E5).Vmg
0.010888694235142216
```

property Vmg_STP

Gas-phase molar volume of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [m^3/mol].

Examples

```
>>> Mixture(['nitrogen'], ws=[1]).Vmg_STP
0.02445443688838904
```

property Vmgs

Pure component gas-phase molar volumes of the chemicals in the mixture at its current temperature and pressure, in units of [m^3/mol].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Vmgs
[0.024929001982294974, 0.024150186467130488]
```

property Vml

Liquid-phase molar volume of the mixture at its current temperature, pressure, and composition in units of [m^3/mol]. For calculation of this property at other temperatures or pressures or compositions, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeLiquidMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['cyclobutane'], ws=[1], T=225).Vml
7.42395423425395e-05
```

property Vml_STP

Liquid-phase molar volume of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [m^3/mol].

Examples

```
>>> Mixture(['cyclobutane'], ws=[1]).Vml_STP
8.143327329133706e-05
```

property Vmls

Pure component liquid-phase molar volumes of the chemicals in the mixture at its current temperature and pressure, in units of [m^3/mol].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Vmls
[9.188896727673715e-05, 0.00010946199496993461]
```

Vms = None

property Vmss

Pure component solid-phase molar volumes of the chemicals in the mixture at its current temperature, in units of [m^3/mol].

Examples

```
>>> Mixture(['iron'], ws=[1], T=320).Vmss
[7.09593392630242e-06]
```

Weber(*V=None*, *D=None*)

property Z

Compressibility factor of the mixture at its current phase and temperature and pressure, [dimensionless]. Available only if single phase.

Examples

```
>>> Mixture(['MTBE'], ws=[1], T=900, P=1E-2).Z
0.9999999999056374
```

property Zg

Compressibility factor of the mixture in the gas phase at the current temperature, pressure, and composition, [dimensionless].

Utilizes the object oriented interface and *thermo.volume.VolumeGasMixture* to perform the actual calculation of molar volume.

Examples

```
>>> Mixture(['hexane'], ws=[1], T=300, P=1E5).Zg
0.9403859376888885
```

property Zg_STP
Gas-phase compressibility factor of the mixture at 298.15 K and 101.325 kPa, and the current composition, [dimensionless].

Examples

```
>>> Mixture(['nitrogen'], ws=[1]).Zg_STP
0.9995520809691023
```

property Zgs
Pure component compressibility factors of the chemicals in the mixture in the gas phase at the current temperature and pressure, [dimensionless].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Zgs
[0.9493743379816593, 0.9197146081359057]
```

property Zl
Compressibility factor of the mixture in the liquid phase at the current temperature, pressure, and composition, [dimensionless].

Utilizes the object oriented interface and *thermo.volume.VolumeLiquidMixture* to perform the actual calculation of molar volume.

Examples

```
>>> Mixture(['water'], ws=[1]).Zl
0.0007385375470263454
```

property Zl_STP
Liquid-phase compressibility factor of the mixture at 298.15 K and 101.325 kPa, and the current composition, [dimensionless].

Examples

```
>>> Mixture(['cyclobutane'], ws=[1]).Zl_STP
0.0033285083663950068
```

property Zls
Pure component compressibility factors of the chemicals in the liquid phase at the current temperature and pressure, [dimensionless].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).Zls
[0.003494191720201235, 0.004168655010037687]
```

property Zss

Pure component compressibility factors of the chemicals in the mixture in the solid phase at the current temperature and pressure, [dimensionless].

Examples

```
>>> Mixture(['palladium'], ws=[1]).Zss
[0.00036248477437931853]
```

property alpha

Thermal diffusivity of the mixture at its current temperature, pressure, and phase in units of [m^2/s]. Available only if single phase.

$$\alpha = \frac{k}{\rho Cp}$$

Examples

```
>>> Mixture(['furfural'], ws=[1]).alpha
8.696537158635412e-08
```

property alphag

Thermal diffusivity of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Examples

```
>>> Mixture(['ammonia'], ws=[1]).alphag
1.6968517002221566e-05
```

property alphags

Pure component thermal diffusivities of the chemicals in the mixture in the gas phase at the current temperature and pressure, in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).alphags
[3.3028044028118324e-06, 2.4412958544059014e-06]
```

property alphal

Thermal diffusivity of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Examples

```
>>> Mixture(['nitrogen'], ws=[1], T=70).alphal
9.444949636299626e-08
```

property alphals

Pure component thermal diffusivities of the chemicals in the mixture in the liquid phase at the current temperature and pressure, in units of [m^2/s].

$$\alpha = \frac{k}{\rho Cp}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).alphals
[8.732683564481583e-08, 7.57355434073289e-08]
```

property atom_fractions

Dictionary of atomic fractions for each atom in the mixture.

Examples

```
>>> Mixture(['CO2', 'O2'], zs=[0.5, 0.5]).atom_fractions
{'C': 0.2, 'O': 0.8}
```

property atom_fractionss

List of dictionaries of atomic fractions for all chemicals in the mixture.

Examples

```
>>> Mixture(['oxygen', 'nitrogen'], zs=[.5, .5]).atom_fractionss
[{'O': 1.0}, {'N': 1.0}]
```

property atoms

Mole-averaged dictionary of atom counts for all atoms of the chemicals in the mixture.

Examples

```
>>> Mixture(['nitrogen', 'oxygen'], zs=[.01, .99]).atoms
{'O': 1.98, 'N': 0.02}
```

property atomss

List of dictionaries of atom counts for all chemicals in the mixture.

Examples

```
>>> Mixture(['nitrogen', 'oxygen'], zs=[.01, .99]).atomss
[{'N': 2}, {'O': 2}]
```

autoflash = True

property charge_balance

Charge imbalance of the mixture, in units of [faraday]. Mixtures meeting the electroneutrality condition will have an imbalance of 0.

Examples

```
>>> Mixture(['Na+', 'Cl-', 'water'], zs=[.01, .01, .98]).charge_balance
0.0
```

property charges

Charges for all chemicals in the mixture, [faraday].

Examples

```
>>> Mixture(['water', 'sodium ion', 'chloride ion'], zs=[.9, .05, .05]).charges
[0, 1, -1]
```

compound_index(*CAS*)

conductivity = None

property constants

Returns a :obj:`thermo.chemical_package.ChemicalConstantsPackage instance with constants from the mixture, [-].

draw_2d(*Hs=False*)

Interface for drawing a 2D image of all the molecules in the mixture. Requires an HTML5 browser, and the libraries RDKit and IPython. An exception is raised if either of these libraries is absent.

Parameters

Hs [bool] Whether or not to show hydrogen

Examples

Mixture([‘natural gas’]).draw_2d()

property economic_statuses
List of dictionaries of the economic status for all chemicals in the mixture.

Examples

```
>>> Mixture(['o-xylene', 'm-xylene'], zs=[.5, .5]).economic_statuses
[["US public: {'Manufactured': 0.0, 'Imported': 0.0, 'Exported': 0.0}",
  u'100,000 - 1,000,000 tonnes per annum',
  'OECD HPV Chemicals'],
 ["US public: {'Manufactured': 39.805, 'Imported': 0.0, 'Exported': 0.0}",
  u'100,000 - 1,000,000 tonnes per annum',
  'OECD HPV Chemicals']]
```

property eos
Equation of state object held by the mixture. See : obj:*thermo.eos_mix* for a full listing.

eos_in_a_box = []

eos_pures(*eos=<class 'thermo.eos.PR'>*, *T=None*, *P=None*)

flash_caloric(*T=None*, *P=None*, *VF=None*, *Hm=None*, *Sm=None*, *H=None*, *S=None*)

flashed = True

property formulas
Chemical formulas for all chemicals in the mixture.

Examples

```
>>> Mixture(['ethanol', 'trichloroethylene', 'furfuryl alcohol'],
... ws=[0.5, 0.2, 0.3]).formulas
['C2H6O', 'C2HCl3', 'C5H6O2']
```

property isentropic_exponent
Gas-phase ideal-gas isentropic exponent of the mixture at its current temperature, [dimensionless]. Does not include pressure-compensation from an equation of state.

Examples

```
>>> Mixture(['hydrogen'], ws=[1]).isentropic_exponent
1.405237786321222
```

property isentropic_exponents
Gas-phase pure component ideal-gas isentropic exponent of the chemicals in the mixture at its current temperature, [dimensionless].

Does not include pressure-compensation from an equation of state.

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).isentropic_exponents
[1.1023398979313739, 1.080418846592871]
```

property isobaric_expansion

Isobaric (constant-pressure) expansion of the mixture at its current phase, temperature, and pressure in units of [1/K]. Available only if single phase.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Examples

```
>>> Mixture(['water'], ws=[1], T=647.1, P=22048320.0).isobaric_expansion
0.34074205839222449
```

property isobaric_expansion_g

Isobaric (constant-pressure) expansion of the gas phase of the mixture at its current temperature and pressure, in units of [1/K]. Available only if single phase.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Examples

```
>>> Mixture(['argon'], ws=[1], T=647.1, P=22048320.0).isobaric_expansion_g
0.0015661100323025273
```

property isobaric_expansion_gs

Pure component isobaric (constant-pressure) expansions of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [1/K].

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).isobaric_expansion_gs
[0.0038091518363900499, 0.0043556759306508453]
```

property isobaric_expansion_l

Isobaric (constant-pressure) expansion of the liquid phase of the mixture at its current temperature and pressure, in units of [1/K]. Available only if single phase.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Examples

```
>>> Mixture(['argon'], ws=[1], T=647.1, P=22048320.0).isobaric_expansion_l
0.001859152875154442
```

property isobaric_expansion_ls

Pure component isobaric (constant-pressure) expansions of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [1/K].

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).isobaric_expansion_ls
[0.0012736035771253886, 0.0011234157437069571]
```

property k

Thermal conductivity of the mixture at its current phase, temperature, and pressure in units of [W/m/K]. Available only if single phase.

Examples

```
>>> Mixture(['ethanol'], ws=[1], T=300).kl
0.16313594741877802
```

property kg

Thermal conductivity of the mixture in the gas phase at its current temperature, pressure, and composition in units of [Pa*s].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.thermal_conductivity.ThermalConductivityGasMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['water'], ws=[1], T=500).kg
0.036035173297862676
```

property kgs

Pure component thermal conductivies of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [W/m/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).kgs
[0.011865404482987936, 0.010981336502491088]
```

property kl

Thermal conductivity of the mixture in the liquid phase at its current temperature, pressure, and composition in units of [Pa*s].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.thermal_conductivity.ThermalConductivityLiquidMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['water'], ws=[1], T=320).kl
0.6369957248212118
```

property kls

Pure component thermal conductivities of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [W/m/K].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).kls
[0.13391538485205587, 0.12429339088930591]
```

ks = None

property legal_statuses

List of dictionaries of the legal status for all chemicals in the mixture.

Examples

```
>>> Mixture(['oxygen', 'nitrogen'], zs=[.5, .5]).legal_statuses
[{'DSL': 'LISTED',
  'EINECS': 'LISTED',
  'NLP': 'UNLISTED',
  'SPIN': 'LISTED',
  'TSCA': 'LISTED'},
 {'DSL': 'LISTED',
  'EINECS': 'LISTED',
  'NLP': 'UNLISTED',
  'SPIN': 'LISTED',
  'TSCA': 'LISTED'}]
```

property mass_fractions

Dictionary of mass fractions for each atom in the mixture.

Examples

```
>>> Mixture(['CO2', 'O2'], zs=[0.5, 0.5]).mass_fractions
{'C': 0.15801826905745822, 'O': 0.8419817309425419}
```

property mass_fractionss
List of dictionaries of mass fractions for all chemicals in the mixture.

Examples

```
>>> Mixture(['oxygen', 'nitrogen'], zs=[.5, .5]).mass_fractionss
[{'O': 1.0}, {'N': 1.0}]
```

property mu
Viscosity of the mixture at its current phase, temperature, and pressure in units of [Pa*s]. Available only if single phase.

Examples

```
>>> Mixture(['ethanol'], ws=[1], T=400).mu
1.1853097849748213e-05
```

property mug
Viscosity of the mixture in the gas phase at its current temperature, pressure, and composition in units of [Pa*s].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.viscosity.ViscosityGasMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['water'], ws=[1], T=500).mug
1.7298722343367148e-05
```

property mugs
Pure component viscosities of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [Pa*s].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).mugs
[8.082880451060605e-06, 7.442602145854158e-06]
```

property mul
Viscosity of the mixture in the liquid phase at its current temperature, pressure, and composition in units of [Pa*s].

For calculation of this property at other temperatures and pressures, or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.viscosity.ViscosityLiquidMixture*; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['water'], ws=[1], T=320).mul
0.0005767262693751547
```

property muls

Pure component viscosities of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [Pa*s].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).muls
[0.00045545522798131764, 0.00043274394349114754]
```

property nu

Kinematic viscosity of the the mixture at its current temperature, pressure, and phase in units of [m^2/s]. Available only if single phase.

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> Mixture(['argon'], ws=[1]).nu
1.3842643382482236e-05
```

property nug

Kinematic viscosity of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> Mixture(['methane'], ws=[1], T=115).nug
2.5118460023343146e-06
```

property nugs

Pure component kinematic viscosities of the gas phase of the chemicals in the mixture at its current temperature and pressure, in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).nugs
[5.357870271650772e-07, 3.8127962283230277e-07]
```

property nul

Kinematic viscosity of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> Mixture(['methane'], ws=[1], T=110).nul
2.858088468937333e-07
```

property nuls

Pure component kinematic viscosities of the liquid phase of the chemicals in the mixture at its current temperature and pressure, in units of [m^2/s].

$$\nu = \frac{\mu}{\rho}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).nuls
[5.357870271650772e-07, 3.8127962283230277e-07]
```

property permittivites

Pure component relative permittivities of the chemicals in the mixture at its current temperature, [dimensionless].

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).permittivites
[2.23133472, 1.8508128]
```

phase = None

properties(*copy_pures=True*, *copy_mixtures=True*)

property_package_constants = None

property rho

Mass density of the mixture at its current phase and temperature and pressure, in units of [kg/m^3]. Available only if single phase.

Examples

```
>>> Mixture(['decane'], ws=[1], T=550, P=2E6).rho
498.67008448640604
```

property rhog

Gas-phase mass density of the mixture at its current temperature, pressure, and composition in units of [kg/m^3]. For calculation of this property at other temperatures, pressures, or compositions or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeGasMixture*; each Mixture instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Mixture(['hexane'], ws=[1], T=300, P=2E5).rhog
7.914447603999089
```

property rhog_STP

Gas-phase mass density of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [kg/m^3].

Examples

```
>>> Mixture(['nitrogen'], ws=[1]).rhog_STP
1.145534453639403
```

property rhogm

Molar density of the mixture in the gas phase at the current temperature, pressure, and composition in units of [mol/m^3].

Utilizes the object oriented interface and *thermo.volume.VolumeGasMixture* to perform the actual calculation of molar volume.

Examples

```
>>> Mixture(['water'], ws=[1], T=500).rhogm
24.467426039789093
```

property rhogm_STP

Molar density of the mixture in the gas phase at 298.15 K and 101.325 kPa, and the current composition, in units of [mol/m^3].

Examples

```
>>> Mixture(['nitrogen'], ws=[1]).rhogm_STP
40.892374850585895
```

property rhogms

Pure component molar densities of the chemicals in the gas phase at the current temperature and pressure, in units of [mol/m^3].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).rhogms
[40.11392035309789, 41.407547778608084]
```

property rhogs

Pure-component gas-phase mass densities of the chemicals in the mixture at its current temperature and pressure, in units of [kg/m^3].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).rhogs
[3.1333721283939258, 3.8152260283954584]
```

property rhol

Liquid-phase mass density of the mixture at its current temperature, pressure, and composition in units of [kg/m^3]. For calculation of this property at other temperatures, pressures, compositions or specifying manually the method used to calculate it, and more - see the object oriented interface *thermo.volume.VolumeLiquidMixture*; each Mixture instance creates one to actually perform the calculations. Note that that interface provides output in molar units.

Examples

```
>>> Mixture(['o-xylene'], ws=[1], T=297).rhol
876.9946785618097
```

property rhol_STP

Liquid-phase mass density of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [kg/m^3].

Examples

```
>>> Mixture(['cyclobutane'], ws=[1]).rhol_STP
688.9851989526821
```

property rholm

Molar density of the mixture in the liquid phase at the current temperature, pressure, and composition in units of [mol/m^3].

Utilizes the object oriented interface and *thermo.volume.VolumeLiquidMixture* to perform the actual calculation of molar volume.

Examples

```
>>> Mixture(['water'], ws=[1], T=300).rholm
55317.352773503124
```

property rholm_STP
Molar density of the mixture in the liquid phase at 298.15 K and 101.325 kPa, and the current composition, in units of [mol/m^3].

Examples

```
>>> Mixture(['water'], ws=[1]).rholm_STP
55344.59086372442
```

property rholms
Pure component molar densities of the chemicals in the mixture in the liquid phase at the current temperature and pressure, in units of [mol/m^3].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).rholms
[10882.699301520635, 9135.590853014008]
```

property rhols
Pure-component liquid-phase mass density of the chemicals in the mixture at its current temperature and pressure, in units of [kg/m^3].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).rhols
[850.0676666084917, 841.7389069631628]
```

property rhom
Molar density of the mixture at its current phase and temperature and pressure, in units of [mol/m^3]. Available only if single phase.

Examples

```
>>> Mixture(['1-hexanol'], ws=[1]).rhom
7983.414573003429
```

rhos = None

property rhosms
Pure component molar densities of the chemicals in the solid phase at the current temperature and pressure, in units of [mol/m^3].

Examples

```
>>> Mixture(['iron'], ws=[1], T=320).rhosms
[140925.7767033753]
```

property rhoss

Pure component solid-phase mass density of the chemicals in the mixture at its current temperature, in units of [kg/m^3].

Examples

```
>>> Mixture(['iron'], ws=[1], T=320).rhoss
[7869.999999999994]
```

property ringss

List of ring counts for all chemicals in the mixture.

Examples

```
>>> Mixture(['Docetaxel', 'Paclitaxel'], zs=[.5, .5]).ringss
[6, 7]
```

set_Chemical_property_objects()

set_TP_sources()

set_chemical_TP(*T=None*, *P=None*)

Basic method to change all chemical instances to be at the T and P specified. If they are not specified, the the values of the mixture will be used. This is not necessary for using the Mixture instance unless values specified to chemicals are required.

set_chemical_constants()

Basic method which retrieves and sets constants of chemicals to be accessible as lists from a Mixture object. This gets called automatically on the instantiation of a new Mixture instance.

set_constant_sources()

set_constants()

set_eos(*T*, *P*, *eos=<class 'thermo.eos_mix.PRMIX'>*)

set_property_package(*pkg=None*)

property sigma

Surface tension of the mixture at its current temperature and composition, in units of [N/m].

For calculation of this property at other temperatures, or specifying manually the method used to calculate it, and more - see the object oriented interface `thermo.interface.SurfaceTensionMixture`; each Mixture instance creates one to actually perform the calculations.

Examples

```
>>> Mixture(['water'], ws=[1], T=300, P=1E5).sigma
0.07176932405246211
```

property sigmas

Pure component surface tensions of the chemicals in the mixture at its current temperature, in units of [N/m].

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5], T=320).sigmas
[0.02533469712937521, 0.025254723406585546]
```

property similarity_variables

Similarity variables for all chemicals in the mixture, see `chemicals.elements.similarity_variable` for the definition, [mol/g]

Examples

```
>>> Mixture(['benzene', 'toluene'], ws=[0.5, 0.5]).similarity_variables
[0.15362587797189262, 0.16279853724428964]
```

property smiless

SMILES strings for all chemicals in the mixture.

Examples

```
>>> Mixture(['methane', 'ethane', 'propane', 'butane'],
... zs=[0.25, 0.25, 0.25, 0.25]).smiless
['C', 'CC', 'CCC', 'CCCC']
```

property solubility_parameters

Pure component solubility parameters of the chemicals in the mixture at its current temperature and pressure, in units of [Pa^0.5].

$$\delta = \sqrt{\frac{\Delta H_{vap} - RT}{V_m}}$$

Examples

```
>>> Mixture(['benzene', 'hexane'], ws=[0.5, 0.5], T=320).solubility_parameters
[18062.51359608708, 14244.12852702228]
```

property speed_of_sound

Bulk speed of sound of the mixture at its current temperature, [m/s].

Examples

```
>>> Mixture(['toluene'], P=1E5, VF=0.5, ws=[1]).speed_of_sound
478.99527258140211
```

property speed_of_sound_g
Gas-phase speed of sound of the mixture at its current temperature, [m/s].

Examples

```
>>> Mixture(['nitrogen'], ws=[1]).speed_of_sound_g
351.77445481641661
```

property speed_of_sound_l
Liquid-phase speed of sound of the mixture at its current temperature, [m/s].

Examples

```
>>> Mixture(['toluene'], P=1E5, T=300, ws=[1]).speed_of_sound_l
1116.0852487852942
```

property synonymss
Lists of synonyms for all chemicals in the mixture.

Examples

```
>>> Mixture(['Tetradecene', 'Pentadecene'], zs=[.1, .9]).synonymss
[['tetradec-2-ene', 'tetradecene', '2-tetradecene', 'tetradec-2-ene', '26952-13-
↪6', '35953-53-8', '1652-97-7'], ['pentadec-1-ene', '1-pentadecene',
↪'pentadecene,1-', 'pentadec-1-ene', '13360-61-7', 'pentadecene']]
```

xs = None

ys = None

7.21 Permittivity/Dielectric Constant (thermo.permittivity)

This module contains implementations of *TDependentProperty* representing liquid permittivity. A variety of estimation and data methods are available as included in the *chemicals* library.

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

7.21.1 Pure Liquid Permittivity

class thermo.permittivity.**PermittivityLiquid**(*CASRN=''*, *extrapolation='linear'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with liquid permittivity as a function of temperature. Consists of one temperature-dependent simple expression, one constant value source, and IAPWS.

Parameters

CASRN [str, optional] The CAS number of the chemical

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

Notes

To iterate over all methods, use the list stored in *permittivity_methods*.

CRC: Simple polynomials for calculating permittivity over a specified temperature range only. The full expression is:

$$\epsilon_r = A + BT + CT^2 + DT^3$$

Not all chemicals use all terms; in fact, few do. Data is available for 759 liquids, from [1].

CRC_CONSTANT: Constant permittivity values at specified temperatures only. Data is from [1], and is available for 1303 liquids.

IAPWS: The IAPWS model for water permittivity as a liquid.

References

[1]

Attributes

Tmax Maximum temperature (K) at which the current method can calculate the property.

Tmin Minimum temperature (K) at which the current method can calculate the property.

Methods

calculate(T, method)	Method to calculate permittivity of a liquid at temperature T with a given method.
test_method_validity(T, method)	Method to check the validity of a method.

property Tmax

Maximum temperature (K) at which the current method can calculate the property.

property Tmin

Minimum temperature (K) at which the current method can calculate the property.

calculate(*T*, *method*)

Method to calculate permittivity of a liquid at temperature T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate relative permittivity, [K]

method [str] Name of the method to use

Returns

epsilon [float] Relative permittivity of the liquid at T, [-]

name = 'liquid relative permittivity'

property_max = 1000.0

Maximum valid of permittivity; highest in the data available is ~240.

property_min = 1.0

Relative permittivity must always be larger than 1; nothing is better than a vacuum.

ranked_methods = ['IAPWS', 'CRC', 'CRC_CONSTANT']

Default rankings of the available methods.

test_method_validity(*T*, *method*)

Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = '-'

thermo.permittivity.**permittivity_methods = ['CRC', 'CRC_CONSTANT', 'IAPWS']**

Holds all methods available for the *PermittivityLiquid* class, for use in iterating over them.

7.22 Phase Models (thermo.phases)

The phases subpackage exposes classes that represent the state of single phase mixture, including the composition, temperature, pressure, enthalpy, and entropy. Phase objects are immutable and know nothing about bulk properties or transport properties. The goal is for each phase to be able to compute all of its thermodynamic properties, including volume-based ones. Use settings to handle different assumptions.

7.22.1 Base Class

class thermo.phases.**Phase**

Bases: `object`

Phase is the base class for all phase objects in *thermo*. Each sub-class implements a number of core properties; many other properties can be calculated from them.

Among those properties are *H*, *S*, *Cp*, *dP_dT*, *dP_dV*, *d2P_dT2*, *d2P_dV2*, and *d2P_dTdV*.

An additional set of properties that can be implemented and that enable more functionality are *dH_dP*, *dS_dT*, *dS_dP*, *d2H_dT2*, *d2H_dP2*, *d2S_dP2*, *dH_dT_V*, *dH_dP_V*, *dH_dV_T*, *dH_dV_P*, *dS_dT_V*, *dS_dP_V*, *d2H_dTdP*, *d2H_dT2_V*, *d2P_dTdP*, *d2P_dVdP*, *d2P_dVdT_TP*, *d2P_dT2_PV*.

Some models may re-implement properties which would normally be calculated by this *Phase* base class because they have more explicit, faster ways of calculating the property.

When a phase object is the result of a Flash calculation, the resulting phase objects have a reference to a `ChemicalConstantsPackage` object and all of its properties can be accessed from from the resulting phase objects as well.

A `ChemicalConstantsPackage` object can also be manually set to the attribute *constants* to enable access to those properties. This includes mass-based properties, which are not accessible from Phase objects without a reference to the constants.

Attributes

`CASs` CAS registration numbers for each component, [-].

`Carcinogens` Status of each component in cancer causing registries, [-].

`Ceilings` Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

`GWPs` Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

`Gfgs` Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

`Gfgs_mass` Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

`Hcs` Higher standard molar heats of combustion for each component, [J/mol].

`Hcs_lower` Lower standard molar heats of combustion for each component, [J/mol].

`Hcs_lower_mass` Lower standard heats of combustion for each component, [J/kg].

`Hcs_mass` Higher standard heats of combustion for each component, [J/kg].

`Hf_STPs` Standard state molar enthalpies of formation for each component, [J/mol].

`Hf_STPs_mass` Standard state mass enthalpies of formation for each component, [J/kg].

Hfgs Ideal gas standard molar enthalpies of formation for each component, [J/mol].

Hfgs_mass Ideal gas standard enthalpies of formation for each component, [J/kg].

Hfus_Tms Molar heats of fusion for each component at their respective melting points, [J/mol].

Hfus_Tms_mass Heats of fusion for each component at their respective melting points, [J/kg].

Hsub_Tts Heats of sublimation for each component at their respective triple points, [J/mol].

Hsub_Tts_mass Heats of sublimation for each component at their respective triple points, [J/kg].

Hvap_298s Molar heats of vaporization for each component at 298.15 K, [J/mol].

Hvap_298s_mass Heats of vaporization for each component at 298.15 K, [J/kg].

Hvap_Tbs Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

Hvap_Tbs_mass Heats of vaporization for each component at their respective normal boiling points, [J/kg].

InChI_Keys InChI Keys for each component, [-].

InChIs InChI strings for each component, [-].

LFLs Lower flammability limits for each component, [-].

MWs Similatiry variables for each component, [g/mol].

ODPs Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

PSRK_groups PSRK subgroup: count groups for each component, [-].

Parachors Parachors for each component, [N^0.25*m^2.75/mol].

Pcs Critical pressures for each component, [Pa].

Psat_298s Vapor pressures for each component at 298.15 K, [Pa].

Pts Triple point pressures for each component, [Pa].

PubChems Pubchem IDs for each component, [-].

RI_Ts Temperatures at which the refractive indexes were reported for each component, [K].

RIs Refractive indexes for each component, [-].

S0gs Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

S0gs_mass Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

STELs Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Sfgs Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

Sfgs_mass Ideal gas standard entropies of formation for each component, [J/(kg*K)].

Skins Whether each compound can be absorbed through the skin or not, [-].

StielPolars Stiel polar factors for each component, [-].

Stockmayers Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

TWAs Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Tautoignitions Autoignition temperatures for each component, [K].

Tbs Boiling temperatures for each component, [K].

Tcs Critical temperatures for each component, [K].

Tflashs Flash point temperatures for each component, [K].

Tms Melting temperatures for each component, [K].

Tts Triple point temperatures for each component, [K].

UFLs Upper flammability limits for each component, [-].

UNIFAC_Dortmund_groups UNIFAC_Dortmund_group: count groups for each component, [-].

UNIFAC_Qs UNIFAC Q parameters for each component, [-].

UNIFAC_Rs UNIFAC R parameters for each component, [-].

UNIFAC_groups UNIFAC_group: count groups for each component, [-].

VF Method to return the vapor fraction of the phase.

Van_der_Waals_areas Unnormalized Van der Waals areas for each component, [m^2/mol].

Van_der_Waals_volumes Unnormalized Van der Waals volumes for each component, [m^3/mol].

Vcs Critical molar volumes for each component, [m^3/mol].

Vmg_STPs Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

Vml_60Fs Liquid molar volumes for each component at 60 °F, [m^3/mol].

Vml_STPs Liquid molar volumes for each component at STP, [m^3/mol].

Vml_Tms Liquid molar volumes for each component at their respective melting points, [m^3/mol].

Vms_Tms Solid molar volumes for each component at their respective melting points, [m^3/mol].

Zcs Critical compressibilities for each component, [-].

atomss Breakdown of each component into its elements and their counts, as a dict, [-].

beta Method to return the phase fraction of this phase.

beta_mass Method to return the mass phase fraction of this phase.

beta_volume Method to return the volumetric phase fraction of this phase.

charges Charge number (valence) for each component, [-].

conductivities Electrical conductivities for each component, [S/m].

conductivity_Ts Temperatures at which the electrical conductivities for each component were measured, [K].

dipoles Dipole moments for each component, [debye].

economic_statuses Status of each component in in relation to import and export from various regions, [-].

force_phase

formulas Formulas of each component, [-].

legal_statuses Status of each component in in relation to import and export rules from various regions, [-].

logPs Octanol-water partition coefficients for each component, [-].

molecular_diameters Lennard-Jones molecular diameters for each component, [angstrom].

names Names for each component, [-].

omegas Acentric factors for each component, [-].

phase_STPs Standard states ('g', 'l', or 's') for each component, [-].

rhocs Molar densities at the critical point for each component, [mol/m^3].

rhocs_mass Densities at the critical point for each component, [kg/m^3].

rhog_STPs Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

rhog_STPs_mass Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

rhol_60Fs Liquid molar densities for each component at 60 °F, [mol/m^3].

rhol_60Fs_mass Liquid mass densities for each component at 60 °F, [kg/m^3].

rhol_STPs Molar liquid densities at STP for each component, [mol/m^3].

rhol_STPs_mass Liquid densities at STP for each component, [kg/m^3].

rhos_Tms Solid molar densities for each component at their respective melting points, [mol/m^3].

rhos_Tms_mass Solid mass densities for each component at their melting point, [kg/m^3].

sigma_STPs Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

sigma_Tbs Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

sigma_Tms Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

similarity_variables Similarity variables for each component, [mol/g].

smiless SMILES identifiers for each component, [-].

solubility_parameters Solubility parameters for each component at 298.15 K, [Pa^0.5].

Methods

A()	Method to calculate and return the Helmholtz energy of the phase.
API()	Method to calculate and return the API of the phase.
A_dep()	Method to calculate and return the departure Helmholtz energy of the phase.
A_formation_ideal_gas()	Method to calculate and return the ideal-gas Helmholtz energy of formation of the phase (as if the phase was an ideal gas).
A_ideal_gas()	Method to calculate and return the ideal-gas Helmholtz energy of the phase.

continues on next page

Table 80 – continued from previous page

A_mass()	Method to calculate and return mass Helmholtz energy of the phase.
A_reactive()	Method to calculate and return the Helmholtz free energy of the phase on a reactive basis.
Cp()	Method to calculate and return the constant-pressure heat capacity of the phase.
Cp_Cv_ratio()	Method to calculate and return the Cp/Cv ratio of the phase.
Cp_Cv_ratio_ideal_gas()	Method to calculate and return the ratio of the ideal-gas heat capacity to its constant-volume heat capacity.
Cp_ideal_gas()	Method to calculate and return the ideal-gas heat capacity of the phase.
Cp_mass()	Method to calculate and return mass constant pressure heat capacity of the phase.
Cpig_integrals_over_T_pure()	Method to calculate and return the integrals of the ideal-gas heat capacities divided by temperature of every component in the phase from a temperature of Phase.T_REF_IG to the system temperature.
Cpig_integrals_pure()	Method to calculate and return the integrals of the ideal-gas heat capacities of every component in the phase from a temperature of Phase.T_REF_IG to the system temperature.
Cpigs_pure()	Method to calculate and return the ideal-gas heat capacities of every component in the phase.
Cv()	Method to calculate and return the constant-volume heat capacity Cv of the phase.
Cv_dep()	Method to calculate and return the difference between the actual Cv and the ideal-gas constant volume heat capacity C_v^{ig} of the phase.
Cv_ideal_gas()	Method to calculate and return the ideal-gas constant volume heat capacity of the phase.
Cv_mass()	Method to calculate and return mass constant volume heat capacity of the phase.
G()	Method to calculate and return the Gibbs free energy of the phase.
G_dep()	Method to calculate and return the departure Gibbs free energy of the phase.
G_dep_phi_consistency()	Method to calculate and return a consistency check between departure Gibbs free energy, and the fugacity coefficients.
G_formation_ideal_gas()	Method to calculate and return the ideal-gas Gibbs free energy of formation of the phase (as if the phase was an ideal gas).
G_ideal_gas()	Method to calculate and return the ideal-gas Gibbs free energy of the phase.
G_mass()	Method to calculate and return mass Gibbs energy of the phase.
G_min()	Method to calculate and return the Gibbs free energy of the phase.

continues on next page

Table 80 – continued from previous page

G_min_criteria()	Method to calculate and return the Gibbs energy criteria required for comparing phase stability.
G_reactive()	Method to calculate and return the Gibbs free energy of the phase on a reactive basis.
H()	Method to calculate and return the enthalpy of the phase.
H_C_ratio()	Method to calculate and return the atomic ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.
H_C_ratio_mass()	Method to calculate and return the mass ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.
H_dep_phi_consistency()	Method to calculate and return a consistency check between departure enthalpy, and the fugacity coefficients' temperature derivatives.
H_formation_ideal_gas()	Method to calculate and return the ideal-gas enthalpy of formation of the phase (as if the phase was an ideal gas).
H_from_phi()	Method to calculate and return the enthalpy of the fluid as calculated from the ideal-gas enthalpy and the the fugacity coefficients' temperature derivatives.
H_ideal_gas()	Method to calculate and return the ideal-gas enthalpy of the phase.
H_mass()	Method to calculate and return mass enthalpy of the phase.
H_phi_consistency()	Method to calculate and return a consistency check between ideal gas enthalpy behavior, and the fugacity coefficients and their temperature derivatives.
H_reactive()	Method to calculate and return the enthalpy of the phase on a reactive basis, using the *Hfs* values of the phase.
Hc()	Method to calculate and return the molar ideal-gas higher heat of combustion of the object, [J/mol]
Hc_lower()	Method to calculate and return the molar ideal-gas lower heat of combustion of the object, [J/mol]
Hc_lower_mass()	Method to calculate and return the mass ideal-gas lower heat of combustion of the object, [J/mol]
Hc_lower_normal()	Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the normal gas volume, [J/m^3]
Hc_lower_standard()	Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the standard gas volume, [J/m^3]
Hc_mass()	Method to calculate and return the mass ideal-gas higher heat of combustion of the object, [J/mol]
Hc_normal()	Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the normal gas volume, [J/m^3]
Hc_standard()	Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the standard gas volume, [J/m^3]

continues on next page

Table 80 – continued from previous page

Joule_Thomson()	Method to calculate and return the Joule-Thomson coefficient of the phase.
MW()	Method to calculate and return molecular weight of the phase.
MW_inv()	Method to calculate and return inverse of molecular weight of the phase.
PIP()	Method to calculate and return the phase identification parameter of the phase.
P_max_at_V(V)	Dummy method.
P_transitions()	Dummy method.
Pmc()	Method to calculate and return the mechanical critical pressure of the phase.
S()	Method to calculate and return the entropy of the phase.
SG()	Method to calculate and return the standard liquid specific gravity of the phase, using constant liquid pure component densities not calculated by the phase object, at 60 °F.
SG_gas()	Method to calculate and return the specific gravity of the phase with respect to a gas reference density.
S_dep_phi_consistency()	Method to calculate and return a consistency check between ideal gas entropy behavior, and the fugacity coefficients and their temperature derivatives.
S_formation_ideal_gas()	Method to calculate and return the ideal-gas entropy of formation of the phase (as if the phase was an ideal gas).
S_from_phi()	Method to calculate and return the entropy of the fluid as calculated from the ideal-gas entropy and the the fugacity coefficients' temperature derivatives.
S_ideal_gas()	Method to calculate and return the ideal-gas entropy of the phase.
S_mass()	Method to calculate and return mass entropy of the phase.
S_phi_consistency()	Method to calculate and return a consistency check between ideal gas entropy behavior, and the fugacity coefficients and their temperature derivatives.
S_reactive()	Method to calculate and return the entropy of the phase on a reactive basis, using the *Sfs* values of the phase.
T_max_at_V(V)	Method to calculate the maximum temperature the phase can create at a constant volume, if one exists; returns None otherwise.
Tmc()	Method to calculate and return the mechanical critical temperature of the phase.
U()	Method to calculate and return the internal energy of the phase.
U_dep()	Method to calculate and return the departure internal energy of the phase.
U_formation_ideal_gas()	Method to calculate and return the ideal-gas internal energy of formation of the phase (as if the phase was an ideal gas).

continues on next page

Table 80 – continued from previous page

U_ideal_gas()	Method to calculate and return the ideal-gas internal energy of the phase.
U_mass()	Method to calculate and return mass internal energy of the phase.
U_reactive()	Method to calculate and return the internal energy of the phase on a reactive basis.
V()	Method to return the molar volume of the phase.
V_dep()	Method to calculate and return the departure (from ideal gas behavior) molar volume of the phase.
V_from_phi()	Method to calculate and return the molar volume of the fluid as calculated from the pressure derivatives of fugacity coefficients.
V_gas()	Method to calculate and return the ideal-gas molar volume of the phase at the chosen reference temperature and pressure, according to the temperature variable *T_gas_ref* and pressure variable *P_gas_ref* of the thermo.bulk.BulkSettings.
V_gas_normal()	Method to calculate and return the ideal-gas molar volume of the phase at the normal temperature and pressure, according to the temperature variable *T_normal* and pressure variable *P_normal* of the thermo.bulk.BulkSettings.
V_gas_standard()	Method to calculate and return the ideal-gas molar volume of the phase at the standard temperature and pressure, according to the temperature variable *T_standard* and pressure variable *P_standard* of the thermo.bulk.BulkSettings.
V_ideal_gas()	Method to calculate and return the ideal-gas molar volume of the phase.
V_iter([force])	Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure.
V_liquid_ref()	Method to calculate and return the liquid reference molar volume according to the temperature variable *T_liquid_volume_ref* of thermo.bulk.BulkSettings and the composition of the phase.
V_mass()	Method to calculate and return the specific volume of the phase.
V_phi_consistency()	Method to calculate and return a consistency check between molar volume, and the fugacity coefficients' pressures derivatives.
Vfgs()	Method to calculate and return the ideal-gas volume fractions of the components of the phase.
Vfls()	Method to calculate and return the ideal-liquid volume fractions of the components of the phase, using the standard liquid densities at the temperature variable *T_liquid_volume_ref* of thermo.bulk.BulkSettings and the composition of the phase.
Vmc()	Method to calculate and return the mechanical critical volume of the phase.

continues on next page

Table 80 – continued from previous page

Wobbe_index()	Method to calculate and return the molar Wobbe index of the object, [J/mol].
Wobbe_index_lower()	Method to calculate and return the molar lower Wobbe index of the
Wobbe_index_lower_mass()	Method to calculate and return the lower mass Wobbe index of the object, [J/kg].
Wobbe_index_lower_normal()	Method to calculate and return the volumetric normal lower Wobbe index of the object, [J/m^3].
Wobbe_index_lower_standard()	Method to calculate and return the volumetric standard lower Wobbe index of the object, [J/m^3].
Wobbe_index_mass()	Method to calculate and return the mass Wobbe index of the object, [J/kg].
Wobbe_index_normal()	Method to calculate and return the volumetric normal Wobbe index of the object, [J/m^3].
Wobbe_index_standard()	Method to calculate and return the volumetric standard Wobbe index of the object, [J/m^3].
Z()	Method to calculate and return the compressibility factor of the phase.
Zmc()	Method to calculate and return the mechanical critical compressibility of the phase.
activities()	Method to calculate and return the activities of each component in the phase [-].
as_json()	Method to create a JSON-friendly serialization of the phase which can be stored, and reloaded later.
atom_fractions()	Method to calculate and return the atomic composition of the phase; returns a dictionary of atom fraction (by count), containing only those elements who are present.
atom_mass_fractions()	Method to calculate and return the atomic mass fractions of the phase; returns a dictionary of atom fraction (by mass), containing only those elements who are present.
chemical_potential()	Method to calculate and return the chemical potentials of each component in the phase [-].
d2P_dT2()	Method to calculate and return the second temperature derivative of pressure of the phase.
d2P_dTdV()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.
d2P_dTdrho()	Method to calculate and return the temperature derivative and then molar density derivative of the pressure of the phase.
d2P_dV2()	Method to calculate and return the second volume derivative of pressure of the phase.
d2P_dVdT()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.
d2P_drho2()	Method to calculate and return the second molar density derivative of pressure of the phase.

continues on next page

Table 80 – continued from previous page

d2T_dP2()	Method to calculate and return the constant-volume second pressure derivative of temperature of the phase.
d2T_dP2_V()	Method to calculate and return the constant-volume second pressure derivative of temperature of the phase.
d2T_dPdV()	Method to calculate and return the derivative of pressure and then the derivative of volume of temperature of the phase.
d2T_dPdrho()	Method to calculate and return the pressure derivative and then molar density derivative of the temperature of the phase.
d2T_dV2()	Method to calculate and return the constant-pressure second volume derivative of temperature of the phase.
d2T_dV2_P()	Method to calculate and return the constant-pressure second volume derivative of temperature of the phase.
d2T_dVdP()	Method to calculate and return the derivative of pressure and then the derivative of volume of temperature of the phase.
d2T_drho2()	Method to calculate and return the second molar density derivative of temperature of the phase.
d2V_dP2()	Method to calculate and return the constant-temperature pressure derivative of volume of the phase.
d2V_dP2_T()	Method to calculate and return the constant-temperature pressure derivative of volume of the phase.
d2V_dPdT()	Method to calculate and return the derivative of pressure and then the derivative of temperature of volume of the phase.
d2V_dT2()	Method to calculate and return the constant-pressure second temperature derivative of volume of the phase.
d2V_dT2_P()	Method to calculate and return the constant-pressure second temperature derivative of volume of the phase.
d2V_dTdP()	Method to calculate and return the derivative of pressure and then the derivative of temperature of volume of the phase.
d2rho_dP2()	Method to calculate and return the second pressure derivative of molar density of the phase.
d2rho_dPdT()	Method to calculate and return the pressure derivative and then temperature derivative of the molar density of the phase.
d2rho_dT2()	Method to calculate and return the second temperature derivative of molar density of the phase.
dA_dP()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.

continues on next page

Table 80 – continued from previous page

dA_dP_T()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.
dA_dP_V()	Method to calculate and return the constant-volume pressure derivative of Helmholtz energy.
dA_dT()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dA_dT_P()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dA_dT_V()	Method to calculate and return the constant-volume temperature derivative of Helmholtz energy.
dA_dV_P()	Method to calculate and return the constant-pressure volume derivative of Helmholtz energy.
dA_dV_T()	Method to calculate and return the constant-temperature volume derivative of Helmholtz energy.
dA_mass_dP([prop])	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.
dA_mass_dP_T([prop])	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.
dA_mass_dP_V([prop])	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant volume.
dA_mass_dT([prop])	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dT_P([prop])	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dT_V([prop])	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant volume.
dA_mass_dV_P([prop])	Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dV_T([prop])	Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant temperature.
dCpigs_dT_pure()	Method to calculate and return the first temperature derivative of ideal-gas heat capacities of every component in the phase.
dCv_dP_T()	Method to calculate the pressure derivative of Cv, constant volume heat capacity, at constant temperature.
dCv_dT_P()	Method to calculate the temperature derivative of Cv, constant volume heat capacity, at constant pressure.
dCv_mass_dP_T([prop])	Method to calculate and return the pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature.

continues on next page

Table 80 – continued from previous page

dCv_mass_dT_P([prop])	Method to calculate and return the temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure.
dG_dP()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dP_T()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dP_V()	Method to calculate and return the constant-volume pressure derivative of Gibbs free energy.
dG_dT()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.
dG_dT_P()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.
dG_dT_V()	Method to calculate and return the constant-volume temperature derivative of Gibbs free energy.
dG_dV_P()	Method to calculate and return the constant-pressure volume derivative of Gibbs free energy.
dG_dV_T()	Method to calculate and return the constant-temperature volume derivative of Gibbs free energy.
dG_mass_dP([prop])	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.
dG_mass_dP_T([prop])	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.
dG_mass_dP_V([prop])	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant volume.
dG_mass_dT([prop])	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dT_P([prop])	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dT_V([prop])	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant volume.
dG_mass_dV_P([prop])	Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dV_T([prop])	Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant temperature.
dH_dP_T()	Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.
dH_dT_P()	Method to calculate and return the temperature derivative of enthalpy of the phase at constant pressure.

continues on next page

Table 80 – continued from previous page

dH_dns()	Method to calculate and return the mole number derivative of the enthalpy of the phase.
dH_mass_dP([prop])	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.
dH_mass_dP_T([prop])	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.
dH_mass_dP_V([prop])	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant volume.
dH_mass_dT([prop])	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dT_P([prop])	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dT_V([prop])	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant volume.
dH_mass_dV_P([prop])	Method to calculate and return the volume derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dV_T([prop])	Method to calculate and return the volume derivative of mass enthalpy of the phase at constant temperature.
dP_dP_A([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of pressure of the phase at constant Helmholtz energy.
dP_dP_G([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of pressure of the phase at constant Gibbs energy.
dP_dP_H([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of pressure of the phase at constant enthalpy.
dP_dP_S([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of pressure of the phase at constant entropy.
dP_dP_T()	Method to calculate and return the pressure derivative of pressure of the phase at constant temperature.
dP_dP_U([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of pressure of the phase at constant internal energy.
dP_dP_V()	Method to calculate and return the pressure derivative of pressure of the phase at constant volume.
dP_dT()	Method to calculate and return the first temperature derivative of pressure of the phase.
dP_dT_A([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of pressure of the phase at constant Helmholtz energy.
dP_dT_G([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of pressure of the phase at constant Gibbs energy.
dP_dT_H([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of pressure of the phase at constant enthalpy.

continues on next page

Table 80 – continued from previous page

dP_dT_P()	Method to calculate and return the temperature derivative of temperature of the phase at constant pressure.
dP_dT_S([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of pressure of the phase at constant entropy.
dP_dT_U([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of pressure of the phase at constant internal energy.
dP_dV()	Method to calculate and return the first volume derivative of pressure of the phase.
dP_dV_A([property, differentiate_by, ...])	Method to calculate and return the volume derivative of pressure of the phase at constant Helmholtz energy.
dP_dV_G([property, differentiate_by, ...])	Method to calculate and return the volume derivative of pressure of the phase at constant Gibbs energy.
dP_dV_H([property, differentiate_by, ...])	Method to calculate and return the volume derivative of pressure of the phase at constant enthalpy.
dP_dV_P()	Method to calculate and return the volume derivative of pressure of the phase at constant pressure.
dP_dV_S([property, differentiate_by, ...])	Method to calculate and return the volume derivative of pressure of the phase at constant entropy.
dP_dV_U([property, differentiate_by, ...])	Method to calculate and return the volume derivative of pressure of the phase at constant internal energy.
dP_drho()	Method to calculate and return the molar density derivative of pressure of the phase.
dP_drho_A([property, differentiate_by, ...])	Method to calculate and return the density derivative of pressure of the phase at constant Helmholtz energy.
dP_drho_G([property, differentiate_by, ...])	Method to calculate and return the density derivative of pressure of the phase at constant Gibbs energy.
dP_drho_H([property, differentiate_by, ...])	Method to calculate and return the density derivative of pressure of the phase at constant enthalpy.
dP_drho_S([property, differentiate_by, ...])	Method to calculate and return the density derivative of pressure of the phase at constant entropy.
dP_drho_U([property, differentiate_by, ...])	Method to calculate and return the density derivative of pressure of the phase at constant internal energy.
dS_dP_T()	Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.
dS_dV_P()	Method to calculate and return the volume derivative of entropy of the phase at constant pressure.
dS_dV_T()	Method to calculate and return the volume derivative of entropy of the phase at constant temperature.
dS_dns()	Method to calculate and return the mole number derivative of the entropy of the phase.
dS_mass_dP([prop])	Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.
dS_mass_dP_T([prop])	Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.
dS_mass_dP_V([prop])	Method to calculate and return the pressure derivative of mass entropy of the phase at constant volume.

continues on next page

Table 80 – continued from previous page

dS_mass_dT([prop])	Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.
dS_mass_dT_P([prop])	Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.
dS_mass_dT_V([prop])	Method to calculate and return the temperature derivative of mass entropy of the phase at constant volume.
dS_mass_dV_P([prop])	Method to calculate and return the volume derivative of mass entropy of the phase at constant pressure.
dS_mass_dV_T([prop])	Method to calculate and return the volume derivative of mass entropy of the phase at constant temperature.
dT_dP()	Method to calculate and return the constant-volume pressure derivative of temperature of the phase.
dT_dP_A([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of temperature of the phase at constant Helmholtz energy.
dT_dP_G([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of temperature of the phase at constant Gibbs energy.
dT_dP_H([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of temperature of the phase at constant enthalpy.
dT_dP_S([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of temperature of the phase at constant entropy.
dT_dP_T()	Method to calculate and return the pressure derivative of temperature of the phase at constant temperature.
dT_dP_U([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of temperature of the phase at constant internal energy.
dT_dP_V()	Method to calculate and return the constant-volume pressure derivative of temperature of the phase.
dT_dT_A([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of temperature of the phase at constant Helmholtz energy.
dT_dT_G([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of temperature of the phase at constant Gibbs energy.
dT_dT_H([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of temperature of the phase at constant enthalpy.
dT_dT_P()	Method to calculate and return the temperature derivative of temperature of the phase at constant pressure.
dT_dT_S([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of temperature of the phase at constant entropy.
dT_dT_U([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of temperature of the phase at constant internal energy.

continues on next page

Table 80 – continued from previous page

dT_dT_V()	Method to calculate and return the temperature derivative of temperature of the phase at constant volume.
dT_dV()	Method to calculate and return the constant-pressure volume derivative of temperature of the phase.
dT_dV_A([property, differentiate_by, ...])	Method to calculate and return the volume derivative of temperature of the phase at constant Helmholtz energy.
dT_dV_G([property, differentiate_by, ...])	Method to calculate and return the volume derivative of temperature of the phase at constant Gibbs energy.
dT_dV_H([property, differentiate_by, ...])	Method to calculate and return the volume derivative of temperature of the phase at constant enthalpy.
dT_dV_P()	Method to calculate and return the constant-pressure volume derivative of temperature of the phase.
dT_dV_S([property, differentiate_by, ...])	Method to calculate and return the volume derivative of temperature of the phase at constant entropy.
dT_dV_T()	Method to calculate and return the volume derivative of temperature of the phase at constant temperature.
dT_dV_U([property, differentiate_by, ...])	Method to calculate and return the volume derivative of temperature of the phase at constant internal energy.
dT_drho()	Method to calculate and return the molar density derivative of temperature of the phase.
dT_drho_A([property, differentiate_by, ...])	Method to calculate and return the density derivative of temperature of the phase at constant Helmholtz energy.
dT_drho_G([property, differentiate_by, ...])	Method to calculate and return the density derivative of temperature of the phase at constant Gibbs energy.
dT_drho_H([property, differentiate_by, ...])	Method to calculate and return the density derivative of temperature of the phase at constant enthalpy.
dT_drho_S([property, differentiate_by, ...])	Method to calculate and return the density derivative of temperature of the phase at constant entropy.
dT_drho_U([property, differentiate_by, ...])	Method to calculate and return the density derivative of temperature of the phase at constant internal energy.
dU_dP()	Method to calculate and return the constant-temperature pressure derivative of internal energy.
dU_dP_T()	Method to calculate and return the constant-temperature pressure derivative of internal energy.
dU_dP_V()	Method to calculate and return the constant-volume pressure derivative of internal energy.
dU_dT()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
dU_dT_P()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
dU_dT_V()	Method to calculate and return the constant-volume temperature derivative of internal energy.
dU_dV_P()	Method to calculate and return the constant-pressure volume derivative of internal energy.
dU_dV_T()	Method to calculate and return the constant-temperature volume derivative of internal energy.

continues on next page

Table 80 – continued from previous page

dU_mass_dP([prop])	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.
dU_mass_dP_T([prop])	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.
dU_mass_dP_V([prop])	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant volume.
dU_mass_dT([prop])	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.
dU_mass_dT_P([prop])	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.
dU_mass_dT_V([prop])	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant volume.
dU_mass_dV_P([prop])	Method to calculate and return the volume derivative of mass internal energy of the phase at constant pressure.
dU_mass_dV_T([prop])	Method to calculate and return the volume derivative of mass internal energy of the phase at constant temperature.
dV_dP()	Method to calculate and return the constant-temperature pressure derivative of volume of the phase.
dV_dP_A([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of volume of the phase at constant Helmholtz energy.
dV_dP_G([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of volume of the phase at constant Gibbs energy.
dV_dP_H([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of volume of the phase at constant enthalpy.
dV_dP_S([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of volume of the phase at constant entropy.
dV_dP_T()	Method to calculate and return the constant-temperature pressure derivative of volume of the phase.
dV_dP_U([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of volume of the phase at constant internal energy.
dV_dP_V()	Method to calculate and return the volume derivative of pressure of the phase at constant volume.
dV_dT()	Method to calculate and return the constant-pressure temperature derivative of volume of the phase.
dV_dT_A([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of volume of the phase at constant Helmholtz energy.
dV_dT_G([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of volume of the phase at constant Gibbs energy.

continues on next page

Table 80 – continued from previous page

dV_dT_H([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of volume of the phase at constant enthalpy.
dV_dT_P()	Method to calculate and return the constant-pressure temperature derivative of volume of the phase.
dV_dT_S([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of volume of the phase at constant entropy.
dV_dT_U([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of volume of the phase at constant internal energy.
dV_dT_V()	Method to calculate and return the temperature derivative of volume of the phase at constant volume.
dV_dV_A([property, differentiate_by, ...])	Method to calculate and return the volume derivative of volume of the phase at constant Helmholtz energy.
dV_dV_G([property, differentiate_by, ...])	Method to calculate and return the volume derivative of volume of the phase at constant Gibbs energy.
dV_dV_H([property, differentiate_by, ...])	Method to calculate and return the volume derivative of volume of the phase at constant enthalpy.
dV_dV_P()	Method to calculate and return the volume derivative of volume of the phase at constant pressure.
dV_dV_S([property, differentiate_by, ...])	Method to calculate and return the volume derivative of volume of the phase at constant entropy.
dV_dV_T()	Method to calculate and return the volume derivative of volume of the phase at constant temperature.
dV_dV_U([property, differentiate_by, ...])	Method to calculate and return the volume derivative of volume of the phase at constant internal energy.
dV_dns()	Method to calculate and return the mole number derivatives of the molar volume V of the phase.
dV_drho_A([property, differentiate_by, ...])	Method to calculate and return the density derivative of volume of the phase at constant Helmholtz energy.
dV_drho_G([property, differentiate_by, ...])	Method to calculate and return the density derivative of volume of the phase at constant Gibbs energy.
dV_drho_H([property, differentiate_by, ...])	Method to calculate and return the density derivative of volume of the phase at constant enthalpy.
dV_drho_S([property, differentiate_by, ...])	Method to calculate and return the density derivative of volume of the phase at constant entropy.
dV_drho_U([property, differentiate_by, ...])	Method to calculate and return the density derivative of volume of the phase at constant internal energy.
dZ_dP()	Method to calculate and return the pressure derivative of compressibility of the phase.
dZ_dT()	Method to calculate and return the temperature derivative of compressibility of the phase.
dZ_dV()	Method to calculate and return the volume derivative of compressibility of the phase.
dZ_dns()	Method to calculate and return the mole number derivatives of the compressibility factor Z of the phase.
dZ_dzs()	Method to calculate and return the mole fraction derivatives of the compressibility factor Z of the phase.

continues on next page

Table 80 – continued from previous page

dfugacities_dP()	Method to calculate and return the pressure derivative of the fugacities of the components in the phase.
dfugacities_dT()	Method to calculate and return the temperature derivative of fugacities of the phase.
dfugacities_dns()	Method to calculate and return the mole number derivative of the fugacities of the components in the phase.
dfugacity_dP()	Method to calculate and return the pressure derivative of fugacity of the phase; provided the phase is 1 component.
dfugacity_dT()	Method to calculate and return the temperature derivative of fugacity of the phase; provided the phase is 1 component.
disobaric_expansion_dP()	Method to calculate and return the pressure derivative of isobatic expansion coefficient of the phase.
disobaric_expansion_dT()	Method to calculate and return the temperature derivative of isobatic expansion coefficient of the phase.
disothermal_compressibility_dT()	Method to calculate and return the temperature derivative of isothermal compressibility of the phase.
dkappa_dT()	Method to calculate and return the temperature derivative of isothermal compressibility of the phase.
dlnfugacities_dns()	Method to calculate and return the mole number derivative of the log of fugacities of the components in the phase.
dlnfugacities_dzs()	Method to calculate and return the mole fraction derivative of the log of fugacities of the components in the phase.
dlnphis_dP()	Method to calculate and return the pressure derivative of the log of fugacity coefficients of each component in the phase.
dlnphis_dT()	Method to calculate and return the temperature derivative of the log of fugacity coefficients of each component in the phase.
dphis_dP()	Method to calculate and return the pressure derivative of fugacity coefficients of the phase.
dphis_dT()	Method to calculate and return the temperature derivative of fugacity coefficients of the phase.
dphis_dzs()	Method to calculate and return the molar composition derivative of fugacity coefficients of the phase.
drho_dP()	Method to calculate and return the pressure derivative of molar density of the phase.
drho_dP_A([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of density of the phase at constant Helmholtz energy.
drho_dP_G([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of density of the phase at constant Gibbs energy.
drho_dP_H([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of density of the phase at constant enthalpy.
drho_dP_S([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of density of the phase at constant entropy.

continues on next page

Table 80 – continued from previous page

drho_dP_U([property, differentiate_by, ...])	Method to calculate and return the pressure derivative of density of the phase at constant internal energy.
drho_dT()	Method to calculate and return the temperature derivative of molar density of the phase.
drho_dT_A([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of density of the phase at constant Helmholtz energy.
drho_dT_G([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of density of the phase at constant Gibbs energy.
drho_dT_H([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of density of the phase at constant enthalpy.
drho_dT_S([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of density of the phase at constant entropy.
drho_dT_U([property, differentiate_by, ...])	Method to calculate and return the temperature derivative of density of the phase at constant internal energy.
drho_dT_V()	Method to calculate and return the temperature derivative of molar density of the phase at constant volume.
drho_dV_A([property, differentiate_by, ...])	Method to calculate and return the volume derivative of density of the phase at constant Helmholtz energy.
drho_dV_G([property, differentiate_by, ...])	Method to calculate and return the volume derivative of density of the phase at constant Gibbs energy.
drho_dV_H([property, differentiate_by, ...])	Method to calculate and return the volume derivative of density of the phase at constant enthalpy.
drho_dV_S([property, differentiate_by, ...])	Method to calculate and return the volume derivative of density of the phase at constant entropy.
drho_dV_T()	Method to calculate and return the volume derivative of molar density of the phase.
drho_dV_U([property, differentiate_by, ...])	Method to calculate and return the volume derivative of density of the phase at constant internal energy.
drho_drho_A([property, differentiate_by, ...])	Method to calculate and return the density derivative of density of the phase at constant Helmholtz energy.
drho_drho_G([property, differentiate_by, ...])	Method to calculate and return the density derivative of density of the phase at constant Gibbs energy.
drho_drho_H([property, differentiate_by, ...])	Method to calculate and return the density derivative of density of the phase at constant enthalpy.
drho_drho_S([property, differentiate_by, ...])	Method to calculate and return the density derivative of density of the phase at constant entropy.
drho_drho_U([property, differentiate_by, ...])	Method to calculate and return the density derivative of density of the phase at constant internal energy.
drho_mass_dP()	Method to calculate the mass density derivative with respect to pressure, at constant temperature.
drho_mass_dT()	Method to calculate the mass density derivative with respect to temperature, at constant pressure.
dspeed_of_sound_dP_T()	Method to calculate the pressure derivative of speed of sound at constant temperature in molar units.
dspeed_of_sound_dT_P()	Method to calculate the temperature derivative of speed of sound at constant pressure in molar units.

continues on next page

Table 80 – continued from previous page

from_json(json_repr)	Method to create a phase from a JSON serialization of another phase.
fugacities()	Method to calculate and return the fugacities of the phase.
fugacities_at_zs(zs)	Method to directly calculate the figacities at a different composition than the current phase.
fugacities_lowest_Gibbs()	Method to calculate and return the fugacities of the phase.
fugacity()	Method to calculate and return the fugacity of the phase; provided the phase is 1 component.
gammas()	Method to calculate and return the activity coefficients of the phase, [-].
isentropic_exponent()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.
isentropic_exponent_PT()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $P^{(1-k)}T^k = \text{const}$.
isentropic_exponent_PV()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.
isentropic_exponent_TV()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $TV^{k-1} = \text{const}$.
isobaric_expansion()	Method to calculate and return the isobatic expansion coefficient of the phase.
isothermal_bulk_modulus()	Method to calculate and return the isothermal bulk modulus of the phase.
isothermal_compressibility()	Method to calculate and return the isothermal compressibility of the phase.
kappa()	Method to calculate and return the isothermal compressibility of the phase.
lnfugacities()	Method to calculate and return the log of fugacities of the phase.
lnphi()	Method to calculate and return the log of fugacity coefficient of the phase; provided the phase is 1 component.
lnphis()	Method to calculate and return the log of fugacity coefficients of each component in the phase.
lnphis_G_min()	Method to calculate and return the log fugacity coefficients of the phase.
lnphis_at_zs(zs)	Method to directly calculate the log fugacity coefficients at a different composition than the current phase.
log_zs()	Method to calculate and return the log of mole fractions specified.
model_hash([ignore_phase])	Method to compute a hash of a phase.
molar_water_content()	Method to calculate and return the molar water content; this is the g/mol of the fluid which is coming from water, [g/mol].

continues on next page

Table 80 – continued from previous page

mu()	
phi()	Method to calculate and return the fugacity coefficient of the phase; provided the phase is 1 component.
phis()	Method to calculate and return the fugacity coefficients of the phase.
pseudo_Pc()	Method to calculate and return the pseudocritical pressure calculated using Kay's rule (linear mole fractions):
pseudo_Tc()	Method to calculate and return the pseudocritical temperature calculated using Kay's rule (linear mole fractions):
pseudo_Vc()	Method to calculate and return the pseudocritical volume calculated using Kay's rule (linear mole fractions):
pseudo_Zc()	Method to calculate and return the pseudocritical compressibility calculated using Kay's rule (linear mole fractions):
rho()	Method to calculate and return the molar density of the phase.
rho_mass()	Method to calculate and return mass density of the phase.
rho_mass_liquid_ref()	Method to calculate and return the liquid reference mass density according to the temperature variable *T_liquid_volume_ref* of thermo.bulk.BulkSettings and the composition of the phase.
sigma()	Calculate and return the surface tension of the phase.
speed_of_sound()	Method to calculate and return the molar speed of sound of the phase.
speed_of_sound_mass()	Method to calculate and return the speed of sound of the phase.
state_hash()	Basic method to calculate a hash of the state of the phase and its model parameters.
to(zs[, T, P, V])	Method to create a new Phase object with the same constants as the existing Phase but at different conditions.
to_TP_zs(T, P, zs)	Method to create a new Phase object with the same constants as the existing Phase but at a different T and P.
value(name)	Method to retrieve a property from a string.
ws()	Method to calculate and return the mass fractions of the phase, [-]
ws_no_water()	Method to calculate and return the mass fractions of all species in the phase, normalized to a water-free basis (the mass fraction of water returned is zero).
zs_no_water()	Method to calculate and return the mole fractions of all species in the phase, normalized to a water-free basis (the mole fraction of water returned is zero).

A()
Method to calculate and return the Helmholtz energy of the phase.

$$A = U - TS$$

Returns

A [float] Helmholtz energy, [J/mol]

API()
Method to calculate and return the API of the phase.

$$\text{API gravity} = \frac{141.5}{\text{SG}} - 131.5$$

Returns

API [float] API of the fluid [-]

A_dep()
Method to calculate and return the departure Helmholtz energy of the phase.

$$A_{dep} = U_{dep} - TS_{dep}$$

Returns

A_dep [float] Departure Helmholtz energy, [J/mol]

A_formation_ideal_gas()
Method to calculate and return the ideal-gas Helmholtz energy of formation of the phase (as if the phase was an ideal gas).

$$A_{reactive}^{ig} = U_{reactive}^{ig} - T_{ref}^{ig} S_{reactive}^{ig}$$

Returns

A_formation_ideal_gas [float] Helmholtz energy of formation of the phase on a reactive basis as an ideal gas, [J/(mol)]

A_ideal_gas()
Method to calculate and return the ideal-gas Helmholtz energy of the phase.

$$A^{ig} = U^{ig} - TS^{ig}$$

Returns

A_ideal_gas [float] Ideal gas Helmholtz free energy, [J/(mol)]

A_mass()
Method to calculate and return mass Helmholtz energy of the phase.

$$A_{mass} = \frac{1000 A_{molar}}{MW}$$

Returns

A_mass [float] Mass Helmholtz energy, [J/(kg)]

A_reactive()
Method to calculate and return the Helmholtz free energy of the phase on a reactive basis.

$$A_{reactive} = U_{reactive} - TS_{reactive}$$

Returns

A_reactive [float] Helmholtz free energy of the phase on a reactive basis, [J/(mol)]

`property CASs`

CAS registration numbers for each component, [-].

Returns

CASs [list[str]] CAS registration numbers for each component, [-].

`property Carcinogens`

Status of each component in cancer causing registries, [-].

Returns

Carcinogens [list[dict]] Status of each component in cancer causing registries, [-].

`property Ceilings`

Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Returns

Ceilings [list[tuple[(float, str)]]] Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

`Cp()`

Method to calculate and return the constant-pressure heat capacity of the phase.

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

`Cp_Cv_ratio()`

Method to calculate and return the Cp/Cv ratio of the phase.

$$\frac{C_p}{C_v}$$

Returns

Cp_Cv_ratio [float] Cp/Cv ratio, [-]

`Cp_Cv_ratio_ideal_gas()`

Method to calculate and return the ratio of the ideal-gas heat capacity to its constant-volume heat capacity.

$$\frac{C_p^{ig}}{C_v^{ig}}$$

Returns

Cp_Cv_ratio_ideal_gas [float] Cp/Cv for the phase as an ideal gas, [-]

`Cp_ideal_gas()`

Method to calculate and return the ideal-gas heat capacity of the phase.

$$C_p^{ig} = \sum_i z_i C_{p,i}^{ig}$$

Returns

Cp [float] Ideal gas heat capacity, [J/(mol*K)]

`Cp_mass()`

Method to calculate and return mass constant pressure heat capacity of the phase.

$$Cp_{mass} = \frac{1000 Cp_{molar}}{MW}$$

Returns

Cp_mass [float] Mass heat capacity, [J/(kg*K)]

`Cpgs_poly_fit = False`

`Cpig_integrals_over_T_pure()`

Method to calculate and return the integrals of the ideal-gas heat capacities divided by temperature of every component in the phase from a temperature of *Phase.T_REF_IG* to the system temperature. This method is powered by the *HeatCapacityGases* objects, except when all components have the same heat capacity form and a fast implementation has been written for it (currently only polynomials).

$$\Delta S^{ig} = \int_{T_{ref}}^{T} \frac{C_p^{ig}}{T} dT$$

Returns

dS_ig [list[float]] Integrals of ideal gas heat capacity over temperature from the reference temperature to the system temperature, [J/(mol)]

`Cpig_integrals_pure()`

Method to calculate and return the integrals of the ideal-gas heat capacities of every component in the phase from a temperature of *Phase.T_REF_IG* to the system temperature. This method is powered by the *HeatCapacityGases* objects, except when all components have the same heat capacity form and a fast implementation has been written for it (currently only polynomials).

$$\Delta H^{ig} = \int_{T_{ref}}^{T} C_p^{ig} dT$$

Returns

dH_ig [list[float]] Integrals of ideal gas heat capacity from the reference temperature to the system temperature, [J/(mol)]

`Cpigs_pure()`

Method to calculate and return the ideal-gas heat capacities of every component in the phase. This method is powered by the *HeatCapacityGases* objects, except when all components have the same heat capacity form and a fast implementation has been written for it (currently only polynomials).

Returns

Cp_ig [list[float]] Molar ideal gas heat capacities, [J/(mol*K)]

`Cv()`

Method to calculate and return the constant-volume heat capacity *Cv* of the phase.

$$C_v = T\left(\frac{\partial P}{\partial T}\right)_V^2 / \left(\frac{\partial P}{\partial V}\right)_T + Cp$$

Returns

Cv [float] Constant volume molar heat capacity, [J/(mol*K)]

`Cv_dep()`

Method to calculate and return the difference between the actual *Cv* and the ideal-gas constant volume heat capacity C_v^{ig} of the phase.

$$C_v^{dep} = C_v - C_v^{ig}$$

Returns

Cv_dep [float] Departure ideal gas constant volume heat capacity, [J/(mol*K)]

Cv_ideal_gas()
Method to calculate and return the ideal-gas constant volume heat capacity of the phase.

$$C_v^{ig} = \sum_i z_i C_{p,i}^{ig} - R$$

Returns

Cv [float] Ideal gas constant volume heat capacity, [J/(mol*K)]

Cv_mass()
Method to calculate and return mass constant volume heat capacity of the phase.

$$Cv_{mass} = \frac{1000 Cv_{molar}}{MW}$$

Returns

Cv_mass [float] Mass constant volume heat capacity, [J/(kg*K)]

G()
Method to calculate and return the Gibbs free energy of the phase.

$$G = H - TS$$

Returns

G [float] Gibbs free energy, [J/mol]

property GWPs
Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

Returns

GWPs [list[float]] Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

G_dep()
Method to calculate and return the departure Gibbs free energy of the phase.

$$G_{dep} = H_{dep} - TS_{dep}$$

Returns

G_dep [float] Departure Gibbs free energy, [J/mol]

G_dep_phi_consistency()
Method to calculate and return a consistency check between departure Gibbs free energy, and the fugacity coefficients.

$$G_{dep}^{\text{from phi}} = RT \sum_i z_i \phi_i$$

Returns

error [float] Relative consistency error $|1 - G_{dep}^{\text{from phi}}/G_{dep}^{\text{implemented}}|$, [-]

G_formation_ideal_gas()
Method to calculate and return the ideal-gas Gibbs free energy of formation of the phase (as if the phase was an ideal gas).

$$G_{reactive}^{ig} = H_{reactive}^{ig} - T_{ref}^{ig} S_{reactive}^{ig}$$

Returns

G_formation_ideal_gas [float] Gibbs free energy of formation of the phase on a reactive basis as an ideal gas, [J/(mol)]

`G_ideal_gas()`
Method to calculate and return the ideal-gas Gibbs free energy of the phase.

$$G^{ig} = H^{ig} - TS^{ig}$$

Returns

G_ideal_gas [float] Ideal gas free energy, [J/(mol)]

`G_mass()`
Method to calculate and return mass Gibbs energy of the phase.

$$G_{mass} = \frac{1000G_{molar}}{MW}$$

Returns

G_mass [float] Mass Gibbs energy, [J/(kg)]

`G_min()`
Method to calculate and return the Gibbs free energy of the phase.

$$G = H - TS$$

Returns

G [float] Gibbs free energy, [J/mol]

`G_min_criteria()`
Method to calculate and return the Gibbs energy criteria required for comparing phase stability. This calculation can be faster than calculating the full Gibbs energy. For this comparison to work, all phases must use the ideal gas basis.

$$G^{\text{criteria}} = G^{dep} + RT\sum_i z_i \ln z_i$$

Returns

G_crit [float] Gibbs free energy like criteria [J/mol]

`G_reactive()`
Method to calculate and return the Gibbs free energy of the phase on a reactive basis.

$$G_{reactive} = H_{reactive} - TS_{reactive}$$

Returns

G_reactive [float] Gibbs free energy of the phase on a reactive basis, [J/(mol)]

`property Gfgs`
Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

Returns

Gfgs [list[float]] Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

`property Gfgs_mass`
Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

Returns

Gfgs_mass [list[float]] Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

`H()`

Method to calculate and return the enthalpy of the phase. The reference state for most subclasses is an ideal-gas enthalpy of zero at 298.15 K and 101325 Pa.

Returns

H [float] Molar enthalpy, [J/(mol)]

`H_C_ratio()`

Method to calculate and return the atomic ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.

Returns

H_C_ratio [float] H/C ratio on a molar basis, [-]

Notes

None is returned if no species are present that have carbon atoms.

`H_C_ratio_mass()`

Method to calculate and return the mass ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.

Returns

H_C_ratio_mass [float] H/C ratio on a mass basis, [-]

Notes

None is returned if no species are present that have carbon atoms.

`H_dep_phi_consistency()`

Method to calculate and return a consistency check between departure enthalpy, and the fugacity coefficients' temperature derivatives.

$$H_{dep}^{\text{from phi}} = -RT^2 \sum_i z_i \frac{\partial \ln \phi_i}{\partial T}$$

Returns

error [float] Relative consistency error $|1 - H_{dep}^{\text{from phi}}/H_{dep}^{\text{implemented}}|$, [-]

`H_formation_ideal_gas()`

Method to calculate and return the ideal-gas enthalpy of formation of the phase (as if the phase was an ideal gas).

$$H_{reactive}^{ig} = \sum_i z_i H_{f,i}$$

Returns

H_formation_ideal_gas [float] Enthalpy of formation of the phase on a reactive basis as an ideal gas, [J/mol]

`H_from_phi()`

Method to calculate and return the enthalpy of the fluid as calculated from the ideal-gas enthalpy and the the fugacity coefficients' temperature derivatives.

$$H^{\text{from phi}} = H^{ig} - RT^2\sum_i z_i \frac{\partial \ln \phi_i}{\partial T}$$

Returns

H [float] Enthalpy as calculated from fugacity coefficient temperature derivatives [J/mol]

`H_ideal_gas()`

Method to calculate and return the ideal-gas enthalpy of the phase.

$$H^{ig} = \sum_i z_i H_i^{ig}$$

Returns

H [float] Ideal gas enthalpy, [J/(mol)]

`H_mass()`

Method to calculate and return mass enthalpy of the phase.

$$H_{mass} = \frac{1000 H_{molar}}{MW}$$

Returns

H_mass [float] Mass enthalpy, [J/kg]

`H_phi_consistency()`

Method to calculate and return a consistency check between ideal gas enthalpy behavior, and the fugacity coefficients and their temperature derivatives.

$$H^{\text{from phi}} = H^{ig} - RT^2\sum_i z_i \frac{\partial \ln \phi_i}{\partial T}$$

Returns

error [float] Relative consistency error $|1 - H^{\text{from phi}}/H^{\text{implemented}}|$, [-]

`H_reactive()`

Method to calculate and return the enthalpy of the phase on a reactive basis, using the *Hfs* values of the phase.

$$H_{reactive} = H + \sum_i z_i H_{f,i}$$

Returns

H_reactive [float] Enthalpy of the phase on a reactive basis, [J/mol]

`Hc()`

Method to calculate and return the molar ideal-gas higher heat of combustion of the object, [J/mol]

Returns

Hc [float] Molar higher heat of combustion, [J/(mol)]

`Hc_lower()`

Method to calculate and return the molar ideal-gas lower heat of combustion of the object, [J/mol]

Returns

Hc_lower [float] Molar lower heat of combustion, [J/(mol)]

`Hc_lower_mass()`
Method to calculate and return the mass ideal-gas lower heat of combustion of the object, [J/mol]

Returns

Hc_lower_mass [float] Mass lower heat of combustion, [J/(kg)]

`Hc_lower_normal()`
Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the normal gas volume, [J/m^3]

Returns

Hc_lower_normal [float] Volumetric (normal) lower heat of combustion, [J/(m^3)]

`Hc_lower_standard()`
Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the standard gas volume, [J/m^3]

Returns

Hc_lower_standard [float] Volumetric (standard) lower heat of combustion, [J/(m^3)]

`Hc_mass()`
Method to calculate and return the mass ideal-gas higher heat of combustion of the object, [J/mol]

Returns

Hc_mass [float] Mass higher heat of combustion, [J/(kg)]

`Hc_normal()`
Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the normal gas volume, [J/m^3]

Returns

Hc_normal [float] Volumetric (normal) higher heat of combustion, [J/(m^3)]

`Hc_standard()`
Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the standard gas volume, [J/m^3]

Returns

Hc_normal [float] Volumetric (standard) higher heat of combustion, [J/(m^3)]

`property Hcs`
Higher standard molar heats of combustion for each component, [J/mol].

Returns

Hcs [list[float]] Higher standard molar heats of combustion for each component, [J/mol].

`property Hcs_lower`
Lower standard molar heats of combustion for each component, [J/mol].

Returns

Hcs_lower [list[float]] Lower standard molar heats of combustion for each component, [J/mol].

`property Hcs_lower_mass`
Lower standard heats of combustion for each component, [J/kg].

Returns

Hcs_lower_mass [list[float]] Lower standard heats of combustion for each component, [J/kg].

`property Hcs_mass`
Higher standard heats of combustion for each component, [J/kg].

Returns

Hcs_mass [list[float]] Higher standard heats of combustion for each component, [J/kg].

`property Hf_STPs`
Standard state molar enthalpies of formation for each component, [J/mol].

Returns

Hf_STPs [list[float]] Standard state molar enthalpies of formation for each component, [J/mol].

`property Hf_STPs_mass`
Standard state mass enthalpies of formation for each component, [J/kg].

Returns

Hf_STPs_mass [list[float]] Standard state mass enthalpies of formation for each component, [J/kg].

`property Hfgs`
Ideal gas standard molar enthalpies of formation for each component, [J/mol].

Returns

Hfgs [list[float]] Ideal gas standard molar enthalpies of formation for each component, [J/mol].

`property Hfgs_mass`
Ideal gas standard enthalpies of formation for each component, [J/kg].

Returns

Hfgs_mass [list[float]] Ideal gas standard enthalpies of formation for each component, [J/kg].

`property Hfus_Tms`
Molar heats of fusion for each component at their respective melting points, [J/mol].

Returns

Hfus_Tms [list[float]] Molar heats of fusion for each component at their respective melting points, [J/mol].

`property Hfus_Tms_mass`
Heats of fusion for each component at their respective melting points, [J/kg].

Returns

Hfus_Tms_mass [list[float]] Heats of fusion for each component at their respective melting points, [J/kg].

`property Hsub_Tts`
Heats of sublimation for each component at their respective triple points, [J/mol].

Returns

Hsub_Tts [list[float]] Heats of sublimation for each component at their respective triple points, [J/mol].

property Hsub_Tts_mass
Heats of sublimation for each component at their respective triple points, [J/kg].

Returns

Hsub_Tts_mass [list[float]] Heats of sublimation for each component at their respective triple points, [J/kg].

property Hvap_298s
Molar heats of vaporization for each component at 298.15 K, [J/mol].

Returns

Hvap_298s [list[float]] Molar heats of vaporization for each component at 298.15 K, [J/mol].

property Hvap_298s_mass
Heats of vaporization for each component at 298.15 K, [J/kg].

Returns

Hvap_298s_mass [list[float]] Heats of vaporization for each component at 298.15 K, [J/kg].

property Hvap_Tbs
Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

Returns

Hvap_Tbs [list[float]] Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

property Hvap_Tbs_mass
Heats of vaporization for each component at their respective normal boiling points, [J/kg].

Returns

Hvap_Tbs_mass [list[float]] Heats of vaporization for each component at their respective normal boiling points, [J/kg].

INCOMPRESSIBLE_CONST = 1e+30

property InChI_Keys
InChI Keys for each component, [-].

Returns

InChI_Keys [list[str]] InChI Keys for each component, [-].

property InChIs
InChI strings for each component, [-].

Returns

InChIs [list[str]] InChI strings for each component, [-].

Joule_Thomson()
Method to calculate and return the Joule-Thomson coefficient of the phase.

$$\mu_{JT} = \left(\frac{\partial T}{\partial P}\right)_H = \frac{1}{C_p}\left[T\left(\frac{\partial V}{\partial T}\right)_P - V\right] = \frac{V}{C_p}\left(\beta T - 1\right)$$

Returns

mu_JT [float] Joule-Thomson coefficient [K/Pa]

property LFLs
Lower flammability limits for each component, [-].

Returns

LFLs [list[float]] Lower flammability limits for each component, [-].

`LOG_P_REF_IG = 11.52608845149651`

`MW()`

Method to calculate and return molecular weight of the phase.

$$\text{MW} = \sum_i z_i \text{MW}_i$$

Returns

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

`MW_inv()`

Method to calculate and return inverse of molecular weight of the phase.

$$\frac{1}{\text{MW}} = \frac{1}{\sum_i z_i \text{MW}_i}$$

Returns

MW_inv [float] Inverse of molecular weight, [mol/g]

`property MWs`

Similatiry variables for each component, [g/mol].

Returns

MWs [list[float]] Similatiry variables for each component, [g/mol].

`property ODPs`

Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

Returns

ODPs [list[float]] Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

`PIP()`

Method to calculate and return the phase identification parameter of the phase.

$$\Pi = V\left[\frac{\frac{\partial^2 P}{\partial V \partial T}}{\frac{\partial P}{\partial T}} - \frac{\frac{\partial^2 P}{\partial V^2}}{\frac{\partial P}{\partial V}}\right]$$

Returns

PIP [float] Phase identification parameter, [-]

`property PSRK_groups`

PSRK subgroup: count groups for each component, [-].

Returns

PSRK_groups [list[dict]] PSRK subgroup: count groups for each component, [-].

`P_MAX_FIXED = 1000000000.0`

`P_MIN_FIXED = 0.01`

`P_REF_IG = 101325.0`

`P_REF_IG_INV = 9.869232667160129e-06`

`P_max_at_V(V)`

Dummy method. The idea behind this method, which is implemented by some subclasses, is to calculate the maximum pressure the phase can create at a constant volume, if one exists; returns None otherwise. This method, as a dummy method, always returns None.

Parameters

V [float] Constant molar volume, [m^3/mol]

Returns

P [float] Maximum possible isochoric pressure, [Pa]

`P_transitions()`

Dummy method. The idea behind this method is to calculate any pressures (at constant temperature) which cause the phase properties to become discontinuous.

Returns

P_transitions [list[float]] Transition pressures, [Pa]

`property Parachors`

Parachors for each component, [N^0.25*m^2.75/mol].

Returns

Parachors [list[float]] Parachors for each component, [N^0.25*m^2.75/mol].

`property Pcs`

Critical pressures for each component, [Pa].

Returns

Pcs [list[float]] Critical pressures for each component, [Pa].

`Pmc()`

Method to calculate and return the mechanical critical pressure of the phase.

Returns

Pmc [float] Mechanical critical pressure, [Pa]

`property Psat_298s`

Vapor pressures for each component at 298.15 K, [Pa].

Returns

Psat_298s [list[float]] Vapor pressures for each component at 298.15 K, [Pa].

`Psats_poly_fit = False`

`property Pts`

Triple point pressures for each component, [Pa].

Returns

Pts [list[float]] Triple point pressures for each component, [Pa].

`property PubChems`

Pubchem IDs for each component, [-].

Returns

PubChems [list[int]] Pubchem IDs for each component, [-].

`R = 8.31446261815324`

`R2 = 69.13028862866763`

property RI_Ts

Temperatures at which the refractive indexes were reported for each component, [K].

Returns

RI_Ts [list[float]] Temperatures at which the refractive indexes were reported for each component, [K].

property RIs

Refractive indexes for each component, [-].

Returns

RIs [list[float]] Refractive indexes for each component, [-].

R_inv = 0.12027235504272604

S()

Method to calculate and return the entropy of the phase. The reference state for most subclasses is an ideal-gas entropy of zero at 298.15 K and 101325 Pa.

Returns

S [float] Molar entropy, [J/(mol*K)]

property S0gs

Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

Returns

S0gs [list[float]] Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

property S0gs_mass

Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

Returns

S0gs_mass [list[float]] Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

SG()

Method to calculate and return the standard liquid specific gravity of the phase, using constant liquid pure component densities not calculated by the phase object, at 60 °F.

Returns

SG [float] Specific gravity of the liquid, [-]

Notes

The reference density of water is from the IAPWS-95 standard - 999.0170824078306 kg/m^3.

SG_gas()

Method to calculate and return the specific gravity of the phase with respect to a gas reference density.

Returns

SG_gas [float] Specific gravity of the gas, [-]

Notes

The reference molecular weight of air used is 28.9586 g/mol.

`property STELs`

Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Returns

STELs [list[tuple[(float, str)]]] Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

`S_dep_phi_consistency()`

Method to calculate and return a consistency check between ideal gas entropy behavior, and the fugacity coefficients and their temperature derivatives.

$$S_{dep}^{\text{from phi}} = -\sum_i z_i R \left(\ln \phi_i + T\frac{\partial \ln \phi_i}{\partial T}\right)$$

Returns

error [float] Relative consistency error $|1 - S_{dep}^{\text{from phi}}/S_{dep}^{\text{implemented}}|$, [-]

`S_formation_ideal_gas()`

Method to calculate and return the ideal-gas entropy of formation of the phase (as if the phase was an ideal gas).

$$S_{reactive}^{ig} = \sum_i z_i S_{f,i}$$

Returns

S_formation_ideal_gas [float] Entropy of formation of the phase on a reactive basis as an ideal gas, [J/(mol*K)]

`S_from_phi()`

Method to calculate and return the entropy of the fluid as calculated from the ideal-gas entropy and the the fugacity coefficients' temperature derivatives.

$$S = S^{ig} - \sum_i z_i R \left(\ln \phi_i + T\frac{\partial \ln \phi_i}{\partial T}\right)$$

Returns

S [float] Entropy as calculated from fugacity coefficient temperature derivatives [J/(mol*K)]

`S_ideal_gas()`

Method to calculate and return the ideal-gas entropy of the phase.

$$S^{ig} = \sum_i z_i S_i^{ig} - R\ln\left(\frac{P}{P_{ref}}\right) - R\sum_i z_i \ln(z_i)$$

Returns

S [float] Ideal gas molar entropy, [J/(mol*K)]

`S_mass()`

Method to calculate and return mass entropy of the phase.

$$S_{mass} = \frac{1000 S_{molar}}{MW}$$

Returns

S_mass [float] Mass enthalpy, [J/(kg*K)]

`S_phi_consistency()`

Method to calculate and return a consistency check between ideal gas entropy behavior, and the fugacity coefficients and their temperature derivatives.

$$S = S^{ig} - \sum_i z_i R\left(\ln \phi_i + T\frac{\partial \ln \phi_i}{\partial T}\right)$$

Returns

error [float] Relative consistency error $|1 - S^{\text{from phi}}/S^{\text{implemented}}|$, [-]

`S_reactive()`

Method to calculate and return the entropy of the phase on a reactive basis, using the *Sfs* values of the phase.

$$S_{reactive} = S + \sum_i z_i S_{f,i}$$

Returns

S_reactive [float] Entropy of the phase on a reactive basis, [J/(mol*K)]

`property Sfgs`

Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

Returns

Sfgs [list[float]] Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

`property Sfgs_mass`

Ideal gas standard entropies of formation for each component, [J/(kg*K)].

Returns

Sfgs_mass [list[float]] Ideal gas standard entropies of formation for each component, [J/(kg*K)].

`property Skins`

Whether each compound can be absorbed through the skin or not, [-].

Returns

Skins [list[bool]] Whether each compound can be absorbed through the skin or not, [-].

`property StielPolars`

Stiel polar factors for each component, [-].

Returns

StielPolars [list[float]] Stiel polar factors for each component, [-].

`property Stockmayers`

Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

Returns

Stockmayers [list[float]] Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

`property TWAs`

Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Returns

TWAs [list[tuple[(float, str)]]] Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

`T_MAX_FIXED = 10000.0`

`T_MIN_FIXED = 0.001`

`T_MIN_FLASH = 1e-300`

`T_REF_IG = 298.15`

`T_REF_IG_INV = 0.003354016434680530 3`

The numerical inverse of *T_REF_IG*, stored to save a division.

`T_max_at_V(V)`

Method to calculate the maximum temperature the phase can create at a constant volume, if one exists; returns None otherwise.

Parameters

V [float] Constant molar volume, [m^3/mol]

Pmax [float] Maximum possible isochoric pressure, if already known [Pa]

Returns

T [float] Maximum possible temperature, [K]

`property Tautoignitions`

Autoignition temperatures for each component, [K].

Returns

Tautoignitions [list[float]] Autoignition temperatures for each component, [K].

`property Tbs`

Boiling temperatures for each component, [K].

Returns

Tbs [list[float]] Boiling temperatures for each component, [K].

`property Tcs`

Critical temperatures for each component, [K].

Returns

Tcs [list[float]] Critical temperatures for each component, [K].

`property Tflashs`

Flash point temperatures for each component, [K].

Returns

Tflashs [list[float]] Flash point temperatures for each component, [K].

`Tmc()`

Method to calculate and return the mechanical critical temperature of the phase.

Returns

Tmc [float] Mechanical critical temperature, [K]

`property Tms`

Melting temperatures for each component, [K].

Returns

Tms [list[float]] Melting temperatures for each component, [K].

`property Tts`
Triple point temperatures for each component, [K].

Returns

Tts [list[float]] Triple point temperatures for each component, [K].

`U()`
Method to calculate and return the internal energy of the phase.

$$U = H - PV$$

Returns

U [float] Internal energy, [J/mol]

`property UFLs`
Upper flammability limits for each component, [-].

Returns

UFLs [list[float]] Upper flammability limits for each component, [-].

`property UNIFAC_Dortmund_groups`
UNIFAC_Dortmund_group: count groups for each component, [-].

Returns

UNIFAC_Dortmund_groups [list[dict]] UNIFAC_Dortmund_group: count groups for each component, [-].

`property UNIFAC_Qs`
UNIFAC Q parameters for each component, [-].

Returns

UNIFAC_Qs [list[float]] UNIFAC Q parameters for each component, [-].

`property UNIFAC_Rs`
UNIFAC R parameters for each component, [-].

Returns

UNIFAC_Rs [list[float]] UNIFAC R parameters for each component, [-].

`property UNIFAC_groups`
UNIFAC_group: count groups for each component, [-].

Returns

UNIFAC_groups [list[dict]] UNIFAC_group: count groups for each component, [-].

`U_dep()`
Method to calculate and return the departure internal energy of the phase.

$$U_{dep} = H_{dep} - PV_{dep}$$

Returns

U_dep [float] Departure internal energy, [J/mol]

U_formation_ideal_gas()
Method to calculate and return the ideal-gas internal energy of formation of the phase (as if the phase was an ideal gas).

$$U_{reactive}^{ig} = H_{reactive}^{ig} - P_{ref}^{ig} V^{ig}$$

Returns

U_formation_ideal_gas [float] Internal energy of formation of the phase on a reactive basis as an ideal gas, [J/(mol)]

U_ideal_gas()
Method to calculate and return the ideal-gas internal energy of the phase.

$$U^{ig} = H^{ig} - PV^{ig}$$

Returns

U_ideal_gas [float] Ideal gas internal energy, [J/(mol)]

U_mass()
Method to calculate and return mass internal energy of the phase.

$$U_{mass} = \frac{1000 U_{molar}}{MW}$$

Returns

U_mass [float] Mass internal energy, [J/(kg)]

U_reactive()
Method to calculate and return the internal energy of the phase on a reactive basis.

$$U_{reactive} = H_{reactive} - PV$$

Returns

U_reactive [float] Internal energy of the phase on a reactive basis, [J/(mol)]

V()
Method to return the molar volume of the phase.

Returns

V [float] Molar volume, [m^3/mol]

property VF
Method to return the vapor fraction of the phase. If no vapor/gas is present, 0 is always returned. This method is only available when the phase is linked to an EquilibriumState.

Returns

VF [float] Vapor fraction, [-]

V_MAX_FIXED = 1000000000.0

V_MIN_FIXED = 1e-09

V_dep()
Method to calculate and return the departure (from ideal gas behavior) molar volume of the phase.

$$V_{dep} = V - \frac{RT}{P}$$

Returns

V_dep [float] Departure molar volume, [m^3/mol]

V_from_phi()

Method to calculate and return the molar volume of the fluid as calculated from the pressure derivatives of fugacity coefficients.

$$V^{\text{from phi P der}} = \left(\left(\sum_i z_i \frac{\partial \ln \phi_i}{\partial P}\right) P + 1\right) RT/P$$

Returns

V [float] Molar volume, [m^3/mol]

V_gas()

Method to calculate and return the ideal-gas molar volume of the phase at the chosen reference temperature and pressure, according to the temperature variable *T_gas_ref* and pressure variable *P_gas_ref* of the `thermo.bulk.BulkSettings`.

$$V^{ig} = \frac{RT_{ref}}{P_{ref}}$$

Returns

V_gas [float] Ideal gas molar volume at the reference temperature and pressure, [m^3/mol]

V_gas_normal()

Method to calculate and return the ideal-gas molar volume of the phase at the normal temperature and pressure, according to the temperature variable *T_normal* and pressure variable *P_normal* of the `thermo.bulk.BulkSettings`.

$$V^{ig} = \frac{RT_{norm}}{P_{norm}}$$

Returns

V_gas_normal [float] Ideal gas molar volume at normal temperature and pressure, [m^3/mol]

V_gas_standard()

Method to calculate and return the ideal-gas molar volume of the phase at the standard temperature and pressure, according to the temperature variable *T_standard* and pressure variable *P_standard* of the `thermo.bulk.BulkSettings`.

$$V^{ig} = \frac{RT_{std}}{P_{std}}$$

Returns

V_gas_standard [float] Ideal gas molar volume at standard temperature and pressure, [m^3/mol]

V_ideal_gas()

Method to calculate and return the ideal-gas molar volume of the phase.

$$V^{ig} = \frac{RT}{P}$$

Returns

V [float] Ideal gas molar volume, [m^3/mol]

V_iter(*force=False*)
Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure. This often means the return value of this method is an mpmath *mpf*. This dummy method simply returns the implemented V method.

Returns

V [float or mpf] Molar volume, [m^3/mol]

V_liquid_ref()
Method to calculate and return the liquid reference molar volume according to the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings` and the composition of the phase.

$$V = \sum_i z_i V_i$$

Returns

V_liquid_ref [float] Liquid molar volume at the reference condition, [m^3/mol]

V_mass()
Method to calculate and return the specific volume of the phase.

$$V_{mass} = \frac{1000 \cdot VM}{MW}$$

Returns

V_mass [float] Specific volume of the phase, [m^3/kg]

V_phi_consistency()
Method to calculate and return a consistency check between molar volume, and the fugacity coefficients' pressures derivatives.

$$V^{\text{from phi P der}} = \left(\left(\sum_i z_i \frac{\partial \ln \phi_i}{\partial P}\right) P + 1\right) RT/P$$

Returns

error [float] Relative consistency error $|1 - V^{\text{from phi P der}}/V^{\text{implemented}}|$, [-]

property Van_der_Waals_areas
Unnormalized Van der Waals areas for each component, [m^2/mol].

Returns

Van_der_Waals_areas [list[float]] Unnormalized Van der Waals areas for each component, [m^2/mol].

property Van_der_Waals_volumes
Unnormalized Van der Waals volumes for each component, [m^3/mol].

Returns

Van_der_Waals_volumes [list[float]] Unnormalized Van der Waals volumes for each component, [m^3/mol].

property Vcs
Critical molar volumes for each component, [m^3/mol].

Returns

Vcs [list[float]] Critical molar volumes for each component, [m^3/mol].

Vfgs()

Method to calculate and return the ideal-gas volume fractions of the components of the phase. This is the same as the mole fractions.

Returns

Vfgs [list[float]] Ideal-gas volume fractions of the components of the phase, [-]

Vfls()

Method to calculate and return the ideal-liquid volume fractions of the components of the phase, using the standard liquid densities at the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings` and the composition of the phase.

Returns

Vfls [list[float]] Ideal-liquid volume fractions of the components of the phase, [-]

Vmc()

Method to calculate and return the mechanical critical volume of the phase.

Returns

Vmc [float] Mechanical critical volume, [m^3/mol]

property Vmg_STPs

Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

Returns

Vmg_STPs [list[float]] Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

property Vml_60Fs

Liquid molar volumes for each component at 60 °F, [m^3/mol].

Returns

Vml_60Fs [list[float]] Liquid molar volumes for each component at 60 °F, [m^3/mol].

property Vml_STPs

Liquid molar volumes for each component at STP, [m^3/mol].

Returns

Vml_STPs [list[float]] Liquid molar volumes for each component at STP, [m^3/mol].

property Vml_Tms

Liquid molar volumes for each component at their respective melting points, [m^3/mol].

Returns

Vml_Tms [list[float]] Liquid molar volumes for each component at their respective melting points, [m^3/mol].

property Vms_Tms

Solid molar volumes for each component at their respective melting points, [m^3/mol].

Returns

Vms_Tms [list[float]] Solid molar volumes for each component at their respective melting points, [m^3/mol].

`Wobbe_index()`

Method to calculate and return the molar Wobbe index of the object, [J/mol].

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index [float] Molar Wobbe index, [J/(mol)]

`Wobbe_index_lower()`

Method to calculate and return the molar lower Wobbe index of the object, [J/mol].

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower [float] Molar lower Wobbe index, [J/(mol)]

`Wobbe_index_lower_mass()`

Method to calculate and return the lower mass Wobbe index of the object, [J/kg].

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower_mass [float] Mass lower Wobbe index, [J/(kg)]

`Wobbe_index_lower_normal()`

Method to calculate and return the volumetric normal lower Wobbe index of the object, [J/m^3]. The normal gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower_normal [float] Volumetric normal lower Wobbe index, [J/(m^3)]

`Wobbe_index_lower_standard()`

Method to calculate and return the volumetric standard lower Wobbe index of the object, [J/m^3]. The standard gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{lower}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_lower_standard [float] Volumetric standard lower Wobbe index, [J/(m^3)]

`Wobbe_index_mass()`

Method to calculate and return the mass Wobbe index of the object, [J/kg].

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_mass [float] Mass Wobbe index, [J/(kg)]

`Wobbe_index_normal()`
Method to calculate and return the volumetric normal Wobbe index of the object, [J/m^3]. The normal gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index [float] Volumetric normal Wobbe index, [J/(m^3)]

`Wobbe_index_standard()`
Method to calculate and return the volumetric standard Wobbe index of the object, [J/m^3]. The standard gas volume is used in this calculation.

$$I_W = \frac{H_{comb}^{higher}}{\sqrt{\text{SG}}}$$

Returns

Wobbe_index_standard [float] Volumetric standard Wobbe index, [J/(m^3)]

`Z()`
Method to calculate and return the compressibility factor of the phase.

$$Z = \frac{PV}{RT}$$

Returns

Z [float] Compressibility factor, [-]

`property Zcs`
Critical compressibilities for each component, [-].

Returns

Zcs [list[float]] Critical compressibilities for each component, [-].

`Zmc()`
Method to calculate and return the mechanical critical compressibility of the phase.

Returns

Zmc [float] Mechanical critical compressibility, [-]

`__eq__(other)`
Return self==value.

`__hash__()`
Method to calculate and return a hash representing the exact state of the object.

Returns

hash [int] Hash of the object, [-]

`activities()`
Method to calculate and return the activities of each component in the phase [-].

$$a_i(T, P, x; f_i^0) = \frac{f_i(T, P, x)}{f_i^0(T, P_i^0)}$$

Returns

activities [list[float]] Activities, [-]

as_json()
Method to create a JSON-friendly serialization of the phase which can be stored, and reloaded later.

Returns

json_repr [dict] JSON-friendly representation, [-]

Examples

```
>>> import json
>>> from thermo import IAPWS95Liquid
>>> phase = IAPWS95Liquid(T=300, P=1e5, zs=[1])
>>> new_phase = Phase.from_json(json.loads(json.dumps(phase.as_json())))
>>> assert phase == new_phase
```

atom_fractions()
Method to calculate and return the atomic composition of the phase; returns a dictionary of atom fraction (by count), containing only those elements who are present.

Returns

atom_fractions [dict[str: float]] Atom fractions, [-]

atom_mass_fractions()
Method to calculate and return the atomic mass fractions of the phase; returns a dictionary of atom fraction (by mass), containing only those elements who are present.

Returns

atom_mass_fractions [dict[str: float]] Atom mass fractions, [-]

property atomss
Breakdown of each component into its elements and their counts, as a dict, [-].

Returns

atomss [list[dict]] Breakdown of each component into its elements and their counts, as a dict, [-].

property beta
Method to return the phase fraction of this phase. This method is only available when the phase is linked to an EquilibriumState.

Returns

beta [float] Phase fraction on a molar basis, [-]

property beta_mass
Method to return the mass phase fraction of this phase. This method is only available when the phase is linked to an EquilibriumState.

Returns

beta_mass [float] Phase fraction on a mass basis, [-]

property beta_volume
Method to return the volumetric phase fraction of this phase. This method is only available when the phase is linked to an EquilibriumState.

Returns

beta_volume [float] Phase fraction on a volumetric basis, [-] `property charges`

Charge number (valence) for each component, [-].

Returns

charges [list[float]] Charge number (valence) for each component, [-].

`chemical_potential()`

Method to calculate and return the chemical potentials of each component in the phase [-]. For a pure substance, this is the molar Gibbs energy on a reactive basis.

$$\frac{\partial G}{\partial n_i}_{T,P,N_{j \ne i}}$$

Returns

chemical_potential [list[float]] Chemical potentials, [J/mol]

`composition_independent = False`

`property conductivities`

Electrical conductivities for each component, [S/m].

Returns

conductivities [list[float]] Electrical conductivities for each component, [S/m].

`property conductivity_Ts`

Temperatures at which the electrical conductivities for each component were measured, [K].

Returns

conductivity_Ts [list[float]] Temperatures at which the electrical conductivities for each component were measured, [K].

`d2P_dT2()`

Method to calculate and return the second temperature derivative of pressure of the phase.

Returns

d2P_dT2 [float] Second temperature derivative of pressure, [Pa/K^2]

`d2P_dTdV()`

Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.

Returns

d2P_dTdV [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

`d2P_dTdrho()`

Method to calculate and return the temperature derivative and then molar density derivative of the pressure of the phase.

$$\frac{\partial^2 P}{\partial T \partial \rho} = -V^2 \left(\frac{\partial^2 P}{\partial T \partial V}\right)$$

Returns

d2P_dTdrho [float] Temperature derivative and then molar density derivative of the pressure, [Pa*m^3/(K*mol)]

`d2P_dV2()`

Method to calculate and return the second volume derivative of pressure of the phase.

Returns

d2P_dV2 [float] Second volume derivative of pressure, [Pa*mol^2/m^6]

d2P_dVdT()

Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase. This is an alias of *d2P_dTdV*.

$$\frac{\partial^2 P}{\partial V \partial T}$$

Returns

d2P_dVdT [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

d2P_drho2()

Method to calculate and return the second molar density derivative of pressure of the phase.

$$\frac{\partial^2 P}{\partial \rho^2} = -V^2 \left(-V^2 \left(\frac{\partial^2 P}{\partial V^2}\right)_T - 2V \left(\frac{\partial P}{\partial V}\right)_T\right)$$

Returns

d2P_drho2 [float] Second molar density derivative of pressure, [Pa*m^6/mol^2]

d2T_dP2()

Method to calculate and return the constant-volume second pressure derivative of temperature of the phase.

$$\left(\frac{\partial^2 T}{\partial P^2}\right)_V = -\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial T}{\partial P}\right)^3_V$$

Returns

d2T_dP2 [float] Constant-volume second pressure derivative of temperature, [K/Pa^2]

d2T_dP2_V()

Method to calculate and return the constant-volume second pressure derivative of temperature of the phase.

$$\left(\frac{\partial^2 T}{\partial P^2}\right)_V = -\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial T}{\partial P}\right)^3_V$$

Returns

d2T_dP2 [float] Constant-volume second pressure derivative of temperature, [K/Pa^2]

d2T_dPdV()

Method to calculate and return the derivative of pressure and then the derivative of volume of temperature of the phase.

$$\left(\frac{\partial^2 T}{\partial P \partial V}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]\left(\frac{\partial P}{\partial T}\right)^{-3}_V$$

Returns

d2T_dPdV [float] Derivative of pressure and then the derivative of volume of temperature, [K*mol/(Pa*m^3)]

d2T_dPdrho()

Method to calculate and return the pressure derivative and then molar density derivative of the temperature of the phase.

$$\frac{\partial^2 T}{\partial P \partial \rho} = -V^2 \left(\frac{\partial^2 T}{\partial P \partial V}\right)$$

Returns

d2T_dPdrho [float] Pressure derivative and then molar density derivative of the temperature, [K*m^3/(Pa*mol)]

`d2T_dV2()`

Method to calculate and return the constant-pressure second volume derivative of temperature of the phase.

$$\left(\frac{\partial^2 T}{\partial V^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial V^2}\right)_T \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right] \left(\frac{\partial P}{\partial T}\right)_V^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]$$

Returns

d2T_dV2 [float] Constant-pressure second volume derivative of temperature, [K*mol^2/m^6]

`d2T_dV2_P()`

Method to calculate and return the constant-pressure second volume derivative of temperature of the phase.

$$\left(\frac{\partial^2 T}{\partial V^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial V^2}\right)_T \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right] \left(\frac{\partial P}{\partial T}\right)_V^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right]$$

Returns

d2T_dV2 [float] Constant-pressure second volume derivative of temperature, [K*mol^2/m^6]

`d2T_dVdP()`

Method to calculate and return the derivative of pressure and then the derivative of volume of temperature of the phase.

$$\left(\frac{\partial^2 T}{\partial P \partial V}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right) \left(\frac{\partial P}{\partial T}\right)_V - \left(\frac{\partial P}{\partial V}\right)_T \left(\frac{\partial^2 P}{\partial T^2}\right)_V\right] \left(\frac{\partial P}{\partial T}\right)_V^{-3}$$

Returns

d2T_dPdV [float] Derivative of pressure and then the derivative of volume of temperature, [K*mol/(Pa*m^3)]

`d2T_drho2()`

Method to calculate and return the second molar density derivative of temperature of the phase.

$$\frac{\partial^2 T}{\partial \rho^2} = -V^2 \left(-V^2 \left(\frac{\partial^2 T}{\partial V^2}\right)_P - 2V \left(\frac{\partial T}{\partial V}\right)_P\right)$$

Returns

d2T_drho2 [float] Second molar density derivative of temperature, [K*m^6/mol^2]

`d2V_dP2()`

Method to calculate and return the constant-temperature pressure derivative of volume of the phase.

$$\left(\frac{\partial^2 V}{\partial P^2}\right)_T = -\frac{\left(\frac{\partial^2 P}{\partial V^2}\right)_T}{\left(\frac{\partial P}{\partial V}\right)_T^3}$$

Returns

d2V_dP2 [float] Constant-temperature pressure derivative of volume, [m^3/(mol*Pa^2)]

`d2V_dP2_T()`

Method to calculate and return the constant-temperature pressure derivative of volume of the phase.

$$\left(\frac{\partial^2 V}{\partial P^2}\right)_T = -\frac{\left(\frac{\partial^2 P}{\partial V^2}\right)_T}{\left(\frac{\partial P}{\partial V}\right)_T^3}$$

Returns

d2V_dP2 [float] Constant-temperature pressure derivative of volume, [m^3/(mol*Pa^2)]

`d2V_dPdT()`

Method to calculate and return the derivative of pressure and then the derivative of temperature of volume of the phase.

$$\left(\frac{\partial^2 V}{\partial T \partial P}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]\left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

Returns

d2V_dPdT [float] Derivative of pressure and then the derivative of temperature of volume, [m^3/(mol*K*Pa)]

`d2V_dT2()`

Method to calculate and return the constant-pressure second temperature derivative of volume of the phase.

$$\left(\frac{\partial^2 V}{\partial T^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right]\left(\frac{\partial P}{\partial V}\right)_T^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]$$

Returns

d2V_dT2 [float] Constant-pressure second temperature derivative of volume, [m^3/(mol*K^2)]

`d2V_dT2_P()`

Method to calculate and return the constant-pressure second temperature derivative of volume of the phase.

$$\left(\frac{\partial^2 V}{\partial T^2}\right)_P = -\left[\left(\frac{\partial^2 P}{\partial T^2}\right)_V \left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial T \partial V}\right)\right]\left(\frac{\partial P}{\partial V}\right)_T^{-2} + \left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]$$

Returns

d2V_dT2 [float] Constant-pressure second temperature derivative of volume, [m^3/(mol*K^2)]

`d2V_dTdP()`

Method to calculate and return the derivative of pressure and then the derivative of temperature of volume of the phase.

$$\left(\frac{\partial^2 V}{\partial T \partial P}\right) = -\left[\left(\frac{\partial^2 P}{\partial T \partial V}\right)\left(\frac{\partial P}{\partial V}\right)_T - \left(\frac{\partial P}{\partial T}\right)_V \left(\frac{\partial^2 P}{\partial V^2}\right)_T\right]\left(\frac{\partial P}{\partial V}\right)_T^{-3}$$

Returns

d2V_dPdT [float] Derivative of pressure and then the derivative of temperature of volume, [m^3/(mol*K*Pa)]

`d2rho_dP2()`

Method to calculate and return the second pressure derivative of molar density of the phase.

$$\frac{\partial^2 \rho}{\partial P^2} = -\frac{1}{V^2}\left(\frac{\partial^2 V}{\partial P^2}\right)_T + \frac{2}{V^3}\left(\frac{\partial V}{\partial P}\right)_T^2$$

Returns

d2rho_dP2 [float] Second pressure derivative of molar density, [mol^2/(Pa*m^6)]

d2rho_dPdT()

Method to calculate and return the pressure derivative and then temperature derivative of the molar density of the phase.

$$\frac{\partial^2 \rho}{\partial P \partial T} = -\frac{1}{V^2}\left(\frac{\partial^2 V}{\partial P \partial T}\right) + \frac{2}{V^3}\left(\frac{\partial V}{\partial T}\right)_P \left(\frac{\partial V}{\partial P}\right)_T$$

Returns

d2rho_dPdT [float] Pressure derivative and then temperature derivative of the molar density, [mol/(m^3*K*Pa)]

d2rho_dT2()

Method to calculate and return the second temperature derivative of molar density of the phase.

$$\frac{\partial^2 \rho}{\partial T^2} = -\frac{1}{V^2}\left(\frac{\partial^2 V}{\partial T^2}\right)_P + \frac{2}{V^3}\left(\frac{\partial V}{\partial T}\right)_T^2$$

Returns

d2rho_dT2 [float] Second temperature derivative of molar density, [mol^2/(K*m^6)]

dA_dP()

Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial U}{\partial P}\right)_T$$

Returns

dA_dP [float] Constant-temperature pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dP_T()

Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial U}{\partial P}\right)_T$$

Returns

dA_dP [float] Constant-temperature pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dP_V()

Method to calculate and return the constant-volume pressure derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial P}\right)_V = \left(\frac{\partial H}{\partial P}\right)_V - V - S\left(\frac{\partial T}{\partial P}\right)_V - T\left(\frac{\partial S}{\partial P}\right)_V$$

Returns

dA_dP_V [float] Constant-volume pressure derivative of Helmholtz energy, [J/(mol*Pa)]

dA_dT()

Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial U}{\partial T}\right)_P$$

Returns

dA_dT [float] Constant-pressure temperature derivative of Helmholtz energy, [J/(mol*K)]

dA_dT_P()

Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_P = -T\left(\frac{\partial S}{\partial T}\right)_P - S + \left(\frac{\partial U}{\partial T}\right)_P$$

Returns

dA_dT [float] Constant-pressure temperature derivative of Helmholtz energy, [J/(mol*K)]

dA_dT_V()

Method to calculate and return the constant-volume temperature derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial T}\right)_V = \left(\frac{\partial H}{\partial T}\right)_V - V\left(\frac{\partial P}{\partial T}\right)_V - T\left(\frac{\partial S}{\partial T}\right)_V - S$$

Returns

dA_dT_V [float] Constant-volume temperature derivative of Helmholtz energy, [J/(mol*K)]

dA_dV_P()

Method to calculate and return the constant-pressure volume derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial V}\right)_P = \left(\frac{\partial A}{\partial T}\right)_P \left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dA_dV_P [float] Constant-pressure volume derivative of Helmholtz energy, [J/(m^3)]

dA_dV_T()

Method to calculate and return the constant-temperature volume derivative of Helmholtz energy.

$$\left(\frac{\partial A}{\partial V}\right)_T = \left(\frac{\partial A}{\partial P}\right)_T \left(\frac{\partial P}{\partial V}\right)_T$$

Returns

dA_dV_T [float] Constant-temperature volume derivative of Helmholtz energy, [J/(m^3)]

dA_mass_dP(*prop='dA_dP'*)

Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.

$$\left(\frac{\partial A_{\text{mass}}}{\partial P}\right)_T$$

Returns

dA_mass_dP [float] The pressure derivative of mass Helmholtz energy of the phase at constant temperature, [J/mol/Pa]

dA_mass_dP_T(*prop='dA_dP_T'*)

Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.

$$\left(\frac{\partial A_{\text{mass}}}{\partial P}\right)_T$$

Returns

dA_mass_dP_T [float] The pressure derivative of mass Helmholtz energy of the phase at constant temperature, [J/mol/Pa]

`dA_mass_dP_V`(*prop='dA_dP_V'*)

Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant volume.

$$\left(\frac{\partial A_{\text{mass}}}{\partial P}\right)_V$$

Returns

dA_mass_dP_V [float] The pressure derivative of mass Helmholtz energy of the phase at constant volume, [J/mol/Pa]

`dA_mass_dT`(*prop='dA_dT'*)

Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.

$$\left(\frac{\partial A_{\text{mass}}}{\partial T}\right)_P$$

Returns

dA_mass_dT [float] The temperature derivative of mass Helmholtz energy of the phase at constant pressure, [J/mol/K]

`dA_mass_dT_P`(*prop='dA_dT_P'*)

Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.

$$\left(\frac{\partial A_{\text{mass}}}{\partial T}\right)_P$$

Returns

dA_mass_dT_P [float] The temperature derivative of mass Helmholtz energy of the phase at constant pressure, [J/mol/K]

`dA_mass_dT_V`(*prop='dA_dT_V'*)

Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant volume.

$$\left(\frac{\partial A_{\text{mass}}}{\partial T}\right)_V$$

Returns

dA_mass_dT_V [float] The temperature derivative of mass Helmholtz energy of the phase at constant volume, [J/mol/K]

dA_mass_dV_P(*prop='dA_dV_P'*)

Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant pressure.

$$\left(\frac{\partial A_{\text{mass}}}{\partial V}\right)_P$$

Returns

dA_mass_dV_P [float] The volume derivative of mass Helmholtz energy of the phase at constant pressure, [J/mol/m^3/mol]

dA_mass_dV_T(*prop='dA_dV_T'*)

Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant temperature.

$$\left(\frac{\partial A_{\text{mass}}}{\partial V}\right)_T$$

Returns

dA_mass_dV_T [float] The volume derivative of mass Helmholtz energy of the phase at constant temperature, [J/mol/m^3/mol]

dCpigs_dT_pure()

Method to calculate and return the first temperature derivative of ideal-gas heat capacities of every component in the phase. This method is powered by the *HeatCapacityGases* objects, except when all components have the same heat capacity form and a fast implementation has been written for it (currently only polynomials).

$$\frac{\partial C_p^{ig}}{\partial T}$$

Returns

dCp_ig_dT [list[float]] First temperature derivatives of molar ideal gas heat capacities, [J/(mol*K^2)]

dCv_dP_T()

Method to calculate the pressure derivative of Cv, constant volume heat capacity, at constant temperature.

$$\left(\frac{\partial C_v}{\partial P}\right)_T = -T\,\mathrm{dPdT_V}(P)\frac{d}{dP}\mathrm{dVdT_P}(P) - T\,\mathrm{dVdT_P}(P)\frac{d}{dP}\mathrm{dPdT_V}(P) + \frac{d}{dP}\mathrm{Cp}(P)$$

Returns

dCv_dP_T [float] Pressure derivative of constant volume heat capacity at constant temperature, [J/mol/K/Pa]

Notes

Requires *d2V_dTdP*, *d2P_dTdP*, and *d2H_dTdP*.

dCv_dT_P()
Method to calculate the temperature derivative of Cv, constant volume heat capacity, at constant pressure.

$$\left(\frac{\partial C_v}{\partial T}\right)_P = -\frac{T \operatorname{dPdT_V}^2(T) \frac{d}{dT} \operatorname{dPdV_T}(T)}{\operatorname{dPdV_T}^2(T)} + \frac{2T \operatorname{dPdT_V}(T) \frac{d}{dT} \operatorname{dPdT_V}(T)}{\operatorname{dPdV_T}(T)} + \frac{\operatorname{dPdT_V}^2(T)}{\operatorname{dPdV_T}(T)} + \frac{d}{dT} \operatorname{Cp}(T)$$

Returns

dCv_dT_P [float] Temperature derivative of constant volume heat capacity at constant pressure, [J/mol/K^2]

Notes

Requires *d2P_dT2_PV*, *d2P_dVdT_TP*, and *d2H_dT2*.

dCv_mass_dP_T(*prop='dCv_dP_T'*)
Method to calculate and return the pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature.

$$\left(\frac{\partial Cv_{\text{mass}}}{\partial P}\right)_T$$

Returns

dCv_mass_dP_T [float] The pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature, [J/(mol*K)/Pa]

dCv_mass_dT_P(*prop='dCv_dT_P'*)
Method to calculate and return the temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure.

$$\left(\frac{\partial Cv_{\text{mass}}}{\partial T}\right)_P$$

Returns

dCv_mass_dT_P [float] The temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure, [J/(mol*K)/K]

dG_dP()
Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_T = -T\left(\frac{\partial S}{\partial P}\right)_T + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dG_dP [float] Constant-temperature pressure derivative of Gibbs free energy, [J/(mol*Pa)]

dG_dP_T()

Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_{T} = -T\left(\frac{\partial S}{\partial P}\right)_{T} + \left(\frac{\partial H}{\partial P}\right)_{T}$$

Returns

dG_dP [float] Constant-temperature pressure derivative of Gibbs free energy, [J/(mol*Pa)]

dG_dP_V()

Method to calculate and return the constant-volume pressure derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial P}\right)_{V} = -T\left(\frac{\partial S}{\partial P}\right)_{V} - S\left(\frac{\partial T}{\partial P}\right)_{V} + \left(\frac{\partial H}{\partial P}\right)_{V}$$

Returns

dG_dP_V [float] Constant-volume pressure derivative of Gibbs free energy, [J/(mol*Pa)]

dG_dT()

Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_{P} = -T\left(\frac{\partial S}{\partial T}\right)_{P} - S + \left(\frac{\partial H}{\partial T}\right)_{P}$$

Returns

dG_dT [float] Constant-pressure temperature derivative of Gibbs free energy, [J/(mol*K)]

dG_dT_P()

Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_{P} = -T\left(\frac{\partial S}{\partial T}\right)_{P} - S + \left(\frac{\partial H}{\partial T}\right)_{P}$$

Returns

dG_dT [float] Constant-pressure temperature derivative of Gibbs free energy, [J/(mol*K)]

dG_dT_V()

Method to calculate and return the constant-volume temperature derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial T}\right)_{V} = -T\left(\frac{\partial S}{\partial T}\right)_{V} - S + \left(\frac{\partial H}{\partial T}\right)_{V}$$

Returns

dG_dT_V [float] Constant-volume temperature derivative of Gibbs free energy, [J/(mol*K)]

dG_dV_P()

Method to calculate and return the constant-pressure volume derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial V}\right)_{P} = \left(\frac{\partial G}{\partial T}\right)_{P}\left(\frac{\partial T}{\partial V}\right)_{P}$$

Returns

dG_dV_P [float] Constant-pressure volume derivative of Gibbs free energy, [J/(m^3)]

dG_dV_T()

Method to calculate and return the constant-temperature volume derivative of Gibbs free energy.

$$\left(\frac{\partial G}{\partial V}\right)_{T} = \left(\frac{\partial G}{\partial P}\right)_{T}\left(\frac{\partial P}{\partial V}\right)_{T}$$

Returns

dG_dV_T [float] Constant-temperature volume derivative of Gibbs free energy, [J/(m^3)]

`dG_mass_dP`(*prop='dG_dP'*)

Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.

$$\left(\frac{\partial G_{\text{mass}}}{\partial P}\right)_T$$

Returns

dG_mass_dP [float] The pressure derivative of mass Gibbs free energy of the phase at constant temperature, [J/mol/Pa]

`dG_mass_dP_T`(*prop='dG_dP_T'*)

Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.

$$\left(\frac{\partial G_{\text{mass}}}{\partial P}\right)_T$$

Returns

dG_mass_dP_T [float] The pressure derivative of mass Gibbs free energy of the phase at constant temperature, [J/mol/Pa]

`dG_mass_dP_V`(*prop='dG_dP_V'*)

Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant volume.

$$\left(\frac{\partial G_{\text{mass}}}{\partial P}\right)_V$$

Returns

dG_mass_dP_V [float] The pressure derivative of mass Gibbs free energy of the phase at constant volume, [J/mol/Pa]

`dG_mass_dT`(*prop='dG_dT'*)

Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.

$$\left(\frac{\partial G_{\text{mass}}}{\partial T}\right)_P$$

Returns

dG_mass_dT [float] The temperature derivative of mass Gibbs free energy of the phase at constant pressure, [J/mol/K]

dG_mass_dT_P(*prop='dG_dT_P'*)
Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.

$$\left(\frac{\partial G_{\text{mass}}}{\partial T}\right)_P$$

Returns

dG_mass_dT_P [float] The temperature derivative of mass Gibbs free energy of the phase at constant pressure, [J/mol/K]

dG_mass_dT_V(*prop='dG_dT_V'*)
Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant volume.

$$\left(\frac{\partial G_{\text{mass}}}{\partial T}\right)_V$$

Returns

dG_mass_dT_V [float] The temperature derivative of mass Gibbs free energy of the phase at constant volume, [J/mol/K]

dG_mass_dV_P(*prop='dG_dV_P'*)
Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant pressure.

$$\left(\frac{\partial G_{\text{mass}}}{\partial V}\right)_P$$

Returns

dG_mass_dV_P [float] The volume derivative of mass Gibbs free energy of the phase at constant pressure, [J/mol/m^3/mol]

dG_mass_dV_T(*prop='dG_dV_T'*)
Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant temperature.

$$\left(\frac{\partial G_{\text{mass}}}{\partial V}\right)_T$$

Returns

dG_mass_dV_T [float] The volume derivative of mass Gibbs free energy of the phase at constant temperature, [J/mol/m^3/mol]

dH_dP_T()
Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.

Returns

dH_dP_T [float] Pressure derivative of enthalpy, [J/(mol*Pa)]

`dH_dT_P()`

Method to calculate and return the temperature derivative of enthalpy of the phase at constant pressure.

Returns

dH_dT_P [float] Temperature derivative of enthalpy, [J/(mol*K)]

`dH_dns()`

Method to calculate and return the mole number derivative of the enthalpy of the phase.

$$\frac{\partial H}{\partial n_i}$$

Returns

dH_dns [list[float]] Mole number derivatives of the enthalpy of the phase, [J/mol^2]

`dH_mass_dP`(*prop='dH_dP'*)

Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.

$$\left(\frac{\partial H_{\text{mass}}}{\partial P}\right)_T$$

Returns

dH_mass_dP [float] The pressure derivative of mass enthalpy of the phase at constant temperature, [J/mol/Pa]

`dH_mass_dP_T`(*prop='dH_dP_T'*)

Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.

$$\left(\frac{\partial H_{\text{mass}}}{\partial P}\right)_T$$

Returns

dH_mass_dP_T [float] The pressure derivative of mass enthalpy of the phase at constant temperature, [J/mol/Pa]

`dH_mass_dP_V`(*prop='dH_dP_V'*)

Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant volume.

$$\left(\frac{\partial H_{\text{mass}}}{\partial P}\right)_V$$

Returns

dH_mass_dP_V [float] The pressure derivative of mass enthalpy of the phase at constant volume, [J/mol/Pa]

`dH_mass_dT`(*prop='dH_dT'*)

Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.

$$\left(\frac{\partial H_{\text{mass}}}{\partial T}\right)_P$$

Returns

dH_mass_dT [float] The temperature derivative of mass enthalpy of the phase at constant pressure, [J/mol/K]

`dH_mass_dT_P`(*prop='dH_dT_P'*)
Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.

$$\left(\frac{\partial H_{\text{mass}}}{\partial T}\right)_P$$

Returns

dH_mass_dT_P [float] The temperature derivative of mass enthalpy of the phase at constant pressure, [J/mol/K]

`dH_mass_dT_V`(*prop='dH_dT_V'*)
Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant volume.

$$\left(\frac{\partial H_{\text{mass}}}{\partial T}\right)_V$$

Returns

dH_mass_dT_V [float] The temperature derivative of mass enthalpy of the phase at constant volume, [J/mol/K]

`dH_mass_dV_P`(*prop='dH_dV_P'*)
Method to calculate and return the volume derivative of mass enthalpy of the phase at constant pressure.

$$\left(\frac{\partial H_{\text{mass}}}{\partial V}\right)_P$$

Returns

dH_mass_dV_P [float] The volume derivative of mass enthalpy of the phase at constant pressure, [J/mol/m^3/mol]

`dH_mass_dV_T`(*prop='dH_dV_T'*)
Method to calculate and return the volume derivative of mass enthalpy of the phase at constant temperature.

$$\left(\frac{\partial H_{\text{mass}}}{\partial V}\right)_T$$

Returns

dH_mass_dV_T [float] The volume derivative of mass enthalpy of the phase at constant temperature, [J/mol/m^3/mol]

dP_dP_A(*property='P'*, *differentiate_by='P'*, *at_constant='A'*)
Method to calculate and return the pressure derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial P}\right)_{A}$$

Returns

dP_dP_A [float] The pressure derivative of pressure of the phase at constant Helmholtz energy, [Pa/Pa]

dP_dP_G(*property='P'*, *differentiate_by='P'*, *at_constant='G'*)
Method to calculate and return the pressure derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial P}\right)_{G}$$

Returns

dP_dP_G [float] The pressure derivative of pressure of the phase at constant Gibbs energy, [Pa/Pa]

dP_dP_H(*property='P'*, *differentiate_by='P'*, *at_constant='H'*)
Method to calculate and return the pressure derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial P}\right)_{H}$$

Returns

dP_dP_H [float] The pressure derivative of pressure of the phase at constant enthalpy, [Pa/Pa]

dP_dP_S(*property='P'*, *differentiate_by='P'*, *at_constant='S'*)
Method to calculate and return the pressure derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial P}\right)_{S}$$

Returns

dP_dP_S [float] The pressure derivative of pressure of the phase at constant entropy, [Pa/Pa]

dP_dP_T()
Method to calculate and return the pressure derivative of pressure of the phase at constant temperature.

Returns

dP_dP_T [float] Pressure derivative of pressure of the phase at constant temperature, [-]

dP_dP_U(*property='P'*, *differentiate_by='P'*, *at_constant='U'*)
Method to calculate and return the pressure derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial P}\right)_{U}$$

Returns

dP_dP_U [float] The pressure derivative of pressure of the phase at constant internal energy, [Pa/Pa]

dP_dP_V()

Method to calculate and return the pressure derivative of pressure of the phase at constant volume.

Returns

dP_dP_V [float] Pressure derivative of pressure of the phase at constant volume, [-]

dP_dT()

Method to calculate and return the first temperature derivative of pressure of the phase.

Returns

dP_dT [float] First temperature derivative of pressure, [Pa/K]

dP_dT_A(*property='P'*, *differentiate_by='T'*, *at_constant='A'*)

Method to calculate and return the temperature derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial T}\right)_{A}$$

Returns

dP_dT_A [float] The temperature derivative of pressure of the phase at constant Helmholtz energy, [Pa/K]

dP_dT_G(*property='P'*, *differentiate_by='T'*, *at_constant='G'*)

Method to calculate and return the temperature derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial T}\right)_{G}$$

Returns

dP_dT_G [float] The temperature derivative of pressure of the phase at constant Gibbs energy, [Pa/K]

dP_dT_H(*property='P'*, *differentiate_by='T'*, *at_constant='H'*)

Method to calculate and return the temperature derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial T}\right)_{H}$$

Returns

dP_dT_H [float] The temperature derivative of pressure of the phase at constant enthalpy, [Pa/K]

dP_dT_P()
Method to calculate and return the temperature derivative of temperature of the phase at constant pressure.

Returns

dP_dT_P [float] Temperature derivative of temperature, [-]

dP_dT_S(*property='P'*, *differentiate_by='T'*, *at_constant='S'*)
Method to calculate and return the temperature derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial T}\right)_S$$

Returns

dP_dT_S [float] The temperature derivative of pressure of the phase at constant entropy, [Pa/K]

dP_dT_U(*property='P'*, *differentiate_by='T'*, *at_constant='U'*)
Method to calculate and return the temperature derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial T}\right)_U$$

Returns

dP_dT_U [float] The temperature derivative of pressure of the phase at constant internal energy, [Pa/K]

dP_dV()
Method to calculate and return the first volume derivative of pressure of the phase.

Returns

dP_dV [float] First volume derivative of pressure, [Pa*mol/m^3]

dP_dV_A(*property='P'*, *differentiate_by='V'*, *at_constant='A'*)
Method to calculate and return the volume derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial V}\right)_A$$

Returns

dP_dV_A [float] The volume derivative of pressure of the phase at constant Helmholtz energy, [Pa/m^3/mol]

dP_dV_G(*property='P'*, *differentiate_by='V'*, *at_constant='G'*)
Method to calculate and return the volume derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial V}\right)_G$$

Returns

dP_dV_G [float] The volume derivative of pressure of the phase at constant Gibbs energy, [Pa/m^3/mol]

dP_dV_H(*property='P'*, *differentiate_by='V'*, *at_constant='H'*)
Method to calculate and return the volume derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial V}\right)_H$$

Returns

dP_dV_H [float] The volume derivative of pressure of the phase at constant enthalpy, [Pa/m^3/mol]

dP_dV_P()
Method to calculate and return the volume derivative of pressure of the phase at constant pressure.

Returns

dP_dV_P [float] Volume derivative of pressure of the phase at constant pressure, [Pa*mol/m^3]

dP_dV_S(*property='P'*, *differentiate_by='V'*, *at_constant='S'*)
Method to calculate and return the volume derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial V}\right)_S$$

Returns

dP_dV_S [float] The volume derivative of pressure of the phase at constant entropy, [Pa/m^3/mol]

dP_dV_U(*property='P'*, *differentiate_by='V'*, *at_constant='U'*)
Method to calculate and return the volume derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial V}\right)_U$$

Returns

dP_dV_U [float] The volume derivative of pressure of the phase at constant internal energy, [Pa/m^3/mol]

dP_drho()
Method to calculate and return the molar density derivative of pressure of the phase.

$$\frac{\partial P}{\partial \rho} = -V^2 \left(\frac{\partial P}{\partial V}\right)_T$$

Returns

dP_drho [float] Molar density derivative of pressure, [Pa*m^3/mol]

dP_drho_A(*property='P'*, *differentiate_by='rho'*, *at_constant='A'*)
Method to calculate and return the density derivative of pressure of the phase at constant Helmholtz energy.

$$\left(\frac{\partial P}{\partial \rho}\right)_{A}$$

Returns

dP_drho_A [float] The density derivative of pressure of the phase at constant Helmholtz energy, [Pa/mol/m^3]

dP_drho_G(*property='P'*, *differentiate_by='rho'*, *at_constant='G'*)
Method to calculate and return the density derivative of pressure of the phase at constant Gibbs energy.

$$\left(\frac{\partial P}{\partial \rho}\right)_{G}$$

Returns

dP_drho_G [float] The density derivative of pressure of the phase at constant Gibbs energy, [Pa/mol/m^3]

dP_drho_H(*property='P'*, *differentiate_by='rho'*, *at_constant='H'*)
Method to calculate and return the density derivative of pressure of the phase at constant enthalpy.

$$\left(\frac{\partial P}{\partial \rho}\right)_{H}$$

Returns

dP_drho_H [float] The density derivative of pressure of the phase at constant enthalpy, [Pa/mol/m^3]

dP_drho_S(*property='P'*, *differentiate_by='rho'*, *at_constant='S'*)
Method to calculate and return the density derivative of pressure of the phase at constant entropy.

$$\left(\frac{\partial P}{\partial \rho}\right)_{S}$$

Returns

dP_drho_S [float] The density derivative of pressure of the phase at constant entropy, [Pa/mol/m^3]

dP_drho_U(*property='P'*, *differentiate_by='rho'*, *at_constant='U'*)
Method to calculate and return the density derivative of pressure of the phase at constant internal energy.

$$\left(\frac{\partial P}{\partial \rho}\right)_{U}$$

Returns

dP_drho_U [float] The density derivative of pressure of the phase at constant internal energy, [Pa/mol/m^3]

`dS_dP_T()`
Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.

Returns

dS_dP_T [float] Pressure derivative of entropy, [J/(mol*K*Pa)]

`dS_dV_P()`
Method to calculate and return the volume derivative of entropy of the phase at constant pressure.

Returns

dS_dV_P [float] Volume derivative of entropy, [J/(K*m^3)]

`dS_dV_T()`
Method to calculate and return the volume derivative of entropy of the phase at constant temperature.

Returns

dS_dV_T [float] Volume derivative of entropy, [J/(K*m^3)]

`dS_dns()`
Method to calculate and return the mole number derivative of the entropy of the phase.

$$\frac{\partial S}{\partial n_i}$$

Returns

dS_dns [list[float]] Mole number derivatives of the entropy of the phase, [J/(mol^2*K)]

`dS_mass_dP`(*prop='dS_dP'*)
Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.

$$\left(\frac{\partial S_{\text{mass}}}{\partial P}\right)_T$$

Returns

dS_mass_dP [float] The pressure derivative of mass entropy of the phase at constant temperature, [J/(mol*K)/Pa]

`dS_mass_dP_T`(*prop='dS_dP_T'*)
Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.

$$\left(\frac{\partial S_{\text{mass}}}{\partial P}\right)_T$$

Returns

dS_mass_dP_T [float] The pressure derivative of mass entropy of the phase at constant temperature, [J/(mol*K)/Pa]

`dS_mass_dP_V`(*prop='dS_dP_V'*)
Method to calculate and return the pressure derivative of mass entropy of the phase at constant volume.

$$\left(\frac{\partial S_{\text{mass}}}{\partial P}\right)_V$$

Returns

dS_mass_dP_V [float] The pressure derivative of mass entropy of the phase at constant volume, [J/(mol*K)/Pa]

`dS_mass_dT`(*prop='dS_dT'*)

Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.

$$\left(\frac{\partial S_{\text{mass}}}{\partial T}\right)_P$$

Returns

dS_mass_dT [float] The temperature derivative of mass entropy of the phase at constant pressure, [J/(mol*K)/K]

`dS_mass_dT_P`(*prop='dS_dT_P'*)

Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.

$$\left(\frac{\partial S_{\text{mass}}}{\partial T}\right)_P$$

Returns

dS_mass_dT_P [float] The temperature derivative of mass entropy of the phase at constant pressure, [J/(mol*K)/K]

`dS_mass_dT_V`(*prop='dS_dT_V'*)

Method to calculate and return the temperature derivative of mass entropy of the phase at constant volume.

$$\left(\frac{\partial S_{\text{mass}}}{\partial T}\right)_V$$

Returns

dS_mass_dT_V [float] The temperature derivative of mass entropy of the phase at constant volume, [J/(mol*K)/K]

`dS_mass_dV_P`(*prop='dS_dV_P'*)

Method to calculate and return the volume derivative of mass entropy of the phase at constant pressure.

$$\left(\frac{\partial S_{\text{mass}}}{\partial V}\right)_P$$

Returns

dS_mass_dV_P [float] The volume derivative of mass entropy of the phase at constant pressure, [J/(mol*K)/m^3/mol]

dS_mass_dV_T(*prop='dS_dV_T'*)
Method to calculate and return the volume derivative of mass entropy of the phase at constant temperature.

$$\left(\frac{\partial S_{\text{mass}}}{\partial V}\right)_T$$

Returns

dS_mass_dV_T [float] The volume derivative of mass entropy of the phase at constant temperature, [J/(mol*K)/m^3/mol]

dT_dP()
Method to calculate and return the constant-volume pressure derivative of temperature of the phase.

$$\left(\frac{\partial T}{\partial P}\right)_V = \frac{1}{\left(\frac{\partial P}{\partial T}\right)_V}$$

Returns

dT_dP [float] Constant-volume pressure derivative of temperature, [K/Pa]

dT_dP_A(*property='T'*, *differentiate_by='P'*, *at_constant='A'*)
Method to calculate and return the pressure derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial P}\right)_A$$

Returns

dT_dP_A [float] The pressure derivative of temperature of the phase at constant Helmholtz energy, [K/Pa]

dT_dP_G(*property='T'*, *differentiate_by='P'*, *at_constant='G'*)
Method to calculate and return the pressure derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial P}\right)_G$$

Returns

dT_dP_G [float] The pressure derivative of temperature of the phase at constant Gibbs energy, [K/Pa]

dT_dP_H(*property='T'*, *differentiate_by='P'*, *at_constant='H'*)
Method to calculate and return the pressure derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial P}\right)_H$$

Returns

dT_dP_H [float] The pressure derivative of temperature of the phase at constant enthalpy, [K/Pa]

dT_dP_S(*property='T'*, *differentiate_by='P'*, *at_constant='S'*)
Method to calculate and return the pressure derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial P}\right)_S$$

Returns

dT_dP_S [float] The pressure derivative of temperature of the phase at constant entropy, [K/Pa]

dT_dP_T()
Method to calculate and return the pressure derivative of temperature of the phase at constant temperature.

Returns

dT_dP_T [float] Pressure derivative of temperature of the phase at constant temperature, [K/Pa]

dT_dP_U(*property='T'*, *differentiate_by='P'*, *at_constant='U'*)
Method to calculate and return the pressure derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial P}\right)_U$$

Returns

dT_dP_U [float] The pressure derivative of temperature of the phase at constant internal energy, [K/Pa]

dT_dP_V()
Method to calculate and return the constant-volume pressure derivative of temperature of the phase.

$$\left(\frac{\partial T}{\partial P}\right)_V = \frac{1}{\left(\frac{\partial P}{\partial T}\right)_V}$$

Returns

dT_dP [float] Constant-volume pressure derivative of temperature, [K/Pa]

dT_dT_A(*property='T'*, *differentiate_by='T'*, *at_constant='A'*)
Method to calculate and return the temperature derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial T}\right)_A$$

Returns

dT_dT_A [float] The temperature derivative of temperature of the phase at constant Helmholtz energy, [K/K]

dT_dT_G(*property='T'*, *differentiate_by='T'*, *at_constant='G'*)
Method to calculate and return the temperature derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial T}\right)_G$$

Returns

dT_dT_G [float] The temperature derivative of temperature of the phase at constant Gibbs energy, [K/K]

dT_dT_H(*property='T'*, *differentiate_by='T'*, *at_constant='H'*)
Method to calculate and return the temperature derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial T}\right)_H$$

Returns

dT_dT_H [float] The temperature derivative of temperature of the phase at constant enthalpy, [K/K]

dT_dT_P()
Method to calculate and return the temperature derivative of temperature of the phase at constant pressure.

Returns

dT_dT_P [float] Temperature derivative of temperature of the phase at constant pressure, [-]

dT_dT_S(*property='T'*, *differentiate_by='T'*, *at_constant='S'*)
Method to calculate and return the temperature derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial T}\right)_S$$

Returns

dT_dT_S [float] The temperature derivative of temperature of the phase at constant entropy, [K/K]

dT_dT_U(*property='T'*, *differentiate_by='T'*, *at_constant='U'*)
Method to calculate and return the temperature derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial T}\right)_U$$

Returns

dT_dT_U [float] The temperature derivative of temperature of the phase at constant internal energy, [K/K]

dT_dT_V()
Method to calculate and return the temperature derivative of temperature of the phase at constant volume.

Returns

dT_dT_V [float] Temperature derivative of temperature of the phase at constant volume, [-]

dT_dV()

Method to calculate and return the constant-pressure volume derivative of temperature of the phase.

$$\left(\frac{\partial T}{\partial V}\right)_P = \frac{1}{\left(\frac{\partial V}{\partial T}\right)_P}$$

Returns

dT_dV [float] Constant-pressure volume derivative of temperature, [K*m^3/(m^3)]

dT_dV_A(*property='T'*, *differentiate_by='V'*, *at_constant='A'*)

Method to calculate and return the volume derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial V}\right)_A$$

Returns

dT_dV_A [float] The volume derivative of temperature of the phase at constant Helmholtz energy, [K/m^3/mol]

dT_dV_G(*property='T'*, *differentiate_by='V'*, *at_constant='G'*)

Method to calculate and return the volume derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial V}\right)_G$$

Returns

dT_dV_G [float] The volume derivative of temperature of the phase at constant Gibbs energy, [K/m^3/mol]

dT_dV_H(*property='T'*, *differentiate_by='V'*, *at_constant='H'*)

Method to calculate and return the volume derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial V}\right)_H$$

Returns

dT_dV_H [float] The volume derivative of temperature of the phase at constant enthalpy, [K/m^3/mol]

dT_dV_P()

Method to calculate and return the constant-pressure volume derivative of temperature of the phase.

$$\left(\frac{\partial T}{\partial V}\right)_P = \frac{1}{\left(\frac{\partial V}{\partial T}\right)_P}$$

Returns

dT_dV [float] Constant-pressure volume derivative of temperature, [K*m^3/(m^3)]

dT_dV_S(*property='T'*, *differentiate_by='V'*, *at_constant='S'*)
Method to calculate and return the volume derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial V}\right)_S$$

Returns

dT_dV_S [float] The volume derivative of temperature of the phase at constant entropy, [K/m^3/mol]

dT_dV_T()
Method to calculate and return the volume derivative of temperature of the phase at constant temperature.

Returns

dT_dV_T [float] Pressure derivative of temperature of the phase at constant temperature, [K*mol/m^3]

dT_dV_U(*property='T'*, *differentiate_by='V'*, *at_constant='U'*)
Method to calculate and return the volume derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial V}\right)_U$$

Returns

dT_dV_U [float] The volume derivative of temperature of the phase at constant internal energy, [K/m^3/mol]

dT_drho()
Method to calculate and return the molar density derivative of temperature of the phase.

$$\frac{\partial T}{\partial \rho} = -V^2\left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dT_drho [float] Molar density derivative of temperature, [K*m^3/mol]

dT_drho_A(*property='T'*, *differentiate_by='rho'*, *at_constant='A'*)
Method to calculate and return the density derivative of temperature of the phase at constant Helmholtz energy.

$$\left(\frac{\partial T}{\partial \rho}\right)_A$$

Returns

dT_drho_A [float] The density derivative of temperature of the phase at constant Helmholtz energy, [K/mol/m^3]

dT_drho_G(*property='T'*, *differentiate_by='rho'*, *at_constant='G'*)
Method to calculate and return the density derivative of temperature of the phase at constant Gibbs energy.

$$\left(\frac{\partial T}{\partial \rho}\right)_{G}$$

Returns

dT_drho_G [float] The density derivative of temperature of the phase at constant Gibbs energy, [K/mol/m^3]

dT_drho_H(*property='T'*, *differentiate_by='rho'*, *at_constant='H'*)

Method to calculate and return the density derivative of temperature of the phase at constant enthalpy.

$$\left(\frac{\partial T}{\partial \rho}\right)_{H}$$

Returns

dT_drho_H [float] The density derivative of temperature of the phase at constant enthalpy, [K/mol/m^3]

dT_drho_S(*property='T'*, *differentiate_by='rho'*, *at_constant='S'*)

Method to calculate and return the density derivative of temperature of the phase at constant entropy.

$$\left(\frac{\partial T}{\partial \rho}\right)_{S}$$

Returns

dT_drho_S [float] The density derivative of temperature of the phase at constant entropy, [K/mol/m^3]

dT_drho_U(*property='T'*, *differentiate_by='rho'*, *at_constant='U'*)

Method to calculate and return the density derivative of temperature of the phase at constant internal energy.

$$\left(\frac{\partial T}{\partial \rho}\right)_{U}$$

Returns

dT_drho_U [float] The density derivative of temperature of the phase at constant internal energy, [K/mol/m^3]

dU_dP()

Method to calculate and return the constant-temperature pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_{T} = -P\left(\frac{\partial V}{\partial P}\right)_{T} - V + \left(\frac{\partial H}{\partial P}\right)_{T}$$

Returns

dU_dP [float] Constant-temperature pressure derivative of internal energy, [J/(mol*Pa)]

dU_dP_T()
Method to calculate and return the constant-temperature pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_T = -P\left(\frac{\partial V}{\partial P}\right)_T - V + \left(\frac{\partial H}{\partial P}\right)_T$$

Returns

dU_dP [float] Constant-temperature pressure derivative of internal energy, [J/(mol*Pa)]

dU_dP_V()
Method to calculate and return the constant-volume pressure derivative of internal energy.

$$\left(\frac{\partial U}{\partial P}\right)_V = \left(\frac{\partial H}{\partial P}\right)_V - V$$

Returns

dU_dP_V [float] Constant-volume pressure derivative of internal energy, [J/(mol*Pa)]

dU_dT()
Method to calculate and return the constant-pressure temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_P = -P\left(\frac{\partial V}{\partial T}\right)_P + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dU_dT [float] Constant-pressure temperature derivative of internal energy, [J/(mol*K)]

dU_dT_P()
Method to calculate and return the constant-pressure temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_P = -P\left(\frac{\partial V}{\partial T}\right)_P + \left(\frac{\partial H}{\partial T}\right)_P$$

Returns

dU_dT [float] Constant-pressure temperature derivative of internal energy, [J/(mol*K)]

dU_dT_V()
Method to calculate and return the constant-volume temperature derivative of internal energy.

$$\left(\frac{\partial U}{\partial T}\right)_V = \left(\frac{\partial H}{\partial T}\right)_V - V\left(\frac{\partial P}{\partial T}\right)_V$$

Returns

dU_dT_V [float] Constant-volume temperature derivative of internal energy, [J/(mol*K)]

dU_dV_P()
Method to calculate and return the constant-pressure volume derivative of internal energy.

$$\left(\frac{\partial U}{\partial V}\right)_P = \left(\frac{\partial U}{\partial T}\right)_P\left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dU_dV_P [float] Constant-pressure volume derivative of internal energy, [J/(m^3)]

dU_dV_T()
Method to calculate and return the constant-temperature volume derivative of internal energy.

$$\left(\frac{\partial U}{\partial V}\right)_T = \left(\frac{\partial U}{\partial P}\right)_T\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

dU_dV_T [float] Constant-temperature volume derivative of internal energy, [J/(m^3)]

`dU_mass_dP`(*prop='dU_dP'*)
Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.

$$\left(\frac{\partial U_{\text{mass}}}{\partial P}\right)_T$$

Returns

dU_mass_dP [float] The pressure derivative of mass internal energy of the phase at constant temperature, [J/mol/Pa]

`dU_mass_dP_T`(*prop='dU_dP_T'*)
Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.

$$\left(\frac{\partial U_{\text{mass}}}{\partial P}\right)_T$$

Returns

dU_mass_dP_T [float] The pressure derivative of mass internal energy of the phase at constant temperature, [J/mol/Pa]

`dU_mass_dP_V`(*prop='dU_dP_V'*)
Method to calculate and return the pressure derivative of mass internal energy of the phase at constant volume.

$$\left(\frac{\partial U_{\text{mass}}}{\partial P}\right)_V$$

Returns

dU_mass_dP_V [float] The pressure derivative of mass internal energy of the phase at constant volume, [J/mol/Pa]

`dU_mass_dT`(*prop='dU_dT'*)
Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.

$$\left(\frac{\partial U_{\text{mass}}}{\partial T}\right)_P$$

Returns

dU_mass_dT [float] The temperature derivative of mass internal energy of the phase at constant pressure, [J/mol/K]

dU_mass_dT_P(*prop='dU_dT_P'*)
Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.

$$\left(\frac{\partial U_{\text{mass}}}{\partial T}\right)_P$$

Returns

dU_mass_dT_P [float] The temperature derivative of mass internal energy of the phase at constant pressure, [J/mol/K]

dU_mass_dT_V(*prop='dU_dT_V'*)
Method to calculate and return the temperature derivative of mass internal energy of the phase at constant volume.

$$\left(\frac{\partial U_{\text{mass}}}{\partial T}\right)_V$$

Returns

dU_mass_dT_V [float] The temperature derivative of mass internal energy of the phase at constant volume, [J/mol/K]

dU_mass_dV_P(*prop='dU_dV_P'*)
Method to calculate and return the volume derivative of mass internal energy of the phase at constant pressure.

$$\left(\frac{\partial U_{\text{mass}}}{\partial V}\right)_P$$

Returns

dU_mass_dV_P [float] The volume derivative of mass internal energy of the phase at constant pressure, [J/mol/m^3/mol]

dU_mass_dV_T(*prop='dU_dV_T'*)
Method to calculate and return the volume derivative of mass internal energy of the phase at constant temperature.

$$\left(\frac{\partial U_{\text{mass}}}{\partial V}\right)_T$$

Returns

dU_mass_dV_T [float] The volume derivative of mass internal energy of the phase at constant temperature, [J/mol/m^3/mol]

dV_dP()
Method to calculate and return the constant-temperature pressure derivative of volume of the phase.

$$\left(\frac{\partial V}{\partial P}\right)_T = -\left(\frac{\partial V}{\partial T}\right)_P \left(\frac{\partial T}{\partial P}\right)_V$$

Returns

dV_dP [float] Constant-temperature pressure derivative of volume, [m^3/(mol*Pa)]

dV_dP_A(*property='V'*, *differentiate_by='P'*, *at_constant='A'*)
Method to calculate and return the pressure derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial P}\right)_A$$

Returns

dV_dP_A [float] The pressure derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/Pa]

dV_dP_G(*property='V'*, *differentiate_by='P'*, *at_constant='G'*)
Method to calculate and return the pressure derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial P}\right)_G$$

Returns

dV_dP_G [float] The pressure derivative of volume of the phase at constant Gibbs energy, [m^3/mol/Pa]

dV_dP_H(*property='V'*, *differentiate_by='P'*, *at_constant='H'*)
Method to calculate and return the pressure derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial P}\right)_H$$

Returns

dV_dP_H [float] The pressure derivative of volume of the phase at constant enthalpy, [m^3/mol/Pa]

dV_dP_S(*property='V'*, *differentiate_by='P'*, *at_constant='S'*)
Method to calculate and return the pressure derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial P}\right)_S$$

Returns

dV_dP_S [float] The pressure derivative of volume of the phase at constant entropy, [m^3/mol/Pa]

dV_dP_T()
Method to calculate and return the constant-temperature pressure derivative of volume of the phase.

$$\left(\frac{\partial V}{\partial P}\right)_T = -\left(\frac{\partial V}{\partial T}\right)_P \left(\frac{\partial T}{\partial P}\right)_V$$

Returns

dV_dP [float] Constant-temperature pressure derivative of volume, [m^3/(mol*Pa)]

dV_dP_U(*property='V'*, *differentiate_by='P'*, *at_constant='U'*)

Method to calculate and return the pressure derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial P}\right)_{U}$$

Returns

dV_dP_U [float] The pressure derivative of volume of the phase at constant internal energy, [m^3/mol/Pa]

dV_dP_V()

Method to calculate and return the volume derivative of pressure of the phase at constant volume.

Returns

dV_dP_V [float] Pressure derivative of volume of the phase at constant pressure, [m^3/(mol*Pa)]

dV_dT()

Method to calculate and return the constant-pressure temperature derivative of volume of the phase.

$$\left(\frac{\partial V}{\partial T}\right)_{P} = \frac{-\left(\frac{\partial P}{\partial T}\right)_{V}}{\left(\frac{\partial P}{\partial V}\right)_{T}}$$

Returns

dV_dT [float] Constant-pressure temperature derivative of volume, [m^3/(mol*K)]

dV_dT_A(*property='V'*, *differentiate_by='T'*, *at_constant='A'*)

Method to calculate and return the temperature derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial T}\right)_{A}$$

Returns

dV_dT_A [float] The temperature derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/K]

dV_dT_G(*property='V'*, *differentiate_by='T'*, *at_constant='G'*)

Method to calculate and return the temperature derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial T}\right)_{G}$$

Returns

dV_dT_G [float] The temperature derivative of volume of the phase at constant Gibbs energy, [m^3/mol/K]

dV_dT_H(*property='V'*, *differentiate_by='T'*, *at_constant='H'*)
Method to calculate and return the temperature derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial T}\right)_H$$

Returns

dV_dT_H [float] The temperature derivative of volume of the phase at constant enthalpy, [m^3/mol/K]

dV_dT_P()
Method to calculate and return the constant-pressure temperature derivative of volume of the phase.

$$\left(\frac{\partial V}{\partial T}\right)_P = \frac{-\left(\frac{\partial P}{\partial T}\right)_V}{\left(\frac{\partial P}{\partial V}\right)_T}$$

Returns

dV_dT [float] Constant-pressure temperature derivative of volume, [m^3/(mol*K)]

dV_dT_S(*property='V'*, *differentiate_by='T'*, *at_constant='S'*)
Method to calculate and return the temperature derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial T}\right)_S$$

Returns

dV_dT_S [float] The temperature derivative of volume of the phase at constant entropy, [m^3/mol/K]

dV_dT_U(*property='V'*, *differentiate_by='T'*, *at_constant='U'*)
Method to calculate and return the temperature derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial T}\right)_U$$

Returns

dV_dT_U [float] The temperature derivative of volume of the phase at constant internal energy, [m^3/mol/K]

dV_dT_V()
Method to calculate and return the temperature derivative of volume of the phase at constant volume.

Returns

dV_dT_V [float] Temperature derivative of volume of the phase at constant volume, [m^3/(mol*K)]

dV_dV_A(*property='V'*, *differentiate_by='V'*, *at_constant='A'*)
Method to calculate and return the volume derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial V}\right)_A$$

Returns

dV_dV_A [float] The volume derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/m^3/mol]

dV_dV_G(*property='V'*, *differentiate_by='V'*, *at_constant='G'*)

Method to calculate and return the volume derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial V}\right)_G$$

Returns

dV_dV_G [float] The volume derivative of volume of the phase at constant Gibbs energy, [m^3/mol/m^3/mol]

dV_dV_H(*property='V'*, *differentiate_by='V'*, *at_constant='H'*)

Method to calculate and return the volume derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial V}\right)_H$$

Returns

dV_dV_H [float] The volume derivative of volume of the phase at constant enthalpy, [m^3/mol/m^3/mol]

dV_dV_P()

Method to calculate and return the volume derivative of volume of the phase at constant pressure.

Returns

dV_dV_P [float] Volume derivative of volume of the phase at constant pressure, [-]

dV_dV_S(*property='V'*, *differentiate_by='V'*, *at_constant='S'*)

Method to calculate and return the volume derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial V}\right)_S$$

Returns

dV_dV_S [float] The volume derivative of volume of the phase at constant entropy, [m^3/mol/m^3/mol]

dV_dV_T()

Method to calculate and return the volume derivative of volume of the phase at constant temperature.

Returns

dV_dV_T [float] Volume derivative of volume of the phase at constant temperature, [-]

dV_dV_U(*property='V'*, *differentiate_by='V'*, *at_constant='U'*)
Method to calculate and return the volume derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial V}\right)_{U}$$

Returns

dV_dV_U [float] The volume derivative of volume of the phase at constant internal energy, [m^3/mol/m^3/mol]

dV_dns()
Method to calculate and return the mole number derivatives of the molar volume V of the phase.

$$\frac{\partial V}{\partial n_i}$$

Returns

dV_dns [list[float]] Mole number derivatives of the molar volume of the phase, [m^3]

dV_drho_A(*property='V'*, *differentiate_by='rho'*, *at_constant='A'*)
Method to calculate and return the density derivative of volume of the phase at constant Helmholtz energy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{A}$$

Returns

dV_drho_A [float] The density derivative of volume of the phase at constant Helmholtz energy, [m^3/mol/mol/m^3]

dV_drho_G(*property='V'*, *differentiate_by='rho'*, *at_constant='G'*)
Method to calculate and return the density derivative of volume of the phase at constant Gibbs energy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{G}$$

Returns

dV_drho_G [float] The density derivative of volume of the phase at constant Gibbs energy, [m^3/mol/mol/m^3]

dV_drho_H(*property='V'*, *differentiate_by='rho'*, *at_constant='H'*)
Method to calculate and return the density derivative of volume of the phase at constant enthalpy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{H}$$

Returns

dV_drho_H [float] The density derivative of volume of the phase at constant enthalpy, [m^3/mol/mol/m^3]

dV_drho_S(*property='V'*, *differentiate_by='rho'*, *at_constant='S'*)

Method to calculate and return the density derivative of volume of the phase at constant entropy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{S}$$

Returns

dV_drho_S [float] The density derivative of volume of the phase at constant entropy, [m^3/mol/mol/m^3]

dV_drho_U(*property='V'*, *differentiate_by='rho'*, *at_constant='U'*)

Method to calculate and return the density derivative of volume of the phase at constant internal energy.

$$\left(\frac{\partial V}{\partial \rho}\right)_{U}$$

Returns

dV_drho_U [float] The density derivative of volume of the phase at constant internal energy, [m^3/mol/mol/m^3]

dZ_dP()

Method to calculate and return the pressure derivative of compressibility of the phase.

$$\frac{\partial Z}{\partial P} = \frac{V + P\left(\frac{\partial V}{\partial P}\right)_T}{RT}$$

Returns

dZ_dP [float] Pressure derivative of compressibility, [1/Pa]

dZ_dT()

Method to calculate and return the temperature derivative of compressibility of the phase.

$$\frac{\partial Z}{\partial P} = P\frac{\left(\frac{\partial V}{\partial T}\right)_P - \frac{-V}{T}}{RT}$$

Returns

dZ_dT [float] Temperature derivative of compressibility, [1/K]

dZ_dV()

Method to calculate and return the volume derivative of compressibility of the phase.

$$\frac{\partial Z}{\partial V} = \frac{P - \rho\left(\frac{\partial P}{\partial \rho}\right)_T}{RT}$$

Returns

dZ_dV [float] Volume derivative of compressibility, [mol/(m^3)]

dZ_dns()

Method to calculate and return the mole number derivatives of the compressibility factor Z of the phase.

$$\frac{\partial Z}{\partial n_i}$$

Returns

dZ_dns [list[float]] Mole number derivatives of the compressibility factor of the phase, [1/mol]

`dZ_dzs()`

Method to calculate and return the mole fraction derivatives of the compressibility factor Z of the phase.

$$\frac{\partial Z}{\partial z_i}$$

Returns

dZ_dzs [list[float]] Mole fraction derivatives of the compressibility factor of the phase, [-]

`dfugacities_dP()`

Method to calculate and return the pressure derivative of the fugacities of the components in the phase.

$$\frac{\partial f_i}{\partial P} = z_i \left(P \frac{\partial \phi_i}{\partial P} + \phi_i\right)$$

Returns

dfugacities_dP [list[float]] Pressure derivative of fugacities of all components in the phase, [-]

Notes

For models without pressure dependence of fugacity, the returned result may not be exactly zero due to inaccuracy in floating point results; results are likely on the order of 1e-14 or lower in that case.

`dfugacities_dT()`

Method to calculate and return the temperature derivative of fugacities of the phase.

$$\frac{\partial f_i}{\partial T} = P z_i \frac{\partial \ln \phi_i}{\partial T}$$

Returns

dfugacities_dT [list[float]] Temperature derivative of fugacities of all components in the phase, [Pa/K]

`dfugacities_dns()`

Method to calculate and return the mole number derivative of the fugacities of the components in the phase.

if i != j:

$$\frac{\partial f_i}{\partial n_j} = P\phi_i z_i \left(\frac{\partial \ln \phi_i}{\partial n_j} - 1\right)$$

if i == j:

$$\frac{\partial f_i}{\partial n_j} = P\phi_i z_i \left(\frac{\partial \ln \phi_i}{\partial n_j} - 1\right) + P\phi_i$$

Returns

dfugacities_dns [list[list[float]]] Mole number derivatives of the fugacities of all components in the phase, [Pa/mol]

`dfugacity_dP()`

Method to calculate and return the pressure derivative of fugacity of the phase; provided the phase is 1 component.

Returns

dfugacity_dP [list[float]] Fugacity first pressure derivative, [-]

`dfugacity_dT()`

Method to calculate and return the temperature derivative of fugacity of the phase; provided the phase is 1 component.

Returns

dfugacity_dT [list[float]] Fugacity first temperature derivative, [Pa/K]

`property dipoles`

Dipole moments for each component, [debye].

Returns

dipoles [list[float]] Dipole moments for each component, [debye].

`disobaric_expansion_dP()`

Method to calculate and return the pressure derivative of isobatic expansion coefficient of the phase.

$$\frac{\partial \beta}{\partial P} = \frac{1}{V}\left(\left(\frac{\partial^2 V}{\partial T \partial P}\right) - \frac{\left(\frac{\partial V}{\partial T}\right)_P \left(\frac{\partial V}{\partial P}\right)_T}{V}\right)$$

Returns

dbeta_dP [float] Pressure derivative of isobaric coefficient of a thermal expansion, [1/(K*Pa)]

`disobaric_expansion_dT()`

Method to calculate and return the temperature derivative of isobatic expansion coefficient of the phase.

$$\frac{\partial \beta}{\partial T} = \frac{1}{V}\left(\left(\frac{\partial^2 V}{\partial T^2}\right)_P - \left(\frac{\partial V}{\partial T}\right)_P^2 / V\right)$$

Returns

dbeta_dT [float] Temperature derivative of isobaric coefficient of a thermal expansion, [1/K^2]

`disothermal_compressibility_dT()`

Method to calculate and return the temperature derivative of isothermal compressibility of the phase.

$$\frac{\partial \kappa}{\partial T} = -\frac{\left(\frac{\partial^2 V}{\partial P \partial T}\right)}{V} + \frac{\left(\frac{\partial V}{\partial P}\right)_T \left(\frac{\partial V}{\partial T}\right)_P}{V^2}$$

Returns

dkappa_dT [float] First temperature derivative of isothermal coefficient of compressibility, [1/(Pa*K)]

`dkappa_dT()`

Method to calculate and return the temperature derivative of isothermal compressibility of the phase.

$$\frac{\partial \kappa}{\partial T} = -\frac{\left(\frac{\partial^2 V}{\partial P \partial T}\right)}{V} + \frac{\left(\frac{\partial V}{\partial P}\right)_T \left(\frac{\partial V}{\partial T}\right)_P}{V^2}$$

Returns

dkappa_dT [float] First temperature derivative of isothermal coefficient of compressibility, [1/(Pa*K)]

`dlnfugacities_dns()`
Method to calculate and return the mole number derivative of the log of fugacities of the components in the phase.

$$\frac{\partial \ln f_i}{\partial n_j} = \frac{1}{f_i}\frac{\partial f_i}{\partial n_j}$$

Returns

dlnfugacities_dns [list[list[float]]] Mole number derivatives of the log of fugacities of all components in the phase, [log(Pa)/mol]

`dlnfugacities_dzs()`
Method to calculate and return the mole fraction derivative of the log of fugacities of the components in the phase.

$$\frac{\partial \ln f_i}{\partial z_j} = \frac{1}{f_i}\frac{\partial f_i}{\partial z_j}$$

Returns

dlnfugacities_dzs [list[list[float]]] Mole fraction derivatives of the log of fugacities of all components in the phase, [log(Pa)]

`dlnphis_dP()`
Method to calculate and return the pressure derivative of the log of fugacity coefficients of each component in the phase.

Returns

dlnphis_dP [list[float]] First pressure derivative of log fugacity coefficients, [1/Pa]

`dlnphis_dT()`
Method to calculate and return the temperature derivative of the log of fugacity coefficients of each component in the phase.

Returns

dlnphis_dT [list[float]] First temperature derivative of log fugacity coefficients, [1/K]

`dphis_dP()`
Method to calculate and return the pressure derivative of fugacity coefficients of the phase.

$$\frac{\partial \phi_i}{\partial P} = \phi_i \frac{\partial \ln \phi_i}{\partial P}$$

Returns

dphis_dP [list[float]] Pressure derivative of fugacity coefficients of all components in the phase, [1/Pa]

`dphis_dT()`
Method to calculate and return the temperature derivative of fugacity coefficients of the phase.

$$\frac{\partial \phi_i}{\partial T} = \phi_i \frac{\partial \ln \phi_i}{\partial T}$$

Returns

dphis_dT [list[float]] Temperature derivative of fugacity coefficients of all components in the phase, [1/K]

dphis_dzs()
Method to calculate and return the molar composition derivative of fugacity coefficients of the phase.

$$\frac{\partial \phi_i}{\partial z_j} = \phi_i \frac{\partial \ln \phi_i}{\partial z_j}$$

Returns

dphis_dzs [list[list[float]]] Molar derivative of fugacity coefficients of all components in the phase, [-]

drho_dP()
Method to calculate and return the pressure derivative of molar density of the phase.

$$\frac{\partial \rho}{\partial P} = -\frac{1}{V^2}\left(\frac{\partial V}{\partial P}\right)_T$$

Returns

drho_dP [float] Pressure derivative of Molar density, [mol/(Pa*m^3)]

drho_dP_A(*property='rho'*, *differentiate_by='P'*, *at_constant='A'*)
Method to calculate and return the pressure derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial P}\right)_A$$

Returns

drho_dP_A [float] The pressure derivative of density of the phase at constant Helmholtz energy, [mol/m^3/Pa]

drho_dP_G(*property='rho'*, *differentiate_by='P'*, *at_constant='G'*)
Method to calculate and return the pressure derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial P}\right)_G$$

Returns

drho_dP_G [float] The pressure derivative of density of the phase at constant Gibbs energy, [mol/m^3/Pa]

drho_dP_H(*property='rho'*, *differentiate_by='P'*, *at_constant='H'*)
Method to calculate and return the pressure derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial P}\right)_H$$

Returns

drho_dP_H [float] The pressure derivative of density of the phase at constant enthalpy, [mol/m^3/Pa]

drho_dP_S(*property='rho'*, *differentiate_by='P'*, *at_constant='S'*)
Method to calculate and return the pressure derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial P}\right)_S$$

Returns

drho_dP_S [float] The pressure derivative of density of the phase at constant entropy, [mol/m^3/Pa]

drho_dP_U(*property='rho'*, *differentiate_by='P'*, *at_constant='U'*)
Method to calculate and return the pressure derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial P}\right)_U$$

Returns

drho_dP_U [float] The pressure derivative of density of the phase at constant internal energy, [mol/m^3/Pa]

drho_dT()
Method to calculate and return the temperature derivative of molar density of the phase.

$$\frac{\partial \rho}{\partial T} = -\frac{1}{V^2}\left(\frac{\partial V}{\partial T}\right)_P$$

Returns

drho_dT [float] Temperature derivative of molar density, [mol/(K*m^3)]

drho_dT_A(*property='rho'*, *differentiate_by='T'*, *at_constant='A'*)
Method to calculate and return the temperature derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial T}\right)_A$$

Returns

drho_dT_A [float] The temperature derivative of density of the phase at constant Helmholtz energy, [mol/m^3/K]

drho_dT_G(*property='rho'*, *differentiate_by='T'*, *at_constant='G'*)
Method to calculate and return the temperature derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial T}\right)_G$$

Returns

drho_dT_G [float] The temperature derivative of density of the phase at constant Gibbs energy, [mol/m^3/K]

drho_dT_H(*property='rho'*, *differentiate_by='T'*, *at_constant='H'*)
Method to calculate and return the temperature derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial T}\right)_{H}$$

Returns

drho_dT_H [float] The temperature derivative of density of the phase at constant enthalpy, [mol/m^3/K]

drho_dT_S(*property='rho'*, *differentiate_by='T'*, *at_constant='S'*)
Method to calculate and return the temperature derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial T}\right)_{S}$$

Returns

drho_dT_S [float] The temperature derivative of density of the phase at constant entropy, [mol/m^3/K]

drho_dT_U(*property='rho'*, *differentiate_by='T'*, *at_constant='U'*)
Method to calculate and return the temperature derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial T}\right)_{U}$$

Returns

drho_dT_U [float] The temperature derivative of density of the phase at constant internal energy, [mol/m^3/K]

drho_dT_V()
Method to calculate and return the temperature derivative of molar density of the phase at constant volume.

$$\left(\frac{\partial \rho}{\partial T}\right)_{V} = 0$$

Returns

drho_dT_V [float] Temperature derivative of molar density of the phase at constant volume, [mol/(m^3*K)]

drho_dV_A(*property='rho'*, *differentiate_by='V'*, *at_constant='A'*)
Method to calculate and return the volume derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{A}$$

Returns

drho_dV_A [float] The volume derivative of density of the phase at constant Helmholtz energy, [mol/m^3/m^3/mol]

drho_dV_G(*property='rho'*, *differentiate_by='V'*, *at_constant='G'*)
Method to calculate and return the volume derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{G}$$

Returns

drho_dV_G [float] The volume derivative of density of the phase at constant Gibbs energy, [mol/m^3/m^3/mol]

drho_dV_H(*property='rho'*, *differentiate_by='V'*, *at_constant='H'*)
Method to calculate and return the volume derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{H}$$

Returns

drho_dV_H [float] The volume derivative of density of the phase at constant enthalpy, [mol/m^3/m^3/mol]

drho_dV_S(*property='rho'*, *differentiate_by='V'*, *at_constant='S'*)
Method to calculate and return the volume derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{S}$$

Returns

drho_dV_S [float] The volume derivative of density of the phase at constant entropy, [mol/m^3/m^3/mol]

drho_dV_T()
Method to calculate and return the volume derivative of molar density of the phase.

$$\frac{\partial \rho}{\partial V} = -\frac{1}{V^2}$$

Returns

drho_dV_T [float] Molar density derivative of volume, [mol^2/m^6]

drho_dV_U(*property='rho'*, *differentiate_by='V'*, *at_constant='U'*)
Method to calculate and return the volume derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial V}\right)_{U}$$

Returns

drho_dV_U [float] The volume derivative of density of the phase at constant internal energy, [mol/m^3/m^3/mol]

drho_drho_A(*property='rho'*, *differentiate_by='rho'*, *at_constant='A'*)
Method to calculate and return the density derivative of density of the phase at constant Helmholtz energy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{A}$$

Returns

drho_drho_A [float] The density derivative of density of the phase at constant Helmholtz energy, [mol/m^3/mol/m^3]

drho_drho_G(*property='rho'*, *differentiate_by='rho'*, *at_constant='G'*)
Method to calculate and return the density derivative of density of the phase at constant Gibbs energy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{G}$$

Returns

drho_drho_G [float] The density derivative of density of the phase at constant Gibbs energy, [mol/m^3/mol/m^3]

drho_drho_H(*property='rho'*, *differentiate_by='rho'*, *at_constant='H'*)
Method to calculate and return the density derivative of density of the phase at constant enthalpy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{H}$$

Returns

drho_drho_H [float] The density derivative of density of the phase at constant enthalpy, [mol/m^3/mol/m^3]

drho_drho_S(*property='rho'*, *differentiate_by='rho'*, *at_constant='S'*)
Method to calculate and return the density derivative of density of the phase at constant entropy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{S}$$

Returns

drho_drho_S [float] The density derivative of density of the phase at constant entropy, [mol/m^3/mol/m^3]

drho_drho_U(*property='rho'*, *differentiate_by='rho'*, *at_constant='U'*)
Method to calculate and return the density derivative of density of the phase at constant internal energy.

$$\left(\frac{\partial \rho}{\partial \rho}\right)_{U}$$

Returns

drho_drho_U [float] The density derivative of density of the phase at constant internal energy, [mol/m^3/mol/m^3]

drho_mass_dP()

Method to calculate the mass density derivative with respect to pressure, at constant temperature.

$$\left(\frac{\partial \rho}{\partial P}\right)_T = \frac{-\text{MW} \frac{\partial V_m}{\partial P}}{1000 V_m^2}$$

Returns

drho_mass_dP [float] Pressure derivative of mass density at constant temperature, [kg/m^3/Pa]

Notes

Requires *dV_dP*, *MW*, and *V*.

This expression is readily obtainable with SymTy:

```
>>> from sympy import *
>>> P, T, MW = symbols('P, T, MW')
>>> Vm = symbols('Vm', cls=Function)
>>> rho_mass = (Vm(P))**-1*MW/1000
>>> diff(rho_mass, P)
-MW*Derivative(Vm(P), P)/(1000*Vm(P)**2)
```

drho_mass_dT()

Method to calculate the mass density derivative with respect to temperature, at constant pressure.

$$\left(\frac{\partial \rho}{\partial T}\right)_P = \frac{-\text{MW} \frac{\partial V_m}{\partial T}}{1000 V_m^2}$$

Returns

drho_mass_dT [float] Temperature derivative of mass density at constant pressure, [kg/m^3/K]

Notes

Requires *dV_dT*, *MW*, and *V*.

This expression is readily obtainable with SymPy:

```
>>> from sympy import *
>>> T, P, MW = symbols('T, P, MW')
>>> Vm = symbols('Vm', cls=Function)
>>> rho_mass = (Vm(T))**-1*MW/1000
>>> diff(rho_mass, T)
-MW*Derivative(Vm(T), T)/(1000*Vm(T)**2)
```

dspeed_of_sound_dP_T()

Method to calculate the pressure derivative of speed of sound at constant temperature in molar units.

$$\left(\frac{\partial c}{\partial P}\right)_T = -\frac{\sqrt{-\frac{\operatorname{Cp}(P) V^2(P) \operatorname{dPdV_T}(P)}{\operatorname{Cv}(P)}}\left(-\frac{\operatorname{Cp}(P) V^2(P) \frac{d}{dP} \operatorname{dPdV_T}(P)}{2 \operatorname{Cv}(P)} - \frac{\operatorname{Cp}(P) V(P) \operatorname{dPdV_T}(P) \frac{d}{dP} V(P)}{\operatorname{Cv}(P)} + \frac{\operatorname{Cp}(P) V^2(P) \operatorname{dPdV_T}}{2 \operatorname{Cv}^2}\right.}{\operatorname{Cp}(P) V^2(P) \operatorname{dPdV_T}(P)}$$

Returns

dspeed_of_sound_dP_T [float] Pressure derivative of speed of sound at constant temperature, [m*kg^0.5/s/mol^0.5/Pa]

`dspeed_of_sound_dT_P()`

Method to calculate the temperature derivative of speed of sound at constant pressure in molar units.

$$\left(\frac{\partial c}{\partial T}\right)_P = -\frac{\sqrt{-\frac{\operatorname{Cp}(T)V^{2}(T)\operatorname{dPdV_T}(T)}{\operatorname{Cv}(T)}}\left(-\frac{\operatorname{Cp}(T)V^{2}(T)\frac{d}{dT}\operatorname{dPdV_T}(T)}{2\operatorname{Cv}(T)} - \frac{\operatorname{Cp}(T)V(T)\operatorname{dPdV_T}(T)\frac{d}{dT}V(T)}{\operatorname{Cv}(T)} + \frac{\operatorname{Cp}(T)V^{2}(T)\operatorname{dPdV_T}}{2\operatorname{Cv}^{2}(T}\right.}{\operatorname{Cp}(T)V^{2}(T)\operatorname{dPdV_T}(T)}$$

Returns

dspeed_of_sound_dT_P [float] Temperature derivative of speed of sound at constant pressure, [m*kg^0.5/s/mol^0.5/K]

Notes

Requires the temperature derivative of Cp and Cv both at constant pressure, as wel as the volume and temperature derivative of pressure, calculated at constant temperature and then pressure respectively. These can be tricky to obtain.

`property economic_statuses`

Status of each component in in relation to import and export from various regions, [-].

Returns

economic_statuses [list[dict]] Status of each component in in relation to import and export from various regions, [-].

`force_phase = None`

Attribute which can be set to a global Phase object to force the phases identification routines to label it a certain phase. Accepts values of ('g', 'l', 's').

`property formulas`

Formulas of each component, [-].

Returns

formulas [list[str]] Formulas of each component, [-].

`classmethod from_json(json_repr)`

Method to create a phase from a JSON serialization of another phase.

Parameters

json_repr [dict] JSON-friendly representation, [-]

Returns

phase [*Phase*] Newly created phase object from the json serialization, [-]

Notes

It is important that the input string be in the same format as that created by *Phase.as_json*.

fugacities()

Method to calculate and return the fugacities of the phase.

$$f_i = Pz_i \exp(\ln \phi_i)$$

Returns

fugacities [list[float]] Fugacities, [Pa]

fugacities_at_zs(zs)

Method to directly calculate the figacities at a different composition than the current phase. This is implemented to allow for the possibility of more direct calls to obtain fugacities than is possible with the phase interface. This base method simply creates a new phase, gets its log fugacity coefficients, exponentiates them, and multiplies them by *P* and compositions.

Returns

fugacities [list[float]] Fugacities, [Pa]

fugacities_lowest_Gibbs()

Method to calculate and return the fugacities of the phase.

$$f_i = Pz_i \exp(\ln \phi_i)$$

Returns

fugacities [list[float]] Fugacities, [Pa]

fugacity()

Method to calculate and return the fugacity of the phase; provided the phase is 1 component.

Returns

fugacity [list[float]] Fugacity, [Pa]

gammas()

Method to calculate and return the activity coefficients of the phase, [-].

Activity coefficients are defined as the ratio of the actual fugacity coefficients times the pressure to the reference pure fugacity coefficients times the reference pressure. The reference pressure can be set to the actual pressure (the Lewis Randall standard state) which makes the pressures cancel.

$$\gamma_i(T, P, x; f_i^0(T, P_i^0)) = \frac{\phi_i(T, P, x)P}{\phi_i^0(T, P_i^0)P_i^0}$$

Returns

gammas [list[float]] Activity coefficients, [-]

ideal_gas_basis = False

is_solid = False

isentropic_exponent()

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.

$$k = -\frac{V}{P}\frac{C_p}{C_v}\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

k_PV [float] Isentropic exponent of a real fluid, [-]

isentropic_exponent_PT()

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $P^{(1-k)}T^k = \text{const}$.

$$k = \frac{1}{1 - \frac{P}{C_p}\left(\frac{\partial V}{\partial T}\right)_P}$$

Returns

k_PT [float] Isentropic exponent of a real fluid, [-]

isentropic_exponent_PV()

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.

$$k = -\frac{V}{P}\frac{C_p}{C_v}\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

k_PV [float] Isentropic exponent of a real fluid, [-]

isentropic_exponent_TV()

Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $TV^{k-1} = \text{const}$.

$$k = 1 + \frac{V}{C_v}\left(\frac{\partial P}{\partial T}\right)_V$$

Returns

k_TV [float] Isentropic exponent of a real fluid, [-]

isobaric_expansion()

Method to calculate and return the isobatic expansion coefficient of the phase.

$$\beta = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P$$

Returns

beta [float] Isobaric coefficient of a thermal expansion, [1/K]

isothermal_bulk_modulus()

Method to calculate and return the isothermal bulk modulus of the phase.

$$K_T = -V\left(\frac{\partial P}{\partial V}\right)_T$$

Returns

isothermal_bulk_modulus [float] Isothermal bulk modulus, [Pa]

isothermal_compressibility()

Method to calculate and return the isothermal compressibility of the phase.

$$\kappa = -\frac{1}{V}\left(\frac{\partial V}{\partial P}\right)_T$$

Returns

kappa [float] Isothermal coefficient of compressibility, [1/Pa]

`kappa()`

Method to calculate and return the isothermal compressibility of the phase.

$$\kappa = -\frac{1}{V}\left(\frac{\partial V}{\partial P}\right)_T$$

Returns

kappa [float] Isothermal coefficient of compressibility, [1/Pa]

`property legal_statuses`

Status of each component in in relation to import and export rules from various regions, [-].

Returns

legal_statuses [list[dict]] Status of each component in in relation to import and export rules from various regions, [-].

`lnfugacities()`

Method to calculate and return the log of fugacities of the phase.

$$\ln f_i = \ln\left(P z_i \exp(\ln \phi_i)\right) = \ln(P) + \ln(z_i) + \ln \phi_i$$

Returns

lnfugacities [list[float]] Log fugacities, [log(Pa)]

`lnphi()`

Method to calculate and return the log of fugacity coefficient of the phase; provided the phase is 1 component.

Returns

lnphi [list[float]] Log fugacity coefficient, [-]

`lnphis()`

Method to calculate and return the log of fugacity coefficients of each component in the phase.

Returns

lnphis [list[float]] Log fugacity coefficients, [-]

`lnphis_G_min()`

Method to calculate and return the log fugacity coefficients of the phase. If the phase can have multiple solutions at its *T* and *P*, this method should return those with the lowest Gibbs energy. This needs to be implemented on phases with that criteria like cubic EOSs.

Returns

lnphis [list[float]] Log fugacity coefficients, [-]

`lnphis_at_zs(zs)`

Method to directly calculate the log fugacity coefficients at a different composition than the current phase. This is implemented to allow for the possibility of more direct calls to obtain fugacities than is possible with the phase interface. This base method simply creates a new phase, gets its log fugacity coefficients, and returns them.

Returns

lnphis [list[float]] Log fugacity coefficients, [-]

`property logPs`

Octanol-water partition coefficients for each component, [-].

Returns

logPs [list[float]] Octanol-water partition coefficients for each component, [-].

`log_zs()`

Method to calculate and return the log of mole fractions specified. These are used in calculating entropy and in many other formulas.

$$\ln z_i$$

Returns

log_zs [list[float]] Log of mole fractions, [-]

`model_hash(ignore_phase=False)`

Method to compute a hash of a phase.

Parameters

ignore_phase [bool] Whether or not to include the specifc class of the model in the hash

Returns

hash [int] Hash representing the settings of the phase; phases with all identical model parameters should have the same hash.

`molar_water_content()`

Method to calculate and return the molar water content; this is the g/mol of the fluid which is coming from water, [g/mol].

$$\text{water content} = \text{MW}_{H2O} w_{H2O}$$

Returns

molar_water_content [float] Molar water content, [g/mol]

`property molecular_diameters`

Lennard-Jones molecular diameters for each component, [angstrom].

Returns

molecular_diameters [list[float]] Lennard-Jones molecular diameters for each component, [angstrom].

`mu()`

`property names`

Names for each component, [-].

Returns

names [list[str]] Names for each component, [-].

`obj_references = ()`

Tuple of object instances which should be stored as json using their own as_json method.

`property omegas`

Acentric factors for each component, [-].

Returns

omegas [list[float]] Acentric factors for each component, [-].

`property phase_STPs`

Standard states ('g', 'l', or 's') for each component, [-].

Returns

phase_STPs [list[str]] Standard states ('g', 'l', or 's') for each component, [-].

phi()

Method to calculate and return the fugacity coefficient of the phase; provided the phase is 1 component.

Returns

phi [list[float]] Fugacity coefficient, [-]

phis()

Method to calculate and return the fugacity coefficients of the phase.

$$\phi_i = \exp(\ln \phi_i)$$

Returns

phis [list[float]] Fugacity coefficients, [-]

pointer_reference_dicts = ()

Tuple of dictionaries for string -> object

pointer_references = ()

Tuple of attributes which should be stored by converting them to a string, and then they will be looked up in their corresponding *pointer_reference_dicts* entry.

pseudo_Pc()

Method to calculate and return the pseudocritical pressure calculated using Kay's rule (linear mole fractions):

$$P_{c,pseudo} = \sum_i z_i P_{c,i}$$

Returns

pseudo_Pc [float] Pseudocritical pressure of the phase, [Pa]

pseudo_Tc()

Method to calculate and return the pseudocritical temperature calculated using Kay's rule (linear mole fractions):

$$T_{c,pseudo} = \sum_i z_i T_{c,i}$$

Returns

pseudo_Tc [float] Pseudocritical temperature of the phase, [K]

pseudo_Vc()

Method to calculate and return the pseudocritical volume calculated using Kay's rule (linear mole fractions):

$$V_{c,pseudo} = \sum_i z_i V_{c,i}$$

Returns

pseudo_Vc [float] Pseudocritical volume of the phase, [m^3/mol]

pseudo_Zc()

Method to calculate and return the pseudocritical compressibility calculated using Kay's rule (linear mole fractions):

$$Z_{c,pseudo} = \sum_i z_i Z_{c,i}$$

Returns

pseudo_Zc [float] Pseudocritical compressibility of the phase, [-]

pure_reference_types = ()
Tuple of types of *thermo.utils.TDependentProperty* or *thermo.utils.TPDependentProperty* corresponding to *pure_references*.

pure_references = ()
Tuple of attribute names which hold lists of *thermo.utils.TDependentProperty* or *thermo.utils.TPDependentProperty* instances.

reference_pointer_dicts = ()
Tuple of dictionaries for object -> string

rho()
Method to calculate and return the molar density of the phase.

$$\rho = frac1V$$

Returns

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

rho_mass()
Method to calculate and return mass density of the phase.

$$\rho = \frac{MW}{1000 \cdot VM}$$

Returns

rho_mass [float] Mass density, [kg/m^3]

rho_mass_liquid_ref()
Method to calculate and return the liquid reference mass density according to the temperature variable *T_liquid_volume_ref* of *thermo.bulk.BulkSettings* and the composition of the phase.

Returns

rho_mass_liquid_ref [float] Liquid mass density at the reference condition, [kg/m^3]

property rhocs
Molar densities at the critical point for each component, [mol/m^3].

Returns

rhocs [list[float]] Molar densities at the critical point for each component, [mol/m^3].

property rhocs_mass
Densities at the critical point for each component, [kg/m^3].

Returns

rhocs_mass [list[float]] Densities at the critical point for each component, [kg/m^3].

property rhog_STPs
Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

Returns

rhog_STPs [list[float]] Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

property rhog_STPs_mass
Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

Returns

rhog_STPs_mass [list[float]] Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

`property rhol_60Fs`
Liquid molar densities for each component at 60 °F, [mol/m^3].

Returns

rhol_60Fs [list[float]] Liquid molar densities for each component at 60 °F, [mol/m^3].

`property rhol_60Fs_mass`
Liquid mass densities for each component at 60 °F, [kg/m^3].

Returns

rhol_60Fs_mass [list[float]] Liquid mass densities for each component at 60 °F, [kg/m^3].

`property rhol_STPs`
Molar liquid densities at STP for each component, [mol/m^3].

Returns

rhol_STPs [list[float]] Molar liquid densities at STP for each component, [mol/m^3].

`property rhol_STPs_mass`
Liquid densities at STP for each component, [kg/m^3].

Returns

rhol_STPs_mass [list[float]] Liquid densities at STP for each component, [kg/m^3].

`property rhos_Tms`
Solid molar densities for each component at their respective melting points, [mol/m^3].

Returns

rhos_Tms [list[float]] Solid molar densities for each component at their respective melting points, [mol/m^3].

`property rhos_Tms_mass`
Solid mass densities for each component at their melting point, [kg/m^3].

Returns

rhos_Tms_mass [list[float]] Solid mass densities for each component at their melting point, [kg/m^3].

`scalar = True`

`sigma()`
Calculate and return the surface tension of the phase. For details of the implementation, see `SurfaceTensionMixture`.

This property is strictly the ideal-gas to liquid surface tension, not a true inter-phase property.

Returns

sigma [float] Surface tension, [N/m]

`property sigma_STPs`
Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

Returns

sigma_STPs [list[float]] Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

`property sigma_Tbs`

Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

Returns

sigma_Tbs [list[float]] Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

`property sigma_Tms`

Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

Returns

sigma_Tms [list[float]] Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

`property similarity_variables`

Similarity variables for each component, [mol/g].

Returns

similarity_variables [list[float]] Similarity variables for each component, [mol/g].

`property smiless`

SMILES identifiers for each component, [-].

Returns

smiless [list[str]] SMILES identifiers for each component, [-].

`property solubility_parameters`

Solubility parameters for each component at 298.15 K, [Pa^0.5].

Returns

solubility_parameters [list[float]] Solubility parameters for each component at 298.15 K, [Pa^0.5].

`speed_of_sound()`

Method to calculate and return the molar speed of sound of the phase.

$$w = \left[-V^2 \left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

A similar expression based on molar density is:

$$w = \left[\left(\frac{\partial P}{\partial \rho}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

Returns

w [float] Speed of sound for a real gas, [m*kg^0.5/(s*mol^0.5)]

`speed_of_sound_mass()`

Method to calculate and return the speed of sound of the phase.

$$w = \left[-V^2 \frac{1000}{MW}\left(\frac{\partial P}{\partial V}\right)_T \frac{C_p}{C_v}\right]^{1/2}$$

Returns

w [float] Speed of sound for a real gas, [m/s]

state_hash()

Basic method to calculate a hash of the state of the phase and its model parameters.

Note that the hashes should only be compared on the same system running in the same process!

Returns

state_hash [int] Hash of the object's model parameters and state, [-]

to(*zs*, *T=None*, *P=None*, *V=None*)

Method to create a new Phase object with the same constants as the existing Phase but at different conditions. Mole fractions *zs* are always required and any two of *T*, *P*, and *V* are required.

Parameters

zs [list[float]] Molar composition of the new phase, [-]

T [float, optional] Temperature of the new phase, [K]

P [float, optional] Pressure of the new phase, [Pa]

V [float, optional] Molar volume of the new phase, [m^3/mol]

Returns

new_phase [Phase] New phase at the specified conditions, [-]

Examples

These sample cases illustrate the three combinations of inputs. Note that some thermodynamic models may have multiple solutions for some inputs!

```
>>> from thermo import IdealGas
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21], HeatCapacityGases=[])
>>> phase.to(T=1e5, P=1e3, zs=[.5, .5])
IdealGas(HeatCapacityGases=[], T=100000.0, P=1000.0, zs=[0.5, 0.5])
>>> phase.to(V=1e-4, P=1e3, zs=[.1, .9])
IdealGas(HeatCapacityGases=[], T=0.012027235504, P=1000.0, zs=[0.1, 0.9])
>>> phase.to(T=1e5, V=1e12, zs=[.2, .8])
IdealGas(HeatCapacityGases=[], T=100000.0, P=8.31446261e-07, zs=[0.2, 0.8])
```

to_TP_zs(*T*, *P*, *zs*)

Method to create a new Phase object with the same constants as the existing Phase but at a different *T* and *P*.

Parameters

zs [list[float]] Molar composition of the new phase, [-]

T [float] Temperature of the new phase, [K]

P [float] Pressure of the new phase, [Pa]

Returns

new_phase [Phase] New phase at the specified conditions, [-]

Notes

This method is marginally faster than *Phase.to* as it does not need to check what the inputs are.

Examples

```
>>> from thermo import IdealGas
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21], HeatCapacityGases=[])
>>> phase.to_TP_zs(T=1e5, P=1e3, zs=[.5, .5])
IdealGas(HeatCapacityGases=[], T=100000.0, P=1000.0, zs=[0.5, 0.5])
```

value(*name*)
Method to retrieve a property from a string. This more or less wraps *getattr*.

name could be a python property like 'Tms' or a callable method like 'H'.

Parameters

name [str] String representing the property, [-]

Returns

value [various] Value specified, [various]

ws()
Method to calculate and return the mass fractions of the phase, [-]

Returns

ws [list[float]] Mass fractions, [-]

ws_no_water()
Method to calculate and return the mass fractions of all species in the phase, normalized to a water-free basis (the mass fraction of water returned is zero).

Returns

ws_no_water [list[float]] Mass fractions on a water free basis, [-]

zs_no_water()
Method to calculate and return the mole fractions of all species in the phase, normalized to a water-free basis (the mole fraction of water returned is zero).

Returns

zs_no_water [list[float]] Mole fractions on a water free basis, [-]

7.22.2 Ideal Gas Equation of State

class thermo.phases.**IdealGas**(*HeatCapacityGases=None*, *Hfs=None*, *Gfs=None*, *T=None*, *P=None*, *zs=None*)
Bases: *thermo.phases.phase.Phase*

Class for representing an ideal gas as a phase object. All departure properties are zero.

$$P = \frac{RT}{V}$$

Parameters

HeatCapacityGases [list[HeatCapacityGas]] Objects proiding pure-component heat capacity correlations, [-]

Hfs [list[float]] Molar ideal-gas standard heats of formation at 298.15 K and 1 atm, [J/mol]

Gfs [list[float]] Molar ideal-gas standard Gibbs energies of formation at 298.15 K and 1 atm, [J/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

zs [list[float], optional] Mole fractions of each component, [-]

Examples

T-P initialization for oxygen and nitrogen, using Poling's polynomial heat capacities:

```
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13, R*1.
↪57e-09, R*7e-08, R*-0.000261, R*3.539])),
...                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12, R*-6e-
↪09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21],␣
↪HeatCapacityGases=HeatCapacityGases)
>>> phase.Cp()
29.1733530
```

Methods

Cp()	Method to calculate and return the molar heat capacity of the phase.
H()	Method to calculate and return the enthalpy of the phase.
S()	Method to calculate and return the entropy of the phase.
d2H_dP2()	Method to calculate and return the second pressure derivative of molar enthalpy of the phase.
d2H_dT2()	Method to calculate and return the first temperature derivative of molar heat capacity of the phase.
d2P_dT2()	Method to calculate and return the second temperature derivative of pressure of the phase.
d2P_dTdV()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.
d2P_dV2()	Method to calculate and return the second volume derivative of pressure of the phase.
d2S_dP2()	Method to calculate and return the second pressure derivative of molar entropy of the phase.
dH_dP()	Method to calculate and return the first pressure derivative of molar enthalpy of the phase.
dH_dP_V()	Method to calculate and return the pressure derivative of molar enthalpy at constant volume of the phase.

continues on next page

Table 81 – continued from previous page

dH_dT_V()	Method to calculate and return the molar heat capacity of the phase.
dH_dV_P()	Method to calculate and return the volume derivative of molar enthalpy at constant pressure of the phase.
dH_dV_T()	Method to calculate and return the volume derivative of molar enthalpy at constant temperature of the phase.
dP_dT()	Method to calculate and return the first temperature derivative of pressure of the phase.
dP_dV()	Method to calculate and return the first volume derivative of pressure of the phase.
dS_dP()	Method to calculate and return the first pressure derivative of molar entropy of the phase.
dS_dP_V()	Method to calculate and return the first pressure derivative of molar entropy at constant volume of the phase.
dS_dT()	Method to calculate and return the first temperature derivative of molar entropy of the phase.
dS_dT_V()	Method to calculate and return the first temperature derivative of molar entropy at constant volume of the phase.
dlnphis_dP()	Method to calculate and return the pressure derivative of the log of fugacity coefficients of each component in the phase.
dlnphis_dT()	Method to calculate and return the temperature derivative of the log of fugacity coefficients of each component in the phase.
dphis_dP()	Method to calculate and return the pressure derivative of fugacity coefficients of each component in the phase.
dphis_dT()	Method to calculate and return the temperature derivative of fugacity coefficients of each component in the phase.
fugacities()	Method to calculate and return the fugacities of each component in the phase.
lnphis()	Method to calculate and return the log of fugacity coefficients of each component in the phase.
phis()	Method to calculate and return the fugacity coefficients of each component in the phase.

Cp()

Method to calculate and return the molar heat capacity of the phase.

$$C_p = \sum_i z_i C_{p,i}^{ig}$$

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

H()

Method to calculate and return the enthalpy of the phase.

$$H = \sum_i z_i H_{i}^{ig}$$

Returns

H [float] Molar enthalpy, [J/(mol)]

S()

Method to calculate and return the entropy of the phase.

$$S = \sum_i z_i S_i^{ig} - R\ln\left(\frac{P}{P_{ref}}\right) - R\sum_i z_i \ln(z_i)$$

Returns

S [float] Molar entropy, [J/(mol*K)]

__repr__()

Method to create a string representation of the phase object, with the goal of making it easy to obtain standalone code which reproduces the current state of the phase. This is extremely helpful in creating new test cases.

Returns

recreation [str] String which is valid Python and recreates the current state of the object if ran, [-]

Examples

```
>>> from thermo import HeatCapacityGas, IdealGas
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13,␣
↪R*1.57e-09, R*7e-08, R*-0.000261, R*3.539])),
...                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12,␣
↪R*-6e-09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21],␣
↪HeatCapacityGases=HeatCapacityGases)
>>> phase
IdealGas(HeatCapacityGases=[HeatCapacityGas(extrapolation="linear", method=
↪"POLY_FIT", poly_fit=(50.0, 1000.0, [-8.231317991971707e-12, 1.
↪3053706310500586e-08, 5.820123832707268e-07, -0.0021700747433379955, 29.
↪424883205644317])), HeatCapacityGas(extrapolation="linear", method="POLY_FIT",
↪ poly_fit=(50.0, 1000.0, [1.48828880864943e-11, -4.9886775708919434e-08, 5.
↪4709164027448316e-05, -0.014916145936966912, 30.18149930389626]))], T=300,␣
↪P=100000.0, zs=[0.79, 0.21])
```

d2H_dP2()

Method to calculate and return the second pressure derivative of molar enthalpy of the phase.

$$\frac{\partial^2 H}{\partial P^2} = 0$$

Returns

d2H_dP2 [float] Second pressure derivative of molar enthalpy, [J/(mol*Pa^2)]

d2H_dT2()

Method to calculate and return the first temperature derivative of molar heat capacity of the phase.

$$\frac{\partial C_p}{\partial T} = \sum_i z_i \frac{\partial C_{p,i}^{ig}}{\partial T}$$

Returns

d2H_dT2 [float] Second temperature derivative of enthalpy, [J/(mol*K^2)]

`d2P_dT2()`

Method to calculate and return the second temperature derivative of pressure of the phase.

$$\frac{\partial^2 P}{\partial T^2} = 0$$

Returns

d2P_dT2 [float] Second temperature derivative of pressure, [Pa/K^2]

`d2P_dTdV()`

Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.

$$\frac{\partial^2 P}{\partial V \partial T} = \frac{-P^2}{RT^2}$$

Returns

d2P_dTdV [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

`d2P_dV2()`

Method to calculate and return the second volume derivative of pressure of the phase.

$$\frac{\partial^2 P}{\partial V^2} = \frac{2P^3}{R^2T^2}$$

Returns

d2P_dV2 [float] Second volume derivative of pressure, [Pa*mol^2/m^6]

`d2S_dP2()`

Method to calculate and return the second pressure derivative of molar entropy of the phase.

$$\frac{\partial^2 S}{\partial P^2} = \frac{R}{P^2}$$

Returns

d2S_dP2 [float] Second pressure derivative of molar entropy, [J/(mol*K*Pa^2)]

`dH_dP()`

Method to calculate and return the first pressure derivative of molar enthalpy of the phase.

$$\frac{\partial H}{\partial P} = 0$$

Returns

dH_dP [float] First pressure derivative of molar enthalpy, [J/(mol*Pa)]

`dH_dP_V()`

Method to calculate and return the pressure derivative of molar enthalpy at constant volume of the phase.

$$\left(\frac{\partial H}{\partial P}\right)_V = C_p \left(\frac{\partial T}{\partial P}\right)_V$$

Returns

dH_dP_V [float] First pressure derivative of molar enthalpy at constant volume, [J/(mol*Pa)]

dH_dT_V()
Method to calculate and return the molar heat capacity of the phase.

$$C_p = \sum_i z_i C_{p,i}^{ig}$$

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

dH_dV_P()
Method to calculate and return the volume derivative of molar enthalpy at constant pressure of the phase.

$$\left(\frac{\partial H}{\partial V}\right)_P = C_p \left(\frac{\partial T}{\partial V}\right)_P$$

Returns

dH_dV_T [float] First pressure derivative of molar enthalpy at constant volume, [J/(m^3)]

dH_dV_T()
Method to calculate and return the volume derivative of molar enthalpy at constant temperature of the phase.

$$\left(\frac{\partial H}{\partial V}\right)_T = 0$$

Returns

dH_dV_T [float] First pressure derivative of molar enthalpy at constant volume, [J/(m^3)]

dP_dT()
Method to calculate and return the first temperature derivative of pressure of the phase.

$$\frac{\partial P}{\partial T} = \frac{P}{T}$$

Returns

dP_dT [float] First temperature derivative of pressure, [Pa/K]

dP_dV()
Method to calculate and return the first volume derivative of pressure of the phase.

$$\frac{\partial P}{\partial V} = \frac{-P^2}{RT}$$

Returns

dP_dV [float] First volume derivative of pressure, [Pa*mol/m^3]

dS_dP()
Method to calculate and return the first pressure derivative of molar entropy of the phase.

$$\frac{\partial S}{\partial P} = -\frac{R}{P}$$

Returns

dS_dP [float] First pressure derivative of molar entropy, [J/(mol*K*Pa)]

dS_dP_V()
Method to calculate and return the first pressure derivative of molar entropy at constant volume of the phase.

$$\left(\frac{\partial S}{\partial P}\right)_V = \frac{-R}{P} + \frac{C_p}{T}\frac{\partial T}{\partial P}$$

Returns

dS_dP_V [float] First pressure derivative of molar entropy at constant volume, [J/(mol*K*Pa)]

`dS_dT()`

Method to calculate and return the first temperature derivative of molar entropy of the phase.

$$\frac{\partial S}{\partial T} = \frac{C_p}{T}$$

Returns

dS_dT [float] First temperature derivative of molar entropy, [J/(mol*K^2)]

`dS_dT_V()`

Method to calculate and return the first temperature derivative of molar entropy at constant volume of the phase.

$$\left(\frac{\partial S}{\partial T}\right)_V = \frac{C_p}{T} - \frac{R}{P}\frac{\partial P}{\partial T}$$

Returns

dS_dT_V [float] First temperature derivative of molar entropy at constant volume, [J/(mol*K^2)]

`dlnphis_dP()`

Method to calculate and return the pressure derivative of the log of fugacity coefficients of each component in the phase.

$$\frac{\partial \ln \phi_i}{\partial P} = 0$$

Returns

dlnphis_dP [list[float]] Log fugacity coefficients, [1/Pa]

`dlnphis_dT()`

Method to calculate and return the temperature derivative of the log of fugacity coefficients of each component in the phase.

$$\frac{\partial \ln \phi_i}{\partial T} = 0$$

Returns

dlnphis_dT [list[float]] Log fugacity coefficients, [1/K]

`dphis_dP()`

Method to calculate and return the pressure derivative of fugacity coefficients of each component in the phase.

$$\frac{\partial \phi_i}{\partial P} = 0$$

Returns

dphis_dP [list[float]] Pressure derivative of fugacity fugacity coefficients, [1/Pa]

`dphis_dT()`

Method to calculate and return the temperature derivative of fugacity coefficients of each component in the phase.

$$\frac{\partial \phi_i}{\partial T} = 0$$

Returns

dphis_dT [list[float]] Temperature derivative of fugacity fugacity coefficients, [1/K]

fugacities()

Method to calculate and return the fugacities of each component in the phase.

$$\text{fugacitiy}_i = z_i P$$

Returns

fugacities [list[float]] Fugacities, [Pa]

Examples

```
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13,␣
↪R*1.57e-09, R*7e-08, R*-0.000261, R*3.539])),
...                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12,␣
↪R*-6e-09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21],␣
↪HeatCapacityGases=HeatCapacityGases)
>>> phase.fugacities()
[79000.0, 21000.0]
```

lnphis()

Method to calculate and return the log of fugacity coefficients of each component in the phase.

$$\ln \phi_i = 0.0$$

Returns

lnphis [list[float]] Log fugacity coefficients, [-]

phis()

Method to calculate and return the fugacity coefficients of each component in the phase.

$$\phi_i = 1$$

Returns

phis [list[float]] Fugacity fugacity coefficients, [-]

7.22.3 Cubic Equations of State

Gas Phases

class thermo.phases.**CEOSGas**(*eos_class*, *eos_kwargs*, *HeatCapacityGases=None*, *Hfs=None*, *Gfs=None*, *Sfs=None*, *T=None*, *P=None*, *zs=None*)

Bases: `thermo.phases.phase.Phase`

Class for representing a cubic equation of state gas phase as a phase object. All departure properties are actually calculated by the code in `thermo.eos` and `thermo.eos_mix`.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V^2 + \delta V + \epsilon}$$

Parameters

eos_class [*thermo.eos_mix.GCEOSMIX*] EOS class, [-]

eos_kwargs [dict] Parameters to be passed to the created EOS, [-]

HeatCapacityGases [list[HeatCapacityGas]] Objects proiding pure-component heat capacity correlations, [-]

Hfs [list[float]] Molar ideal-gas standard heats of formation at 298.15 K and 1 atm, [J/mol]

Gfs [list[float]] Molar ideal-gas standard Gibbs energies of formation at 298.15 K and 1 atm, [J/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

zs [list[float], optional] Mole fractions of each component, [-]

Examples

T-P initialization for oxygen and nitrogen with the PR EOS, using Poling's polynomial heat capacities:

```
>>> from thermo import HeatCapacityGas, PRMIX, CEOSGas
>>> eos_kwargs = dict(Tcs=[154.58, 126.2], Pcs=[5042945.25, 3394387.5], omegas=[0.
↪021, 0.04], kijs=[[0.0, -0.0159], [-0.0159, 0.0]])
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13, R*1.
↪57e-09, R*7e-08, R*-0.000261, R*3.539])),
...                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12, R*-6e-
↪09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = CEOSGas(eos_class=PRMIX, eos_kwargs=eos_kwargs, T=300, P=1e5, zs=[.79, .
↪21], HeatCapacityGases=HeatCapacityGases)
>>> phase.Cp()
29.2285050
```

Methods

Cp()	Method to calculate and return the constant-pressure heat capacity of the phase.
Cv()	Method to calculate and return the constant-volume heat capacity *Cv* of the phase.
H()	Method to calculate and return the enthalpy of the phase.
S()	Method to calculate and return the entropy of the phase.
V_iter([force])	Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure.
d2P_dT2()	Method to calculate and return the second temperature derivative of pressure of the phase.
d2P_dTdV()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.
d2P_dV2()	Method to calculate and return the second volume derivative of pressure of the phase.

continues on next page

Table 82 – continued from previous page

dP_dT()	Method to calculate and return the first temperature derivative of pressure of the phase.
dP_dV()	Method to calculate and return the first volume derivative of pressure of the phase.
dS_dT_V()	Method to calculate and return the first temperature derivative of molar entropy at constant volume of the phase.
dlnphis_dP()	Method to calculate and return the first pressure derivative of the log of fugacity coefficients of each component in the phase.
dlnphis_dT()	Method to calculate and return the first temperature derivative of the log of fugacity coefficients of each component in the phase.
lnphis()	Method to calculate and return the log of fugacity coefficients of each component in the phase.
to_TP_zs(T, P, zs[, other_eos])	Method to create a new Phase object with the same constants as the existing Phase but at a different T and P.

Cp()

Method to calculate and return the constant-pressure heat capacity of the phase.

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

Cv()

Method to calculate and return the constant-volume heat capacity *Cv* of the phase.

$$C_v = T\left(\frac{\partial P}{\partial T}\right)_V^2 / \left(\frac{\partial P}{\partial V}\right)_T + Cp$$

Returns

Cv [float] Constant volume molar heat capacity, [J/(mol*K)]

H()

Method to calculate and return the enthalpy of the phase. The reference state for most subclasses is an ideal-gas enthalpy of zero at 298.15 K and 101325 Pa.

Returns

H [float] Molar enthalpy, [J/(mol)]

S()

Method to calculate and return the entropy of the phase. The reference state for most subclasses is an ideal-gas entropy of zero at 298.15 K and 101325 Pa.

Returns

S [float] Molar entropy, [J/(mol*K)]

V_iter(*force=False*)

Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure. This often means the return value of this method is an mpmath *mpf*. This dummy method simply returns the implemented V method.

Returns

V [float or mpf] Molar volume, [m^3/mol]

__repr__()

Method to create a string representation of the phase object, with the goal of making it easy to obtain standalone code which reproduces the current state of the phase. This is extremely helpful in creating new test cases.

Returns

recreation [str] String which is valid Python and recreates the current state of the object if ran, [-]

Examples

```
>>> from thermo import HeatCapacityGas, PRMIX, CEOSGas
>>> eos_kwargs = dict(Tcs=[154.58, 126.2], Pcs=[5042945.25, 3394387.5],␣
→omegas=[0.021, 0.04], kijs=[[0.0, -0.0159], [-0.0159, 0.0]])
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13,␣
→R*1.57e-09, R*7e-08, R*-0.000261, R*3.539])),
...                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12,␣
→R*-6e-09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = CEOSGas(eos_class=PRMIX, eos_kwargs=eos_kwargs, T=300, P=1e5, zs=[.
→79, .21], HeatCapacityGases=HeatCapacityGases)
>>> phase
CEOSGas(eos_class=PRMIX, eos_kwargs={"Tcs": [154.58, 126.2], "Pcs": [5042945.25,
→ 3394387.5], "omegas": [0.021, 0.04], "kijs": [[0.0, -0.0159], [-0.0159, 0.
→0]]}, HeatCapacityGases=[HeatCapacityGas(extrapolation="linear", method="POLY_
→FIT", poly_fit=(50.0, 1000.0, [-8.231317991971707e-12, 1.3053706310500586e-08,
→ 5.820123832707268e-07, -0.0021700747433379955, 29.424883205644317])),␣
→HeatCapacityGas(extrapolation="linear", method="POLY_FIT", poly_fit=(50.0,␣
→1000.0, [1.48828880864943e-11, -4.9886775708919434e-08, 5.4709164027448316e-
→05, -0.014916145936966912, 30.18149930389626]))], T=300, P=100000.0, zs=[0.79,
→ 0.21])
```

d2P_dT2()

Method to calculate and return the second temperature derivative of pressure of the phase.

$$\left(\frac{\partial^2 P}{\partial T^2}\right)_V = -\frac{a\frac{d^2\alpha(T)}{dT^2}}{V^2 + V\delta + \epsilon}$$

Returns

d2P_dT2 [float] Second temperature derivative of pressure, [Pa/K^2]

d2P_dTdV()

Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.

$$\left(\frac{\partial^2 P}{\partial T \partial V}\right) = -\frac{R}{(V-b)^2} + \frac{a(2V+\delta)\frac{d\alpha(T)}{dT}}{(V^2+V\delta+\epsilon)^2}$$

Returns

d2P_dTdV [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

d2P_dV2()

Method to calculate and return the second volume derivative of pressure of the phase.

$$\left(\frac{\partial^2 P}{\partial V^2}\right)_T = 2\left(\frac{RT}{(V-b)^3} - \frac{a(2V+\delta)^2\alpha(T)}{(V^2+V\delta+\epsilon)^3} + \frac{a\alpha(T)}{(V^2+V\delta+\epsilon)^2}\right)$$

Returns

d2P_dV2 [float] Second volume derivative of pressure, [Pa*mol^2/m^6]

dP_dT()

Method to calculate and return the first temperature derivative of pressure of the phase.

$$\left(\frac{\partial P}{\partial T}\right)_V = \frac{R}{V-b} - \frac{a\frac{d\alpha(T)}{dT}}{V^2+V\delta+\epsilon}$$

Returns

dP_dT [float] First temperature derivative of pressure, [Pa/K]

dP_dV()

Method to calculate and return the first volume derivative of pressure of the phase.

$$\left(\frac{\partial P}{\partial V}\right)_T = -\frac{RT}{(V-b)^2} - \frac{a\left(-2V-\delta\right)\alpha(T)}{\left(V^2+V\delta+\epsilon\right)^2}$$

Returns

dP_dV [float] First volume derivative of pressure, [Pa*mol/m^3]

dS_dT_V()

Method to calculate and return the first temperature derivative of molar entropy at constant volume of the phase.

$$\left(\frac{\partial S}{\partial T}\right)_V = \frac{C_p^{ig}}{T} - \frac{R}{P}\frac{\partial P}{\partial T} + \left(\frac{\partial S_{dep}}{\partial T}\right)_V$$

Returns

dS_dT_V [float] First temperature derivative of molar entropy at constant volume, [J/(mol*K^2)]

dlnphis_dP()

Method to calculate and return the first pressure derivative of the log of fugacity coefficients of each component in the phase. The calculation is performed by *thermo.eos_mix.GCEOSMIX.dlnphis_dP* or a simpler formula in the case of most specific models.

Returns

dlnphis_dP [list[float]] First pressure derivative of log fugacity coefficients, [1/Pa]

dlnphis_dT()

Method to calculate and return the first temperature derivative of the log of fugacity coefficients of each component in the phase. The calculation is performed by *thermo.eos_mix.GCEOSMIX.dlnphis_dT* or a simpler formula in the case of most specific models.

Returns

dlnphis_dT [list[float]] First temperature derivative of log fugacity coefficients, [1/K]

lnphis()

Method to calculate and return the log of fugacity coefficients of each component in the phase. The calculation is performed by *thermo.eos_mix.GCEOSMIX.fugacity_coefficients* or a simpler formula in the case of most specific models.

Returns

lnphis [list[float]] Log fugacity coefficients, [-]

to_TP_zs(*T*, *P*, *zs*, *other_eos=None*)

Method to create a new Phase object with the same constants as the existing Phase but at a different T and P. This method has a special parameter *other_eos*.

This is added to allow a gas-type phase to be created from a liquid-type phase at the same conditions (and vice-versa), as `GCEOSMIX` objects were designed to have vapor and liquid properties in the same phase. This argument is mostly for internal use.

Parameters

zs [list[float]] Molar composition of the new phase, [-]

T [float] Temperature of the new phase, [K]

P [float] Pressure of the new phase, [Pa]

other_eos [obj:*GCEOSMIX <thermo.eos_mix.GCEOSMIX>* object] Other equation of state object at the same conditions, [-]

Returns

new_phase [Phase] New phase at the specified conditions, [-]

Notes

This method is marginally faster than `Phase.to` as it does not need to check what the inputs are.

Examples

```
>>> from thermo.eos_mix import PRMIX
>>> eos_kwargs = dict(Tcs=[305.32, 369.83], Pcs=[4872000.0, 4248000.0],␣
↪omegas=[0.098, 0.152])
>>> gas = CEOSGas(PRMIX, T=300.0, P=1e6, zs=[.2, .8], eos_kwargs=eos_kwargs)
>>> liquid = CEOSLiquid(PRMIX, T=500.0, P=1e7, zs=[.3, .7], eos_kwargs=eos_
↪kwargs)
>>> new_liq = liquid.to_TP_zs(T=gas.T, P=gas.P, zs=gas.zs, other_eos=gas.eos_
↪mix)
>>> new_liq
CEOSLiquid(eos_class=PRMIX, eos_kwargs={"Tcs": [305.32, 369.83], "Pcs":␣
↪[4872000.0, 4248000.0], "omegas": [0.098, 0.152]}, HeatCapacityGases=[],␣
↪T=300.0, P=1000000.0, zs=[0.2, 0.8])
>>> new_liq.eos_mix is gas.eos_mix
True
```

Liquid Phases

class thermo.phases.**CEOSLiquid**(*eos_class*, *eos_kwargs*, *HeatCapacityGases=None*, *Hfs=None*, *Gfs=None*, *Sfs=None*, *T=None*, *P=None*, *zs=None*)

Bases: `thermo.phases.phase.Phase`

Class for representing a cubic equation of state gas phase as a phase object. All departure properties are actually calculated by the code in `thermo.eos` and `thermo.eos_mix`.

$$P = \frac{RT}{V-b} - \frac{a\alpha(T)}{V^2 + \delta V + \epsilon}$$

Parameters

eos_class [*thermo.eos_mix.GCEOSMIX*] EOS class, [-]

eos_kwargs [dict] Parameters to be passed to the created EOS, [-]

HeatCapacityGases [list[HeatCapacityGas]] Objects proiding pure-component heat capacity correlations, [-]

Hfs [list[float]] Molar ideal-gas standard heats of formation at 298.15 K and 1 atm, [J/mol]

Gfs [list[float]] Molar ideal-gas standard Gibbs energies of formation at 298.15 K and 1 atm, [J/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

zs [list[float], optional] Mole fractions of each component, [-]

Examples

T-P initialization for oxygen and nitrogen with the PR EOS, using Poling's polynomial heat capacities:

```
>>> from thermo import HeatCapacityGas, PRMIX, CEOSLiquid
>>> eos_kwargs = dict(Tcs=[154.58, 126.2], Pcs=[5042945.25, 3394387.5], omegas=[0.
↪021, 0.04], kijs=[[0.0, -0.0159], [-0.0159, 0.0]])
>>> HeatCapacityGases = [HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*-9.9e-13, R*1.
↪57e-09, R*7e-08, R*-0.000261, R*3.539])),
...                      HeatCapacityGas(poly_fit=(50.0, 1000.0, [R*1.79e-12, R*-6e-
↪09, R*6.58e-06, R*-0.001794, R*3.63]))]
>>> phase = CEOSLiquid(eos_class=PRMIX, eos_kwargs=eos_kwargs, T=300, P=1e5, zs=[.
↪79, .21], HeatCapacityGases=HeatCapacityGases)
>>> phase.Cp()
29.2285050
```

7.22.4 Activity Based Liquids

class thermo.phases.**GibbsExcessLiquid**(*VaporPressures*, *VolumeLiquids=None*, *HeatCapacityGases=None*, *GibbsExcessModel=None*, *eos_pure_instances=None*, *EnthalpyVaporizations=None*, *HeatCapacityLiquids=None*, *VolumeSupercriticalLiquids=None*, *use_Hvap_caloric=False*, *use_Poynting=False*, *use_phis_sat=False*, *use_Tait=False*, *use_eos_volume=False*, *Hfs=None*, *Gfs=None*, *Sfs=None*, *henry_components=None*, *henry_data=None*, *T=None*, *P=None*, *zs=None*, *Psat_extrpolation='AB'*, *equilibrium_basis=None*, *caloric_basis=None*)

Bases: *thermo.phases.phase.Phase*

Phase based on combining Raoult's law with a *GibbsExcess* model, optionally including saturation fugacity coefficient corrections (if the vapor phase is a cubic equation of state) and Poynting correction factors (if more accuracy is desired).

The equilibrium equation options (controlled by *equilibrium_basis*) are as follows:

- 'Psat': $\phi_i = \frac{\gamma_i P_{i}^{sat}}{P}$
- 'Poynting&PhiSat': $\phi_i = \frac{\gamma_i P_{i}^{sat} \phi_i^{sat} \text{Poynting}_i}{P}$

- 'Poynting': $\phi_i = \frac{\gamma_i P_{i}^{sat} \text{Poynting}_i}{P}$
- 'PhiSat': $\phi_i = \frac{\gamma_i P_{i}^{sat} \phi_i^{sat}}{P}$

In all cases, the activity coefficient is derived from the `GibbsExcess` model specified as input; use the `IdealSolution` class as an input to set the activity coefficients to one.

The enthalpy H and entropy S (and other caloric properties U, G, A) equation options are similar to the equilibrium ones. If the same option is selected for *equilibrium_basis* and *caloric_basis*, the phase will be *thermodynamically consistent*. This is recommended for many reasons. The full 'Poynting&PhiSat' equations for H and S are as follows; see `GibbsExcessLiquid.H` and `GibbsExcessLiquid.S` for all of the other equations:

$$H = H_{\text{excess}} + \sum_i z_i \left[-RT^2\left(\frac{\frac{\partial \phi_{\text{sat},i}}{\partial T}}{\phi_{\text{sat},i}} + \frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + \frac{\frac{\text{Poynting}}{\partial T}}{\text{Poynting}} \right) + \int_{T,ref}^T C_{p,ig} dT \right]$$

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i \left[R\left(T \frac{\frac{\partial \phi_{\text{sat},i}}{\partial T}}{\phi_{\text{sat},i}} + T\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + T\frac{\frac{\text{Poynting}}{\partial T}}{\text{Poynting}} + \ln(P_{\text{sat},i}) + \ln\left(\frac{\text{Poynting}\cdot\phi_{\text{sat},i}}{P}\right.\right.\right.$$

An additional caloric mode is *Hvap*, which uses enthalpy of vaporization; this mode can never be thermodynamically consistent, but is still widely used.

$$H = H_{\text{excess}} + \sum_i z_i \left[-H_{vap,i} + \int_{T,ref}^T C_{p,ig} dT \right]$$

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i \left[R\left(\ln P_{\text{sat},i} + \ln\left(\frac{1}{P}\right)\right) + \frac{H_{vap,i}}{T} - \int_{T,ref}^T \frac{C_{p,ig,i}}{T} dT \right]$$

Warning: Note that above the critical point, there is no definition for what vapor pressure is. The vapor pressure also tends to reach zero at temperatures in the 4-20 K range. These aspects mean extrapolation in the supercritical and very low temperature region is critical to ensure the equations will still converge. Extrapolation can be performed using either the equation $P^{\text{sat}} = \exp\left(A - \frac{B}{T}\right)$ or $P^{\text{sat}} = \exp\left(A + \frac{B}{T} + C\cdot \ln T\right)$ by setting *Psat_extrpolation* to either 'AB' or 'ABC' respectively. The extremely low temperature region's issue is solved by calculating the logarithm of vapor pressures instead of the actual value. While floating point values in Python (doubles) can reach a minimum value of around 1e-308, if only the logarithm of that number is computed no issues arise. Both of these features only work when the vapor pressure correlations are polynomials.

Warning: When using 'PhiSat' as an option, note that the factor cannot be calculated when a compound is supercritical, as there is no longer any vapor-liquid pure-component equilibrium (by definition).

Parameters

VaporPressures [list[`thermo.vapor_pressure.VaporPressure`]] Objects holding vapor pressure data and methods, [-]

VolumeLiquids [list[`thermo.volume.VolumeLiquid`], optional] Objects holding liquid volume data and methods; required for Poynting factors and volumetric properties, [-]

HeatCapacityGases [list[`thermo.heat_capacity.HeatCapacityGas`], optional] Objects proiding pure-component heat capacity correlations; required for caloric properties, [-]

GibbsExcessModel [`GibbsExcess`, optional] Configured instance for calculating activity coefficients and excess properties; set to `IdealSolution` if not provided, [-]

eos_pure_instances [list[*thermo.eos.GCEOS*], optional] Cubic equation of state object instances for each pure component, [-]

EnthalpyVaporizations [list[*thermo.phase_change.EnthalpyVaporization*], optional] Objects holding enthalpy of vaporization data and methods; used only with the 'Hvap' optional, [-]

HeatCapacityLiquids [list[*thermo.heat_capacity.HeatCapacityLiquid*], optional] Objects holding liquid heat capacity data and methods; not used at present, [-]

VolumeSupercriticalLiquids [list[*thermo.volume.VolumeLiquid*], optional] Objects holding liquid volume data and methods but that are used for supercritical temperatures on a per-component basis only; required for Poynting factors and volumetric properties at supercritical conditions; *VolumeLiquids* is used if not provided, [-]

Hfs [list[float], optional] Molar ideal-gas standard heats of formation at 298.15 K and 1 atm, [J/mol]

Gfs [list[float], optional] Molar ideal-gas standard Gibbs energies of formation at 298.15 K and 1 atm, [J/mol]

T [float, optional] Temperature, [K]

P [float, optional] Pressure, [Pa]

zs [list[float], optional] Mole fractions of each component, [-]

equilibrium_basis [str, optional] Which set of equilibrium equations to use when calculating fugacities and related properties; valid options are 'Psat', 'Poynting&PhiSat', 'Poynting', 'PhiSat', [-]

caloric_basis [str, optional] Which set of caloric equations to use when calculating fugacities and related properties; valid options are 'Psat', 'Poynting&PhiSat', 'Poynting', 'PhiSat', 'Hvap' [-]

Psat_extrpolation [str, optional] One of 'AB' or 'ABC'; configures extrapolation for vapor pressure, [-]

use_Hvap_caloric [bool, optional] If True, enthalpy and entropy will be calculated using ideal-gas heat capacity and the heat of vaporization of the fluid only. This forces enthalpy to be pressure-independent. This supersedes other options which would otherwise impact these properties. The molar volume of the fluid has no impact on enthalpy or entropy if this option is True. This option is not thermodynamically consistent, but is still often an assumption that is made.

Methods

Cp()	Method to calculate and return the constant-pressure heat capacity of the phase.
H()	Method to calculate the enthalpy of the *GibbsExcessLiquid* phase.
Poyntings()	Method to calculate and return the Poynting pressure correction factors of the phase, [-].
S()	Method to calculate the entropy of the *GibbsExcessLiquid* phase.
gammas()	Method to calculate and return the activity coefficients of the phase, [-].

continues on next page

Table 83 – continued from previous page

phis_sat()	Method to calculate and return the saturation fugacity coefficient correction factors of the phase, [-].

Cp()
Method to calculate and return the constant-pressure heat capacity of the phase.

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

H()
Method to calculate the enthalpy of the GibbsExcessLiquid phase. Depending on the settings of the phase, this can include the effects of activity coefficients gammas, pressure correction terms Poyntings, and pure component saturation fugacities phis_sat as well as the pure component vapor pressures.

When *caloric_basis* is 'Poynting&PhiSat':

$$H = H_{\text{excess}} + \sum_i z_i \left[-RT^2\left(\frac{\frac{\partial \phi_{\text{sat},i}}{\partial T}}{\phi_{\text{sat},i}} + \frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + \frac{\frac{\text{Poynting}}{\partial T}}{\text{Poynting}}\right) + \int_{T,ref}^T C_{p,ig} dT\right]$$

When *caloric_basis* is 'PhiSat':

$$H = H_{\text{excess}} + \sum_i z_i \left[-RT^2\left(\frac{\frac{\partial \phi_{\text{sat},i}}{\partial T}}{\phi_{\text{sat},i}} + \frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}}\right) + \int_{T,ref}^T C_{p,ig} dT\right]$$

When *caloric_basis* is 'Poynting':

$$H = H_{\text{excess}} + \sum_i z_i \left[-RT^2\left(+\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + \frac{\frac{\text{Poynting}}{\partial T}}{\text{Poynting}}\right) + \int_{T,ref}^T C_{p,ig} dT\right]$$

When *caloric_basis* is 'Psat':

$$H = H_{\text{excess}} + \sum_i z_i \left[-RT^2\left(+\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}}\right) + \int_{T,ref}^T C_{p,ig} dT\right]$$

When *caloric_basis* is 'Hvap':

$$H = H_{\text{excess}} + \sum_i z_i \left[-H_{vap,i} + \int_{T,ref}^T C_{p,ig} dT\right]$$

Returns

H [float] Enthalpy of the phase, [J/(mol)]

Poyntings()
Method to calculate and return the Poynting pressure correction factors of the phase, [-].

$$\text{Poynting}_i = \exp\left(\frac{V_{m,i}(P - P_{sat})}{RT}\right)$$

Returns

Poyntings [list[float]] Poynting pressure correction factors, [-]

Notes

The above formula is correct for pressure-independent molar volumes. When the volume does depend on pressure, the full expression is:

$$\text{Poynting} = \exp\left[\frac{\int_{P_i^{sat}}^P V_i^l dP}{RT}\right]$$

When a specified model e.g. the Tait equation is used, an analytical integral of this term is normally available.

S()

Method to calculate the entropy of the *GibbsExcessLiquid* phase. Depending on the settings of the phase, this can include the effects of activity coefficients *gammas*, pressure correction terms *Poyntings*, and pure component saturation fugacities *phis_sat* as well as the pure component vapor pressures.

When *caloric_basis* is 'Poynting&PhiSat':

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i\left[R\left(T\frac{\frac{\partial \phi_{\text{sat},i}}{\partial T}}{\phi_{\text{sat},i}} + T\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + T\frac{\frac{\text{Poynting}}{\partial T}}{\text{Poynting}} + \ln(P_{\text{sat},i}) + \ln\left(\frac{\text{Poynting}\cdot}{P}\right.\right.\right.$$

When *caloric_basis* is 'PhiSat':

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i\left[R\left(T\frac{\frac{\partial \phi_{\text{sat},i}}{\partial T}}{\phi_{\text{sat},i}} + T\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + \ln(P_{\text{sat},i}) + \ln\left(\frac{\phi_{\text{sat},i}}{P}\right)\right) - \int_{T,ref}^T \frac{C_{p,ig,}}{T}\right.$$

When *caloric_basis* is 'Poynting':

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i\left[R\left(T\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + T\frac{\frac{\text{Poynting}}{\partial T}}{\text{Poynting}} + \ln(P_{\text{sat},i}) + \ln\left(\frac{\text{Poynting}}{P}\right)\right) - \int_{T,re}^T\right.$$

When *caloric_basis* is 'Psat':

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i\left[R\left(T\frac{\frac{\partial P_{\text{sat},i}}{\partial T}}{P_{\text{sat},i}} + \ln(P_{\text{sat},i}) + \ln\left(\frac{1}{P}\right)\right) - \int_{T,ref}^T \frac{C_{p,ig,i}}{T}dT\right]$$

When *caloric_basis* is 'Hvap':

$$S = S_{\text{excess}} - R\sum_i z_i \ln z_i - R\ln\left(\frac{P}{P_{ref}}\right) - \sum_i z_i\left[R\left(\ln P_{\text{sat},i} + \ln\left(\frac{1}{P}\right)\right) + \frac{H_{vap,i}}{T} - \int_{T,ref}^T \frac{C_{p,ig,i}}{T}dT\right]$$

Returns

S [float] Entropy of the phase, [J/(mol*K)]

gammas()

Method to calculate and return the activity coefficients of the phase, [-]. This is a direct call to *GibbsExcess.gammas*.

Returns

gammas [list[float]] Activity coefficients, [-]

phis_sat()

Method to calculate and return the saturation fugacity coefficient correction factors of the phase, [-].

These are calculated from the provided pure-component equations of state. This term should only be used with a consistent vapor-phase cubic equation of state.

Returns

phis_sat [list[float]] Saturation fugacity coefficient correction factors, [-]

Notes

> **Warning:** This factor cannot be calculated when a compound is supercritical, as there is no longer any vapor-liquid pure-component equilibrium (by definition).

7.22.5 Fundamental Equations of State

HelmholtzEOS is the base class for all Helmholtz energy fundamental equations of state.

class thermo.phases.**HelmholtzEOS**

Bases: `thermo.phases.phase.Phase`

Methods

`Cp`()	Method to calculate and return the constant-pressure heat capacity of the phase.
`Cv`()	Method to calculate and return the constant-volume heat capacity Cv of the phase.
`H`()	Method to calculate and return the enthalpy of the phase.
`S`()	Method to calculate and return the entropy of the phase.
`V_iter`([force])	Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure.
`d2P_dT2`()	Method to calculate and return the second temperature derivative of pressure of the phase.
`d2P_dTdV`()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.
`d2P_dV2`()	Method to calculate and return the second volume derivative of pressure of the phase.
`dH_dP`()	Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.
`dP_dT`()	Method to calculate and return the first temperature derivative of pressure of the phase.
`dP_dV`()	Method to calculate and return the first volume derivative of pressure of the phase.
`dS_dP`()	Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.
`lnphis`()	Method to calculate and return the log of fugacity coefficients of each component in the phase.
`to_TP_zs`(T, P, zs)	Method to create a new Phase object with the same constants as the existing Phase but at a different T and P.

Cp()

Method to calculate and return the constant-pressure heat capacity of the phase.

Returns

Cp [float] Molar heat capacity, [J/(mol*K)]

Cv()

Method to calculate and return the constant-volume heat capacity *Cv* of the phase.

$$C_v = T\left(\frac{\partial P}{\partial T}\right)_V^2/\left(\frac{\partial P}{\partial V}\right)_T + Cp$$

Returns

Cv [float] Constant volume molar heat capacity, [J/(mol*K)]

H()

Method to calculate and return the enthalpy of the phase. The reference state for most subclasses is an ideal-gas enthalpy of zero at 298.15 K and 101325 Pa.

Returns

H [float] Molar enthalpy, [J/(mol)]

S()

Method to calculate and return the entropy of the phase. The reference state for most subclasses is an ideal-gas entropy of zero at 298.15 K and 101325 Pa.

Returns

S [float] Molar entropy, [J/(mol*K)]

V_iter(*force=False*)

Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure. This often means the return value of this method is an mpmath *mpf*. This dummy method simply returns the implemented V method.

Returns

V [float or mpf] Molar volume, [m^3/mol]

__repr__()

Method to create a string representation of the phase object, with the goal of making it easy to obtain standalone code which reproduces the current state of the phase. This is extremely helpful in creating new test cases.

Returns

recreation [str] String which is valid Python and recreates the current state of the object if ran, [-]

Examples

```
>>> from thermo import IAPWS95Gas
>>> phase = IAPWS95Gas(T=300, P=1e5, zs=[1])
>>> phase
IAPWS95Gas(T=300, P=100000.0, zs=[1.0])
```

d2P_dT2()

Method to calculate and return the second temperature derivative of pressure of the phase.

Returns

d2P_dT2 [float] Second temperature derivative of pressure, [Pa/K^2]

`d2P_dTdV()`
Method to calculate and return the second derivative of pressure with respect to temperature and volume of the phase.

Returns

d2P_dTdV [float] Second volume derivative of pressure, [mol*Pa^2/(J*K)]

`d2P_dV2()`
Method to calculate and return the second volume derivative of pressure of the phase.

Returns

d2P_dV2 [float] Second volume derivative of pressure, [Pa*mol^2/m^6]

`dH_dP()`
Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.

Returns

dH_dP_T [float] Pressure derivative of enthalpy, [J/(mol*Pa)]

`dP_dT()`
Method to calculate and return the first temperature derivative of pressure of the phase.

Returns

dP_dT [float] First temperature derivative of pressure, [Pa/K]

`dP_dV()`
Method to calculate and return the first volume derivative of pressure of the phase.

Returns

dP_dV [float] First volume derivative of pressure, [Pa*mol/m^3]

`dS_dP()`
Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.

Returns

dS_dP_T [float] Pressure derivative of entropy, [J/(mol*K*Pa)]

`lnphis()`
Method to calculate and return the log of fugacity coefficients of each component in the phase.

Returns

lnphis [list[float]] Log fugacity coefficients, [-]

`to_TP_zs(`*T*, *P*, *zs*`)`
Method to create a new Phase object with the same constants as the existing Phase but at a different *T* and *P*.

Parameters

zs [list[float]] Molar composition of the new phase, [-]

T [float] Temperature of the new phase, [K]

P [float] Pressure of the new phase, [Pa]

Returns

new_phase [Phase] New phase at the specified conditions, [-]

Notes

This method is marginally faster than *Phase.to* as it does not need to check what the inputs are.

Examples

```
>>> from thermo import IdealGas
>>> phase = IdealGas(T=300, P=1e5, zs=[.79, .21], HeatCapacityGases=[])
>>> phase.to_TP_zs(T=1e5, P=1e3, zs=[.5, .5])
IdealGas(HeatCapacityGases=[], T=100000.0, P=1000.0, zs=[0.5, 0.5])
```

IAPWS95 is the base class for the IAPWS-95 formulation for water; *IAPWS95Gas* and *IAPWS95Liquid* are the gas and liquid sub-phases respectively.

class thermo.phases.**IAPWS95**(*T=None*, *P=None*, *zs=None*)
Bases: *thermo.phases.helmholtz_eos.HelmholtzEOS*

Methods

k()	Calculate and return the thermal conductivity of water according to the IAPWS.
mu()	Calculate and return the viscosity of water according to the IAPWS.

k()
Calculate and return the thermal conductivity of water according to the IAPWS. For details, see `chemicals.thermal_conductivity.k_IAPWS`.

Returns

k [float] Thermal conductivity of water, [W/m/K]

mu()
Calculate and return the viscosity of water according to the IAPWS. For details, see `chemicals.viscosity.mu_IAPWS`.

Returns

mu [float] Viscosity of water, [Pa*s]

class thermo.phases.**IAPWS95Gas**(*T=None*, *P=None*, *zs=None*)
Bases: *thermo.phases.iapws_phase.IAPWS95*

class thermo.phases.**IAPWS95Liquid**(*T=None*, *P=None*, *zs=None*)
Bases: *thermo.phases.iapws_phase.IAPWS95*

DryAirLemmon is an implementation of thermophysical properties of air by Lemmon (2000).

class thermo.phases.**DryAirLemmon**(*T=None*, *P=None*, *zs=None*)
Bases: *thermo.phases.helmholtz_eos.HelmholtzEOS*

Methods

k()	Calculate and return the thermal conductivity of air according to Lemmon and Jacobsen (2004) For details, see `chemicals.thermal_conductivity.k_air_lemmon`.
mu()	Calculate and return the viscosity of air according to the Lemmon and Jacobsen (2003) .

k()

Calculate and return the thermal conductivity of air according to Lemmon and Jacobsen (2004) For details, see `chemicals.thermal_conductivity.k_air_lemmon`.

Returns

k [float] Thermal conductivity of air, [W/m/K]

mu()

Calculate and return the viscosity of air according to the Lemmon and Jacobsen (2003) . For details, see `chemicals.viscosity.mu_air_lemmon`.

Returns

mu [float] Viscosity of air, [Pa*s]

7.22.6 CoolProp Wrapper

class thermo.phases.**CoolPropGas**(*backend*, *fluid*, *T=None*, *P=None*, *zs=None*, *Hfs=None*, *Gfs=None*, *Sfs=None*)

Bases: `thermo.phases.coolprop_phase.CoolPropPhase`

class thermo.phases.**CoolPropLiquid**(*backend*, *fluid*, *T=None*, *P=None*, *zs=None*, *Hfs=None*, *Gfs=None*, *Sfs=None*)

Bases: `thermo.phases.coolprop_phase.CoolPropPhase`

7.23 Phase Change Properties (thermo.phase_change)

This module contains implementations of *thermo.utils.TDependentProperty* representing enthalpy of vaporization and enthalpy of sublimation. A variety of estimation and data methods are available as included in the *chemicals* library.

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

7.23.1 Enthalpy of Vaporization

class thermo.phase_change.**EnthalpyVaporization**(*CASRN=''*, *Tb=None*, *Tc=None*, *Pc=None*, *omega=None*, *similarity_variable=None*, *Psat=None*, *Zl=None*, *Zg=None*, *extrapolation='Watson'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with heat of vaporization as a function of temperature. Consists of three constant value data sources, one source of tabular information, three coefficient-based methods, nine corresponding-states estimators, and the external library CoolProp.

Parameters

- **Tb** [float, optional] Boiling point, [K]
- **Tc** [float, optional] Critical temperature, [K]
- **Pc** [float, optional] Critical pressure, [Pa]
- **omega** [float, optional] Acentric factor, [-]
- **similarity_variable** [float, optional] similarity variable, n_atoms/MW, [mol/g]
- **Psat** [float or callable, optional] Vapor pressure at T or callable for the same, [Pa]
- **Zl** [float or callable, optional] Compressibility of liquid at T or callable for the same, [-]
- **Zg** [float or callable, optional] Compressibility of gas at T or callable for the same, [-]
- **CASRN** [str, optional] The CAS number of the chemical
- **load_data** [bool, optional] If False, do not load property coefficients from data sources in files [-]
- **extrapolation** [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]
- **method** [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.phase_change.MK`

`chemicals.phase_change.SMK`

`chemicals.phase_change.Velasco`

`chemicals.phase_change.Clapeyron`

`chemicals.phase_change.Riedel`

`chemicals.phase_change.Chen`

`chemicals.phase_change.Vetere`

`chemicals.phase_change.Liu`

`chemicals.phase_change.Watson`

Notes

To iterate over all methods, use the list stored in *enthalpy_vaporization_methods*.

CLAPEYRON: The Clapeyron fundamental model desecribed in Clapeyron. This is the model which uses *Zl*, *Zg*, and *Psat*, all of which must be set at each temperature change to allow recalculation of the heat of vaporization.

MORGAN_KOBAYASHI: The MK CSP model equation documented in MK.

SIVARAMAN_MAGEE_KOBAYASHI: The SMK CSP model equation documented in SMK.

VELASCO: The Velasco CSP model equation documented in Velasco.

PITZER: The Pitzer CSP model equation documented in Pitzer.

RIEDEL: The Riedel CSP model equation, valid at the boiling point only, documented in Riedel. This is adjusted with the Watson equation unless *Tc* is not available.

CHEN: The Chen CSP model equation, valid at the boiling point only, documented in Chen. This is adjusted with the Watson equation unless *Tc* is not available.

VETERE: The Vetere CSP model equation, valid at the boiling point only, documented in Vetere. This is adjusted with the Watson equation unless *Tc* is not available.

LIU: The Liu CSP model equation, valid at the boiling point only, documented in Liu. This is adjusted with the Watson equation unless *Tc* is not available.

CRC_HVAP_TB: The constant value available in [4] at the normal boiling point. This is adusted with the Watson equation unless *Tc* is not available. Data is available for 707 chemicals.

CRC_HVAP_298: The constant value available in [4] at 298.15 K. This is adusted with the Watson equation unless *Tc* is not available. Data is available for 633 chemicals.

GHARAGHEIZI_HVAP_298: The constant value available in [5] at 298.15 K. This is adusted with the Watson equation unless *Tc* is not available. Data is available for 2730 chemicals.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [3]. Very slow but accurate.

VDI_TABULAR: Tabular data in [4] along the saturation curve; interpolation is as set by the user or the default.

VDI_PPDS: Coefficients for a equation form developed by the PPDS, published openly in [3]. Extrapolates poorly at low temperatures.

DIPPR_PERRY_8E: A collection of 344 coefficient sets from the DIPPR database published openly in [6]. Provides temperature limits for all its fluids. chemicals.dippr.EQ106 is used for its fluids.

ALIBAKHSHI: One-constant limited temperature range regression method presented in [7], 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.

References

[1], [2], [3], [4], [5], [6], [7]

Attributes

interpolation_T

interpolation_property

interpolation_property_inv

Methods

`calculate`(T, method)	Method to calculate heat of vaporization of a liquid at temperature *T* with a given method.
`test_method_validity`(T, method)	Method to check the validity of a method.

`Watson_exponent = 0.38`
Exponent used in the Watson equation

`calculate`(*T*, *method*)
Method to calculate heat of vaporization of a liquid at temperature *T* with a given method.

This method has no exception handling; see `T_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate heat of vaporization, [K]

method [str] Name of the method to use

Returns

Hvap [float] Heat of vaporization of the liquid at T, [J/mol]

`interpolation_T = None`
No interpolation transformation by default.

`interpolation_property = None`
No interpolation transformation by default.

`interpolation_property_inv = None`
No interpolation transformation by default.

`name = 'Enthalpy of vaporization'`

`property_max = 1000000.0`
Maximum valid of heat of vaporization. Set to twice the value in the available data.

`property_min = 0`
Mimimum valid value of heat of vaporization. This occurs at the critical point exactly.

`ranked_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'MORGAN_KOBAYASHI', 'SIVARAMAN_MAGEE_KOBAYASHI', 'VELASCO', 'PITZER', 'VDI_TABULAR', 'ALIBAKHSHI', 'CRC_HVAP_TB', 'CRC_HVAP_298', 'GHARAGHEIZI_HVAP_298', 'CLAPEYRON', 'RIEDEL', 'CHEN', 'VETERE', 'LIU']`
Default rankings of the available methods.

`test_method_validity`(*T*, *method*)
Method to check the validity of a method. For CSP methods, the models are considered valid from 0 K to the critical point. For tabular data, extrapolation outside of the range is used if

tabular_extrapolation_permitted is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

The constant methods **CRC_HVAP_TB**, **CRC_HVAP_298**, and **GHARAGHEIZI_HVAP** are adjusted for temperature dependence according to the Watson equation, with a temperature exponent as set in *Watson_exponent*, usually regarded as 0.38. However, if Tc is not set, then the adjustment cannot be made. In that case the methods are considered valid for within 5 K of their boiling point or 298.15 K as appropriate.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'J/mol'

thermo.phase_change.**enthalpy_vaporization_methods = ['DIPPR_PERRY_8E', 'VDI_PPDS', 'COOLPROP', 'VDI_TABULAR', 'MORGAN_KOBAYASHI', 'SIVARAMAN_MAGEE_KOBAYASHI', 'VELASCO', 'PITZER', 'ALIBAKHSHI', 'CRC_HVAP_TB', 'CRC_HVAP_298', 'GHARAGHEIZI_HVAP_298', 'CLAPEYRON', 'RIEDEL', 'CHEN', 'VETERE', 'LIU']**

Holds all methods available for the EnthalpyVaporization class, for use in iterating over them.

7.23.2 Enthalpy of Sublimation

class thermo.phase_change.**EnthalpySublimation**(*CASRN=''*, *Tm=None*, *Tt=None*, *Cpg=None*, *Cps=None*, *Hvap=None*, *extrapolation='linear'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with heat of sublimation as a function of temperature. Consists of one temperature-dependent method based on the heat of sublimation at 298.15 K.

Parameters

CASRN [str, optional] The CAS number of the chemical

Tm [float, optional] Normal melting temperature, [K]

Tt [float, optional] Triple point temperature, [K]

Cpg [float or callable, optional] Gaseous heat capacity at a given temperature or callable for the same, [J/mol/K]

Cps [float or callable, optional] Solid heat capacity at a given temperature or callable for the same, [J/mol/K]

Hvap [float of callable, optional] Enthalpy of Vaporization at a given temperature or callable for the same, [J/mol]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

Notes

To iterate over all methods, use the list stored in *enthalpy_sublimation_methods*.

WEBBOOK_HSUB: Enthalpy of sublimation at a constant temperature of 298.15 K as given in [3].

GHARAGHEIZI_HSUB_298: Enthalpy of sublimation at a constant temperature of 298 K as given in [1].

GHARAGHEIZI_HSUB: Enthalpy of sublimation at a constant temperature of 298 K as given in [1] are adjusted using the solid and gas heat capacity functions to correct for any temperature.

CRC_HFUS_HVAP_TM: Enthalpies of fusion in [1] are corrected to be enthalpies of sublimation by adding the enthalpy of vaporization at the fusion temperature, and then adjusted using the solid and gas heat capacity functions to correct for any temperature.

References

[1], [2], [3]

Attributes

interpolation_T

interpolation_property

interpolation_property_inv

Methods

`calculate`(T, method)	Method to calculate heat of sublimation of a solid at temperature T with a given method.
`test_method_validity`(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)

Method to calculate heat of sublimation of a solid at temperature T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate heat of sublimation, [K]

method [str] Name of the method to use

Returns

Hsub [float] Heat of sublimation of the solid at T, [J/mol]

interpolation_T = None

No interpolation transformation by default.

interpolation_property = None

No interpolation transformation by default.

interpolation_property_inv = None

No interpolation transformation by default.

name = 'Enthalpy of sublimation'

property_max = 1000000.0
Maximum valid of heat of sublimation. A theoretical concept only.

property_min = 0
Mimimum valid value of heat of vaporization. A theoretical concept only.

ranked_methods = ['WEBBOOK_HSUB', 'GHARAGHEIZI_HSUB', 'CRC_HFUS_HVAP_TM', 'GHARAGHEIZI_HSUB_298']

test_method_validity(*T*, *method*)
Method to check the validity of a method. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'J/mol'

thermo.phase_change.**enthalpy_sublimation_methods = ['WEBBOOK_HSUB', 'GHARAGHEIZI_HSUB', 'CRC_HFUS_HVAP_TM', 'GHARAGHEIZI_HSUB_298']**
Holds all methods available for the EnthalpySublimation class, for use in iterating over them.

7.24 Legacy Property Packages (thermo.property_package)

Warning: These classes were a first attempt at rigorous multiphase equilibrium. They may be useful in some special cases but they are not complete and further development will not happen. They were never documented as well.

It is recommended to switch over to the *thermo.flash* interface which seeks to be more modular, easier to maintain and extend, higher-performance, and easier to modify.

7.25 Phase Identification (thermo.phase_identification)

This module contains functions for identifying phases as liquid, solid, and gas.

Solid identification is easy using the `phase identification parameter`. There is never more than one gas by definition. For pure species, the phase identification parameter is a clear vapor-liquid differentiator in the subcritical region and it provides line starting at the critical point for the supercritical region.

However for mixtures, there is no clear calcuation that can be performed to identify the phase of a mixture. Many different criteria that have been proposed are included here. The `phase identification parameter or PIP`. is recommended in general and is the default.

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

7.25.1 Phase Identification

Main Interface

thermo.phase_identification.**identify_sort_phases**(*phases*, *betas*, *constants*, *correlations*, *settings*, *skip_solids=False*)

Identify and sort all phases given the provided parameters.

Parameters

phases [list[*Phase*]] Phases to be identified and sorted, [-]

betas [list[float]] Phase molar fractions, [-]

constants [*ChemicalConstantsPackage*] Constants used in the identification, [-]

correlations [*PropertyCorrelationsPackage*] Correlations used in the identification, [-]

settings [*BulkSettings*] Settings object controlling the phase ID, [-]

skip_solids [bool] Set this to True if no phases are provided which can represent a solid phase, [-]

Returns

gas [*Phase*] Gas phase, if one was identified, [-]

liquids [list[*Phase*]] Liquids that were identified and sorted, [-]

solids [list[*Phase*]] Solids that were identified and sorted, [-]

betas [list[float]] Sorted phase molar fractions, in order (gas, liquids…, solids…) [-]

Notes

This step is very important as although phase objects are designed to represent a single phase, cubic equations of state can be switched back and forth by the flash algorithms. Thermodynamics doesn't care about gases, liquids, or solids; it just cares about minimizing Gibbs energy!

Examples

A butanol-water-ethanol flash yields three phases. For brevity we skip the flash and initialize our *gas*, *liq0*, and *liq1* object with the correct phase composition. Then we identify the phases into liquid, gas, and solid.

```
>>> from thermo import ChemicalConstantsPackage, PropertyCorrelationsPackage,␣
↪HeatCapacityGas, SRKMIX, CEOSGas, CEOSLiquid
>>> constants = ChemicalConstantsPackage(Tcs=[563.0, 647.14, 514.0], Vcs=[0.000274,␣
↪5.6e-05, 0.000168], Pcs=[4414000.0, 22048320.0, 6137000.0], omegas=[0.59, 0.344,␣
↪0.635], MWs=[74.1216, 18.01528, 46.06844], CASs=['71-36-3', '7732-18-5', '64-17-5
↪'])
>>> properties = PropertyCorrelationsPackage(constants=constants, skip_missing=True,
...                                     HeatCapacityGases=[HeatCapacityGas(load_
↪data=False, poly_fit=(50.0, 1000.0, [-3.787200194613107e-20, 1.7692887427654656e-
↪16, -3.445247207129205e-13, 3.612771874320634e-10, -2.1953250181084466e-07, 7.
↪707135849197655e-05, -0.014658388538054169, 1.5642629364740657, -7.
↪614560475001724])),
...                                     HeatCapacityGas(load_data=False, poly_
↪fit=(50.0, 1000.0, [5.543665000518528e-22, -2.403756749600872e-18, 4.
↪2166477594350336e-15, -3.7965208514613565e-12, 1.823547122838406e-09, -4.
↪3747690853614695e-07, 5.437938301211039e-05, -0.003220061088723078, 33.
↪32731489750759])),
...                                     HeatCapacityGas(load_data=False, poly_
↪fit=(50.0, 1000.0, [-1.162767978165682e-20, 5.4975285700787494e-17, -1.
↪0861242757337942e-13, 1.1582703354362728e-10, -7.160627710867427e-08, 2.
↪5392014654765875e-05, -0.004732593693568646, 0.5072291035198603, 20.
↪037826650765965])),], )
>>> eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
>>> gas = CEOSGas(SRKMIX, eos_kwargs, HeatCapacityGases=properties.
↪HeatCapacityGases)
>>> liq = CEOSLiquid(SRKMIX, eos_kwargs, HeatCapacityGases=properties.
↪HeatCapacityGases)
>>> T, P = 361, 1e5
>>> gas = gas.to(T=T, P=P, zs=[0.2384009970908655, 0.5786839935180925, 0.
↪1829150093910419])
>>> liq0 = liq.to(T=T, P=P, zs=[7.619975052238032e-05, 0.9989622883894993, 0.
↪0009615118599781474])
>>> liq1 = liq.to(T=T, P=P, zs=[0.6793120076703771, 0.19699746328631124, 0.
↪12369052904331178])
>>> res = identity_phase_states(phases=[liq0, liq1, gas], constants=constants,␣
↪correlations=properties, VL_method='PIP')
>>> res[0] is gas, res[1][0] is liq0, res[1][1] is liq1, res[2]
(True, True, True, [])
```

Secondary Interfaces

`thermo.phase_identification.identity_phase_states(phases, constants, correlations, VL_method='PIP', S_method='d2P_dVdT', VL_ID_settings=None, S_ID_settings=None, skip_solids=False)`

Identify and the actial phase of all the given phases given the provided settings.

Parameters

phases [list[*Phase*]] Phases to be identified and sorted, [-]

constants [*ChemicalConstantsPackage*] Constants used in the identification, [-]

correlations [*PropertyCorrelationsPackage*] Correlations used in the identification, [-]

VL_method [str, optional] One of *VL_ID_METHODS*, [-]

S_method [str, optional] One of *S_ID_METHODS*, [-]

VL_ID_settings [dict[str][float] or None, optional] Additional configuration options for vapor-liquid phase ID, [-]

S_ID_settings [dict[str][float] or None, optional] Additional configuration options for solid-liquid phase ID, [-]

skip_solids [bool] Set this to True if no phases are provided which can represent a solid phase, [-]

Returns

gas [*Phase*] Gas phase, if one was identified, [-]

liquids [list[*Phase*]] Liquids that were identified and sorted, [-]

solids [list[*Phase*]] Solids that were identified and sorted, [-]

`thermo.phase_identification.VL_ID_METHODS = ['Tpc', 'Vpc', 'Tpc Vpc weighted', 'Tpc Vpc', 'Wilson', 'Poling', 'PIP', 'Bennett-Schmidt', 'Traces']`

List of all the methods available to perform the Vapor-Liquid phase ID.

`thermo.phase_identification.S_ID_METHODS = ['d2P_dVdT']`

List of all the methods available to perform the solid-liquid phase ID.

Scoring Functions

`thermo.phase_identification.score_phases_VL(phases, constants, correlations, method)`

Score all phases given the provided parameters and a selected method.

A score above zero indicates a potential gas. More than one phase may have a score above zero, in which case the highest scoring phase is the gas, and the other is a liquid.

Parameters

phases [list[*thermo.phases.Phase*]] Phases to be identified and sorted, [-]

constants [*ChemicalConstantsPackage*] Constants used in the identification, [-]

correlations [*PropertyCorrelationsPackage*] Correlations used in the identification, [-]

method [str] Setting configuring how the scoring is performed; one of 'Tpc', 'Vpc', 'Tpc Vpc weighted', 'Tpc Vpc', 'Wilson', 'Poling', 'PIP', 'Bennett-Schmidt', 'Traces', [-]

Returns

scores [list[float]] Scores for the phases in the order provided, [-]

Examples

```
>>> from thermo import ChemicalConstantsPackage, PropertyCorrelationsPackage,␣
↪CEOSGas, CEOSLiquid, PRMIX, HeatCapacityGas
>>> constants = ChemicalConstantsPackage(CASs=['124-38-9', '110-54-3'], Vcs=[9.4e-
↪05, 0.000368], MWs=[44.0095, 86.17536], names=['carbon dioxide', 'hexane'],␣
↪omegas=[0.2252, 0.2975], Pcs=[7376460.0, 3025000.0], Tbs=[194.67, 341.87],␣
↪Tcs=[304.2, 507.6], Tms=[216.65, 178.075])
>>> correlations = PropertyCorrelationsPackage(constants=constants, skip_
↪missing=True, HeatCapacityGases=[HeatCapacityGas(poly_fit=(50.0, 1000.0, [-3.
↪1115474168865828e-21, 1.39156078498805e-17, -2.5430881416264243e-14, 2.
↪4175307893014295e-11, -1.2437314771044867e-08, 3.1251954264658904e-06, -0.
↪00021220221928610925, 0.000884685506352987, 29.266811602924644])),␣
↪HeatCapacityGas(poly_fit=(200.0, 1000.0, [1.3740654453881647e-21, -8.
↪344496203280677e-18, 2.2354782954548568e-14, -3.4659555330048226e-11, 3.
↪410703030634579e-08, -2.1693611029230923e-05, 0.008373280796376588, -1.
↪356180511425385, 175.67091124888998]))])
>>> T, P, zs = 300.0, 1e6, [.5, .5]
>>> eos_kwargs = {'Pcs': constants.Pcs, 'Tcs': constants.Tcs, 'omegas': constants.
↪omegas}
>>> gas = CEOSGas(PRMIX, eos_kwargs, HeatCapacityGases=correlations.
↪HeatCapacityGases, T=T, P=P, zs=zs)
>>> liq = CEOSLiquid(PRMIX, eos_kwargs, HeatCapacityGases=correlations.
↪HeatCapacityGases, T=T, P=P, zs=zs)
```

A sampling of different phase identification methods is below:

```
>>> score_phases_VL([gas, liq], constants, correlations, method='PIP')
[1.6409446310, -7.5692120928]
>>> score_phases_VL([gas, liq], constants, correlations, method='Vpc')
[0.00144944049, -0.0001393075288]
>>> score_phases_VL([gas, liq], constants, correlations, method='Tpc Vpc')
[113.181283525, -29.806038704]
>>> score_phases_VL([gas, liq], constants, correlations, method='Bennett-Schmidt')
[0.0003538299416, -2.72255439503e-05]
>>> score_phases_VL([gas, liq], constants, correlations, method='Poling')
[0.1767828268, -0.004516837897]
```

thermo.phase_identification.**score_phases_S**(*phases*, *constants*, *correlations*, *method='d2P_dVdT'*, *S_ID_settings=None*)

Score all phases according to how wolid they appear given the provided parameters and a selected method.

A score above zero indicates a solid. More than one phase may have a score above zero. A score under zero means the phase is a liquid or gas.

Parameters

phases [list[*thermo.phases.Phase*]] Phases to be identified and sorted, [-]

constants [*ChemicalConstantsPackage*] Constants used in the identification, [-]

correlations [*PropertyCorrelationsPackage*] Correlations used in the identification, [-]

method [str] Setting configuring how the scoring is performed; one of ('d2P_dVdT',), [-]

S_ID_settings [dict[str][float] or None, optional] Additional configuration options for solid-liquid phase ID, [-]

Returns

scores [list[float]] Scores for the phases in the order provided, [-]

`thermo.phase_identification.`**`vapor_score_traces`**(*zs*, *CASs*, *Tcs*, *trace_CASs=['74-82-8', '7727-37-9']*, *min_trace=0.0*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the concept of which phase has the most of the lightest compound. This nicely sidesteps issues in many other methods, at the expense that it cannot be applied when there is only one phase and it is not smart enough to handle liquid-liquid cases.

If no trace components are present, the component with the lowest critical temperature's concentration is returned. Because of the way this is implemented, the score is always larger than 1.0.

Parameters

zs [list[float]] Mole fractions of the phase being identified, [-]

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

Tcs [list[float]] Critical temperatures of all species, [K]

trace_CASs [list[str]] Trace components to use for identification; if more than one component is given, the first component present in both *CASs* and *trace_CASs* is the one used, [-]

min_trace [float] Minimum concentration to make a phase appear vapor-like; subtracted from the concentration which would otherwise be returned, [-]

Returns

score [float] Vapor like score, [-]

Examples

A flash of equimolar CO2/n-hexane at 300 K and 1 MPa is computed, and there is a two phase solution. The phase must be identified for each result:

Liquid-like phase:

```
>>> vapor_score_traces(zs=[.218, .782], Tcs=[304.2, 507.6], CASs=['124-38-9', '110-
↪54-3'])
0.218
```

Vapor-like phase:

```
>>> vapor_score_traces(zs=[.975, .025], Tcs=[304.2, 507.6], CASs=['124-38-9', '110-
↪54-3'])
0.975
```

`thermo.phase_identification.`**`vapor_score_Tpc`**(*T*, *Tcs*, *zs*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the following criteria

$$T - \sum_i z_i T_{c,i}$$

Parameters

T [float] Temperature, [K]

Tcs [list[float]] Critical temperatures of all species, [K]

zs [list[float]] Mole fractions of the phase being identified, [-]

Returns

score [float] Vapor like score, [-]

Examples

A flash of equimolar CO2/n-hexane at 300 K and 1 MPa is computed, and there is a two phase solution. The phase must be identified for each result:

Liquid-like phase:

```
>>> vapor_score_Tpc(T=300.0, Tcs=[304.2, 507.6], zs=[0.21834418746784942, 0.
↪7816558125321506])
-163.18879226903942
```

Vapor-like phase:

```
>>> vapor_score_Tpc(T=300.0, Tcs=[304.2, 507.6], zs=[0.9752234962374878, 0.
↪024776503762512052])
-9.239540865294941
```

In this result, the vapor phase is not identified as a gas at all! It has a mass density of ~ 20 kg/m^3, which would usually be called a gas by most people.

thermo.phase_identification.**vapor_score_Vpc**(*V*, *Vcs*, *zs*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the following criteria

$$V - \sum_i z_i V_{c,i}$$

Parameters

V [float] Molar volume, [m^3/mol]

Vcs [list[float]] Critical molar volumes of all species, [m^3/mol]

zs [list[float]] Mole fractions of the phase being identified, [-]

Returns

score [float] Vapor like score, [-]

Examples

A flash of equimolar CO2/n-hexane at 300 K and 1 MPa is computed, and there is a two phase solution. The phase must be identified for each result:

Liquid-like phase:

```
>>> vapor_score_Vpc(V=0.00011316308855449715, Vcs=[9.4e-05, 0.000368], zs=[0.
↪21834418746784942, 0.7816558125321506])
-0.000195010604079
```

Vapor-like phase:

```
>>> vapor_score_Vpc(V=0.0023406573328250335, Vcs=[9.4e-05, 0.000368], zs=[0.
↪9752234962374878, 0.024776503762512052])
0.002239868570
```

thermo.phase_identification.**vapor_score_Tpc_weighted**(*T*, *Tcs*, *Vcs*, *zs*, *r1=1.0*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the following criteria, said to be implemented in ECLIPSE [1]:

$$T - T_{pc}$$

$$T_{p,c} = r_1 \frac{\sum_j x_j V_{c,j} T_{c,j}}{\sum_j x_j V_{c,j}}$$

Parameters

T [float] Temperature, [K]

Tcs [list[float]] Critical temperatures of all species, [K]

Vcs [list[float]] Critical molar volumes of all species, [m^3/mol]

zs [list[float]] Mole fractions of the phase being identified, [-]

r1 [float] Tuning factor, [-]

Returns

score [float] Vapor like score, [-]

References

[1]

Examples

A flash of equimolar CO2/n-hexane at 300 K and 1 MPa is computed, and there is a two phase solution. The phase must be identified for each result:

Liquid-like phase:

```
>>> vapor_score_Tpc_weighted(T=300.0, Tcs=[304.2, 507.6], Vcs=[9.4e-05, 0.000368],␣
↪zs=[0.21834418746784942, 0.7816558125321506])
-194.0535694431
```

Vapor-like phase:

```
>>> vapor_score_Tpc_weighted(T=300.0, Tcs=[304.2, 507.6], Vcs=[9.4e-05, 0.000368],␣
↪zs=[0.9752234962374878, 0.024776503762512052])
-22.60037521107
```

As can be seen, the CO2-phase is incorrectly identified as a liquid.

thermo.phase_identification.**vapor_score_Tpc_Vpc**(*T*, *V*, *Tcs*, *Vcs*, *zs*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the following criteria, said to be implemented in Multiflash [1]:

$$VT^2 - V_{pc}T_{pc}^2$$

Parameters

T [float] Temperature, [K]

V [float] Molar volume, [m^3/mol]

Tcs [list[float]] Critical temperatures of all species, [K]

Vcs [list[float]] Critical molar volumes of all species, [m^3/mol]

zs [list[float]] Mole fractions of the phase being identified, [-]

Returns

score [float] Vapor like score, [-]

References

[1]

Examples

A flash of equimolar CO2/n-hexane at 300 K and 1 MPa is computed, and there is a two phase solution. The phase must be identified for each result:

Liquid-like phase:

```
>>> vapor_score_Tpc_Vpc(T=300.0, V=0.00011316308855449715, Tcs=[304.2, 507.6],␣
↪Vcs=[9.4e-05, 0.000368], zs=[0.21834418746784942, 0.7816558125321506])
-55.932094761
```

Vapor-like phase:

```
>>> vapor_score_Tpc_Vpc(T=300.0, V=0.0023406573328250335, Tcs=[304.2, 507.6],␣
↪Vcs=[9.4e-05, 0.000368], zs=[0.9752234962374878, 0.024776503762512052])
201.020821992
```

`thermo.phase_identification.`**`vapor_score_Wilson`**(*T*, *P*, *zs*, *Tcs*, *Pcs*, *omegas*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the Rachford-Rice Wilson method of Perschke [1].

After calculating Wilson's K values, the following expression is evaluated at $\frac{V}{F} = 0.5$; the result is the score.

$$\sum_i \frac{z_i(K_i - 1)}{1 + \frac{V}{F}(K_i - 1)}$$

Parameters

T [float] Temperature, [K]

P [float] Pressure, [Pa]

zs [list[float]] Mole fractions of the phase being identified, [-]

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, [-]

Returns

score [float] Vapor like score, [-]

References

[1]

Examples

A flash of equimolar CO2/n-hexane at 300 K and 1 MPa is computed, and there is a two phase solution. The phase must be identified for each result:

Liquid-like phase:

```
>>> vapor_score_Wilson(T=300.0, P=1e6, zs=[.218, .782], Tcs=[304.2, 507.6],␣
↪Pcs=[7376460.0, 3025000.0], omegas=[0.2252, 0.2975])
-1.16644793
```

Vapor-like phase:

```
>>> vapor_score_Wilson(T=300.0, P=1e6, zs=[.975, .025], Tcs=[304.2, 507.6],␣
↪Pcs=[7376460.0, 3025000.0], omegas=[0.2252, 0.2975])
1.397678492
```

This method works well in many conditions, like the Wilson equation itself, but fundamentally it cannot do a great job because it is not tied to the phase model itself.

A dew point flash at P = 100 Pa for the same mixture shows both phases being identified as vapor-like:

```
>>> T_dew = 206.40935716944634
>>> P = 100.0
>>> vapor_score_Wilson(T=T_dew, P=P, zs=[0.5, 0.5], Tcs=[304.2, 507.6],␣
↪Pcs=[7376460.0, 3025000.0], omegas=[0.2252, 0.2975])
1.074361930956633
>>> vapor_score_Wilson(T=T_dew, P=P, zs=[0.00014597910182360052, 0.
↪9998540208981763], Tcs=[304.2, 507.6], Pcs=[7376460.0, 3025000.0], omegas=[0.2252,
↪ 0.2975])
0.15021784286075726
```

thermo.phase_identification.**vapor_score_Poling**(*kappa*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the isothermal compressibility *kappa* concept by Poling [1].

$$\text{score} = (\kappa - 0.005\text{atm}^{-1})$$

Parameters

kappa [float] Isothermal coefficient of compressibility, [1/Pa]

Returns

score [float] Vapor like score, [-]

Notes

A second criteria which is not implemented as it does not fit with the scoring concept is for liquids:

$$\frac{0.9}{P} < \beta < \frac{3}{P}$$

References

[1]

Examples

CO2 vapor properties computed with Peng-Robinson at 300 K and 1 bar:

```
>>> vapor_score_Poling(1.0054239121594122e-05)
1.013745778995
```

n-hexane liquid properties computed with Peng-Robinson at 300 K and 10 bar:

```
>>> vapor_score_Poling(2.121777078782957e-09)
-0.00478501093
```

thermo.phase_identification.**vapor_score_PIP**(*V*, *dP_dT*, *dP_dV*, *d2P_dV2*, *d2P_dVdT*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the PIP concept.

$$\text{score} = -(\Pi - 1)$$

$$\Pi = V\left[\frac{\frac{\partial^2 P}{\partial V \partial T}}{\frac{\partial P}{\partial T}} - \frac{\frac{\partial^2 P}{\partial V^2}}{\frac{\partial P}{\partial V}}\right]$$

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

score [float] Vapor like score, [-]

References

[1]

Examples

CO2 vapor properties computed with Peng-Robinson at 300 K and 1 bar:

```
>>> vapor_score_PIP(0.024809176851423774, 337.0119286073647, -4009021.959558917,␣
↪321440573.3615088, -13659.63987996052)
0.016373735005
```

n-hexane liquid properties computed with Peng-Robinson at 300 K and 10 bar:

```
>>> vapor_score_PIP(0.00013038156684574785, 578477.8796379718, -3614798144591.8984,␣
↪4.394997991022487e+17, -20247865009.795322)
-10.288635225
```

thermo.phase_identification.**vapor_score_Bennett_Schmidt**(*dbeta_dT*)

Compute a vapor score representing how vapor-like a phase is (higher, above zero = more vapor like) using the Bennet-Schmidt temperature derivative of isobaric expansion suggestion.

$$\text{score} = -\left(\frac{\partial \beta}{\partial T}\right)$$

Parameters

dbeta_dT [float] Temperature derivative of isobaric coefficient of a thermal expansion, [1/K^2]

Returns

score [float] Vapor like score, [-]

References

[1]

Examples

CO2 vapor properties computed with Peng-Robinson at 300 K and 1 bar:

```
>>> vapor_score_Bennett_Schmidt(-1.1776172267959163e-05)
1.1776172267959163e-05
```

n-hexane liquid properties computed with Peng-Robinson at 300 K and 10 bar:

```
>>> vapor_score_Bennett_Schmidt(7.558572848883679e-06)
-7.558572848883679e-06
```

7.25.2 Sorting Phases

thermo.phase_identification.**sort_phases**(*liquids*, *solids*, *constants*, *settings*)

Identify and sort all phases given the provided parameters. This is not a thermodynamic concept; it is just a convinience method to make the results of the flash more consistent, because the flash algorithms don't care about density or ordering the phases.

Parameters

liquids [list[*Phase*]] Liquids that were identified, [-]

solids [list[*Phase*]] Solids that were identified, [-]

constants [*ChemicalConstantsPackage*] Constants used in the identification, [-]

correlations [*PropertyCorrelationsPackage*] Correlations used in the identification, [-]

settings [*BulkSettings*] Settings object controlling the phase sorting, [-]

Returns

liquids [list[*Phase*]] Liquids that were identified and sorted, [-]

solids [list[*Phase*]] Solids that were identified and sorted, [-]

Notes

The settings object uses the preferences *liquid_sort_method*, *liquid_sort_prop*, *liquid_sort_cmps*, *liquid_sort_cmps_neg*, and *phase_sort_higher_first*.

7.26 Regular Solution Gibbs Excess Model (thermo.regular_solution)

This module contains a class `RegularSolution` for performing activity coefficient calculations with the regular solution model.

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

7.26.1 Regular Solution Class

class thermo.regular_solution.**RegularSolution**(*T*, *xs*, *Vs*, *SPs*, *lambda_coeffs=None*)

Bases: `thermo.activity.GibbsExcess`

Class for representing an a liquid with excess gibbs energy represented by the Regular Solution model. This model is not temperature dependent and has limited predictive ability, but can be used without interaction parameters. This model is described in [1].

$$G^E = \frac{\sum_m \sum_n (x_m x_n V_m V_n A_{mn})}{\sum_m x_m V_m}$$

$$A_{mn} = 0.5(\delta_m - \delta_n)^2 - \delta_m \delta_n k_{mn}$$

In the above equation, δ represents the solubility parameters, and k_{mn} is the interaction coefficient between m and n. The model makes no assumption about the symmetry of this parameter.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Vs [list[float]] Molar volumes of each compond at a reference temperature (often 298.15 K), [m^3/mol]

SPs [list[float]] Solubility parameters of each compound; normally at a reference temperature of 298.15 K, [Pa^0.5]

lambda_coeffs [list[list[float]], optional] Optional interaction parameters, [-]

Notes

In addition to the methods presented here, the methods of its base class `thermo.activity.GibbsExcess` are available as well.

Additional equations of note are as follows.

$$G^E = H^E$$

$$S^E = 0$$

$$\delta = \sqrt{\frac{\Delta H_{vap} - RT}{V_m}}$$

References

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

Examples

Example 1

From [2], calculate the activity coefficients at infinite dilution for the system benzene-cyclohexane at 253.15 K using the regular solution model (example 5.20, with unit conversion in-line):

```
>>> from scipy.constants import calorie
>>> GE = RegularSolution(T=353.15, xs=[.5, .5], Vs=[89E-6, 109E-6], SPs=[9.
↪2*(calorie*1e6)**0.5, 8.2*(calorie*1e6)**0.5])
>>> GE.gammas_infinite_dilution()
[1.1352128394, 1.16803058378]
```

This matches the solution given of [1.135, 1.168].

Example 2

Benzene and cyclohexane calculation from [3], without interaction parameters.

```
>>> GE = RegularSolution(T=353, xs=[0.01, 0.99], Vs=[8.90e-05, 1.09e-04], SPs=[9.
↪2*(calorie/1e-6)**0.5, 8.2*(calorie/1e-6)**0.5])
>>> GE.gammas()
[1.1329295, 1.00001039]
```

Example 3

Another common model is the Flory-Huggins model. This isn't implemented as a separate model, but it is possible to modify the activity coefficient results of `RegularSolution` to obtain the activity coefficients from the Flory-Huggins model anyway. ChemSep [4] implements the Flory-Huggins model and calls it the regular solution model, so results can't be compared with ChemSep except when making the following manual solution. The example below uses parameters from ChemSep for ethanol and water.

```
>>> GE = RegularSolution(T=298.15, xs=[0.5, 0.5], Vs=[0.05868e-3, 0.01807e-3],␣
↪SPs=[26140.0, 47860.0])
>>> GE.gammas() # Regular solution activity coefficients
[1.8570955489, 7.464567232]
>>> lngammass = [log(g) for g in GE.gammas()]
>>> thetas = [GE.Vs[i]/sum(GE.xs[i]*GE.Vs[i] for i in range(GE.N)) for i in␣
↪range(GE.N)]
>>> gammas_flory_huggins = [exp(lngammass[i] + log(thetas[i]) + 1 - thetas[i]) for␣
↪i in range(GE.N)]
>>> gammas_flory_huggins
[1.672945693, 5.9663471]
```

This matches the values calculated from ChemSep exactly.

Attributes

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Vs [list[float]] Molar volumes of each compond at a reference temperature (often 298.15 K), [K]

SPs [list[float]] Solubility parameters of each compound; normally at a reference temperature of 298.15 K, [Pa^0.5]

lambda_coeffs [list[list[float]]] Interaction parameters, [-]

Methods

GE()	Calculate and return the excess Gibbs energy of a liquid phase using the regular solution model.
d2GE_dT2()	Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phas.
d2GE_dTdxs()	Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy.
d2GE_dxixjs()	Calculate and return the second mole fraction derivatives of excess Gibbs energy of a liquid phase using the regular solution model.
d3GE_dT3()	Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase.
d3GE_dxixjxks()	Calculate and return the third mole fraction derivatives of excess Gibbs energy.
dGE_dT()	Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase.
dGE_dxs()	Calculate and return the mole fraction derivatives of excess Gibbs energy of a liquid phase using the regular solution model.
to_T_xs(T, xs)	Method to construct a new `RegularSolution` instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

GE()
Calculate and return the excess Gibbs energy of a liquid phase using the regular solution model.

$$G^E = \frac{\sum_m \sum_n (x_m x_n V_m V_n A_{mn})}{\sum_m x_m V_m}$$

$$A_{mn} = 0.5(\delta_m - \delta_n)^2 - \delta_m \delta_n k_{mn}$$

Returns

GE [float] Excess Gibbs energy, [J/mol]

d2GE_dT2()
Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phas.

$$\frac{\partial^2 g^E}{\partial T^2} = 0$$

Returns

d2GE_dT2 [float] Second temperature derivative of excess Gibbs energy, [J/(mol*K^2)]

d2GE_dTdxs()
Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy.

$$\frac{\partial^2 g^E}{\partial x_i \partial T} = 0$$

Returns

d2GE_dTdxs [list[float]] Temperature derivative of mole fraction derivatives of excess Gibbs energy, [J/(mol*K)]

d2GE_dxixjs()
Calculate and return the second mole fraction derivatives of excess Gibbs energy of a liquid phase using the regular solution model.

$$\frac{\partial^2 G^E}{\partial x_i \partial x_j} = \frac{V_j(V_i G^E - H_{ij})}{(\sum_m V_m x_m)^2} - \frac{V_i \frac{\partial G^E}{\partial x_j}}{\sum_m V_m x_m} + \frac{V_i V_j[\delta_i \delta_j (k_{ji} + k_{ij}) + (\delta_i - \delta_j)^2]}{\sum_m V_m x_m}$$

Returns

d2GE_dxixjs [list[list[float]]] Second mole fraction derivatives of excess Gibbs energy, [J/mol]

d3GE_dT3()
Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase.

$$\frac{\partial^3 g^E}{\partial T^3} = 0$$

Returns

d3GE_dT3 [float] Third temperature derivative of excess Gibbs energy, [J/(mol*K^3)]

d3GE_dxixjxks()
Calculate and return the third mole fraction derivatives of excess Gibbs energy.

$$\frac{\partial^3 G^E}{\partial x_i \partial x_j \partial x_k} = \frac{-2V_i V_j V_k G^E + 2 V_j V_k H_{ij}}{(\sum_m V_m x_m)^3} + \frac{V_i \left(V_j \frac{\partial G^E}{\partial x_k} + V_k \frac{\partial G^E}{\partial x_j}\right)}{(\sum_m V_m x_m)^2} - \frac{V_i \frac{\partial^2 G^E}{\partial x_j \partial x_k}}{\sum_m V_m x_m} - \frac{V_i V_j V_k[\delta_i(\delta_j(k_{ij} + k_{ji}) + \delta_k(k_{ik}}{(\sum_m}$$

Returns

d3GE_dxixjxks [list[list[list[float]]]] Third mole fraction derivatives of excess Gibbs energy, [J/mol]

dGE_dT()

Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase.

$$\frac{\partial g^E}{\partial T} = 0$$

Returns

dGE_dT [float] First temperature derivative of excess Gibbs energy, [J/(mol*K)]

dGE_dxs()

Calculate and return the mole fraction derivatives of excess Gibbs energy of a liquid phase using the regular solution model.

$$\frac{\partial G^E}{\partial x_i} = \frac{-V_i G^E + \sum_m V_i V_m x_m[\delta_i\delta_m(k_{mi} + k_{im}) + (\delta_i - \delta_m)^2]}{\sum_m V_m x_m}$$

Returns

dGE_dxs [list[float]] Mole fraction derivatives of excess Gibbs energy, [J/mol]

to_T_xs(*T*, *xs*)

Method to construct a new `RegularSolution` instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions of each component, [-]

Returns

obj [RegularSolution] New `RegularSolution` object at the specified conditions [-]

7.26.2 Regular Solution Regression Calculations

thermo.regular_solution.**regular_solution_gammas_binaries**(*xs*, *Vs*, *SPs*, *Ts*, *lambda12*, *lambda21*, *gammas=None*)

Calculates activity coefficients with the regular solution model at fixed *lambda* values for a binary system at a series of mole fractions at specified temperatures. This is used for regression of *lambda* parameters. This function is highly optimized, and operates on multiple points at a time.

$$\ln \gamma_1 = \frac{V_1\phi_2^2}{RT}\left[(\text{SP}_1 - \text{SP}_2)^2 + \lambda_{12}\text{SP}_1\text{SP}_2 + \lambda_{21}\text{SP}_1\text{SP}_2\right]$$

$$\ln \gamma_2 = \frac{V_2\phi_1^2}{RT}\left[(\text{SP}_1 - \text{SP}_2)^2 + \lambda_{12}\text{SP}_1\text{SP}_2 + \lambda_{21}\text{SP}_1\text{SP}_2\right]$$

$$\phi_1 = \frac{x_1 V_1}{x_1 V_1 + x_2 V_2}$$

$$\phi_2 = \frac{x_2 V_2}{x_1 V_1 + x_2 V_2}$$

Parameters

xs [list[float]] Liquid mole fractions of each species in the format x0_0, x1_0, (component 1 point1, component 2 point 1), x0_1, x1_1, (component 1 point2, component 2 point 2), ... size pts*2 [-]

Vs [list[float]] Molar volumes of each of the two components, [m^3/mol]

SPs [list[float]] Solubility parameters of each of the two components, [Pa^0.5]

Ts [flist[float]] Temperatures of each composition point; half the length of *xs*, [K]

lambda12 [float] *lambda* parameter for 12, [-]

lambda21 [float] *lambda* parameter for 21, [-]

gammas [list[float], optional] Array to store the activity coefficient for each species in the liquid mixture, indexed the same as *xs*; can be omitted or provided for slightly better performance [-]

Returns

gammas [list[float]] Activity coefficient for each species in the liquid mixture, indexed the same as *xs*, [-]

Examples

```
>>> regular_solution_gammas_binaries([.1, .9, 0.3, 0.7, .85, .15], Vs=[7.421e-05, 8.
→068e-05], SPs=[19570.2, 18864.7], Ts=[300.0, 400.0, 500.0], lambda12=0.1759,␣
→lambda21=0.7991)
[6818.90697, 1.105437, 62.6628, 2.01184, 1.181434, 137.6232]
```

7.27 Streams (thermo.stream)

class thermo.stream.**EnergyStream**(*Q*, *medium=None*)

Bases: object

Attributes

Hm

Q

energy

energy_calc

medium

Methods

copy	

Hm = None

Q = None

copy()

property energy

property energy_calc

medium = None

class thermo.stream.**EquilibriumStream**(*flasher*, *zs=None*, *ws=None*, *Vfls=None*, *Vfgs=None*, *ns=None*, *ms=None*, *Qls=None*, *Qgs=None*, *n=None*, *m=None*, *Q=None*, *T=None*, *P=None*, *VF=None*, *H=None*, *H_mass=None*, *S=None*, *S_mass=None*, *energy=None*, *Vf_TP=None*, *Q_TP=None*, *hot_start=None*, *existing_flash=None*)

Bases: *thermo.equilibrium.EquilibriumState*

Attributes

CASs CAS registration numbers for each component, [-].

Carcinogens Status of each component in cancer causing registries, [-].

Ceilings Ceiling exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

EnthalpySublimations Wrapper to obtain the list of EnthalpySublimations objects of the associated *PropertyCorrelationsPackage*.

EnthalpyVaporizations Wrapper to obtain the list of EnthalpyVaporizations objects of the associated *PropertyCorrelationsPackage*.

GWPs Global Warming Potentials for each component (impact/mass chemical)/(impact/mass CO2), [-].

Gfgs Ideal gas standard molar Gibbs free energy of formation for each component, [J/mol].

Gfgs_mass Ideal gas standard Gibbs free energy of formation for each component, [J/kg].

Hcs Higher standard molar heats of combustion for each component, [J/mol].

Hcs_lower Lower standard molar heats of combustion for each component, [J/mol].

Hcs_lower_mass Lower standard heats of combustion for each component, [J/kg].

Hcs_mass Higher standard heats of combustion for each component, [J/kg].

HeatCapacityGasMixture Wrapper to obtain the list of HeatCapacityGasMixture objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityGases Wrapper to obtain the list of HeatCapacityGases objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityLiquidMixture Wrapper to obtain the list of HeatCapacityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

HeatCapacityLiquids Wrapper to obtain the list of HeatCapacityLiquids objects of the associated *PropertyCorrelationsPackage*.

HeatCapacitySolidMixture Wrapper to obtain the list of HeatCapacitySolidMixture objects of the associated *PropertyCorrelationsPackage*.

HeatCapacitySolids Wrapper to obtain the list of HeatCapacitySolids objects of the associated *PropertyCorrelationsPackage*.

Hf_STPs Standard state molar enthalpies of formation for each component, [J/mol].

Hf_STPs_mass Standard state mass enthalpies of formation for each component, [J/kg].

Hfgs Ideal gas standard molar enthalpies of formation for each component, [J/mol].

Hfgs_mass Ideal gas standard enthalpies of formation for each component, [J/kg].

Hfus_Tms Molar heats of fusion for each component at their respective melting points, [J/mol].

Hfus_Tms_mass Heats of fusion for each component at their respective melting points, [J/kg].

Hsub_Tts Heats of sublimation for each component at their respective triple points, [J/mol].

Hsub_Tts_mass Heats of sublimation for each component at their respective triple points, [J/kg].

Hvap_298s Molar heats of vaporization for each component at 298.15 K, [J/mol].

Hvap_298s_mass Heats of vaporization for each component at 298.15 K, [J/kg].

Hvap_Tbs Molar heats of vaporization for each component at their respective normal boiling points, [J/mol].

Hvap_Tbs_mass Heats of vaporization for each component at their respective normal boiling points, [J/kg].

IDs Alias of CASs.

InChI_Keys InChI Keys for each component, [-].

InChIs InChI strings for each component, [-].

LF Method to return the liquid fraction of the equilibrium state.

LFLs Lower flammability limits for each component, [-].

MWs Similatiry variables for each component, [g/mol].

ODPs Ozone Depletion Potentials for each component (impact/mass chemical)/(impact/mass CFC-11), [-].

PSRK_groups PSRK subgroup: count groups for each component, [-].

P_calc

Parachors Parachors for each component, [N^0.25*m^2.75/mol].

Pcs Critical pressures for each component, [Pa].

PermittivityLiquids Wrapper to obtain the list of PermittivityLiquids objects of the associated *PropertyCorrelationsPackage*.

Psat_298s Vapor pressures for each component at 298.15 K, [Pa].

Pts Triple point pressures for each component, [Pa].

PubChems Pubchem IDs for each component, [-].

Q

Q_calc

Qgs

Qgs_calc

Qls

Qls_calc

RI_Ts Temperatures at which the refractive indexes were reported for each component, [K].

RIs Refractive indexes for each component, [-].

S0gs Ideal gas absolute molar entropies at 298.15 K at 1 atm for each component, [J/(mol*K)].

S0gs_mass Ideal gas absolute entropies at 298.15 K at 1 atm for each component, [J/(kg*K)].

STELs Short term exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

Sfgs Ideal gas standard molar entropies of formation for each component, [J/(mol*K)].

Sfgs_mass Ideal gas standard entropies of formation for each component, [J/(kg*K)].

Skins Whether each compound can be absorbed through the skin or not, [-].

StielPolars Stiel polar factors for each component, [-].

Stockmayers Lennard-Jones Stockmayer parameters (depth of potential-energy minimum over k) for each component, [K].

SublimationPressures Wrapper to obtain the list of SublimationPressures objects of the associated *PropertyCorrelationsPackage*.

SurfaceTensionMixture Wrapper to obtain the list of SurfaceTensionMixture objects of the associated *PropertyCorrelationsPackage*.

SurfaceTensions Wrapper to obtain the list of SurfaceTensions objects of the associated *PropertyCorrelationsPackage*.

TWAs Time-weighted average exposure limits to chemicals (and their units; ppm or mg/m^3), [various].

T_calc

Tautoignitions Autoignition temperatures for each component, [K].

Tbs Boiling temperatures for each component, [K].

Tcs Critical temperatures for each component, [K].

Tflashs Flash point temperatures for each component, [K].

ThermalConductivityGasMixture Wrapper to obtain the list of ThermalConductivityGasMixture objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityGases Wrapper to obtain the list of ThermalConductivityGases objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityLiquidMixture Wrapper to obtain the list of ThermalConductivityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

ThermalConductivityLiquids Wrapper to obtain the list of ThermalConductivityLiquids objects of the associated *PropertyCorrelationsPackage*.

Tms Melting temperatures for each component, [K].

Tts Triple point temperatures for each component, [K].

UFLs Upper flammability limits for each component, [-].

UNIFAC_Dortmund_groups UNIFAC_Dortmund_group: count groups for each component, [-].

UNIFAC_Qs UNIFAC *Q* parameters for each component, [-].

UNIFAC_Rs UNIFAC *R* parameters for each component, [-].

UNIFAC_groups UNIFAC_group: count groups for each component, [-].

VF Method to return the vapor fraction of the equilibrium state.

VF_calc

Van_der_Waals_areas Unnormalized Van der Waals areas for each component, [m^2/mol].

Van_der_Waals_volumes Unnormalized Van der Waals volumes for each component, [m^3/mol].

VaporPressures Wrapper to obtain the list of VaporPressures objects of the associated *PropertyCorrelationsPackage*.

Vcs Critical molar volumes for each component, [m^3/mol].

ViscosityGasMixture Wrapper to obtain the list of ViscosityGasMixture objects of the associated *PropertyCorrelationsPackage*.

ViscosityGases Wrapper to obtain the list of ViscosityGases objects of the associated *PropertyCorrelationsPackage*.

ViscosityLiquidMixture Wrapper to obtain the list of ViscosityLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

ViscosityLiquids Wrapper to obtain the list of ViscosityLiquids objects of the associated *PropertyCorrelationsPackage*.

Vmg_STPs Gas molar volumes for each component at STP; metastable if normally another state, [m^3/mol].

Vml_60Fs Liquid molar volumes for each component at 60 °F, [m^3/mol].

Vml_STPs Liquid molar volumes for each component at STP, [m^3/mol].

Vml_Tms Liquid molar volumes for each component at their respective melting points, [m^3/mol].

Vms_Tms Solid molar volumes for each component at their respective melting points, [m^3/mol].

VolumeGasMixture Wrapper to obtain the list of VolumeGasMixture objects of the associated *PropertyCorrelationsPackage*.

VolumeGases Wrapper to obtain the list of VolumeGases objects of the associated *PropertyCorrelationsPackage*.

VolumeLiquidMixture Wrapper to obtain the list of VolumeLiquidMixture objects of the associated *PropertyCorrelationsPackage*.

VolumeLiquids Wrapper to obtain the list of VolumeLiquids objects of the associated *PropertyCorrelationsPackage*.

VolumeSolidMixture Wrapper to obtain the list of VolumeSolidMixture objects of the associated *PropertyCorrelationsPackage*.

VolumeSolids Wrapper to obtain the list of VolumeSolids objects of the associated *PropertyCorrelationsPackage*.

Zcs Critical compressibilities for each component, [-].

atomss Breakdown of each component into its elements and their counts, as a dict, [-].

betas_liquids Method to calculate and return the fraction of the liquid phase that each liquid phase is, by molar phase fraction.

betas_mass Method to calculate and return the mass fraction of all of the phases in the system.

betas_mass_liquids Method to calculate and return the fraction of the liquid phase that each liquid phase is, by mass phase fraction.

betas_mass_states Method to return the mass phase fractions of each of the three fundamental *types* of phases.

betas_states Method to return the molar phase fractions of each of the three fundamental *types* of phases.

betas_volume Method to calculate and return the volume fraction of all of the phases in the system.

betas_volume_liquids Method to calculate and return the fraction of the liquid phase that each liquid phase is, by volume phase fraction.

betas_volume_states Method to return the volume phase fractions of each of the three fundamental *types* of phases.

charges Charge number (valence) for each component, [-].

composition_specified Always needs a composition

conductivities Electrical conductivities for each component, [S/m].

conductivity_Ts Temperatures at which the electrical conductivities for each component were measured, [K].

dipoles Dipole moments for each component, [debye].

economic_statuses Status of each component in in relation to import and export from various regions, [-].

energy

energy_calc

energy_reactive

energy_reactive_calc

flow_specified Always needs a flow specified

formulas Formulas of each component, [-].

heaviest_liquid The liquid-like phase with the highest mass density, [-]

legal_statuses Status of each component in in relation to import and export rules from various regions, [-].

lightest_liquid The liquid-like phase with the lowest mass density, [-]

liquid_bulk

logPs Octanol-water partition coefficients for each component, [-].

molecular_diameters Lennard-Jones molecular diameters for each component, [angstrom].

ms_calc

n_calc

names Names for each component, [-].

non_pressure_spec_specified Cannot have a stream without an energy-type spec.

ns_calc

omegas Acentric factors for each component, [-].

phase Method to calculate and return a string representing the phase of the mixture.

phase_STPs Standard states ('g', 'l', or 's') for each component, [-].

property_package

quality Method to return the mass vapor fraction of the equilibrium state.

rhocs Molar densities at the critical point for each component, [mol/m^3].

rhocs_mass Densities at the critical point for each component, [kg/m^3].

rhog_STPs Molar gas densities at STP for each component; metastable if normally another state, [mol/m^3].

rhog_STPs_mass Gas densities at STP for each component; metastable if normally another state, [kg/m^3].

rhol_60Fs Liquid molar densities for each component at 60 °F, [mol/m^3].

rhol_60Fs_mass Liquid mass densities for each component at 60 °F, [kg/m^3].

rhol_STPs Molar liquid densities at STP for each component, [mol/m^3].

rhol_STPs_mass Liquid densities at STP for each component, [kg/m^3].

rhos_Tms Solid molar densities for each component at their respective melting points, [mol/m^3].

rhos_Tms_mass Solid mass densities for each component at their melting point, [kg/m^3].

sigma_STPs Liquid-air surface tensions at 298.15 K and the higher of 101325 Pa or the saturation pressure, [N/m].

sigma_Tbs Liquid-air surface tensions at the normal boiling point and 101325 Pa, [N/m].

sigma_Tms Liquid-air surface tensions at the melting point and 101325 Pa, [N/m].

similarity_variables Similarity variables for each component, [mol/g].

smiless SMILES identifiers for each component, [-].

solid_bulk

solubility_parameters Solubility parameters for each component at 298.15 K, [Pa^0.5].

specified_composition_vars number of composition variables

specified_flow_vars Always needs only one flow specified

specified_state_vars Always needs two states

state_specified Always needs a state

state_specs Returns a list of tuples of (state_variable, state_value) representing the thermodynamic state of the system.

water_index The index of the component water in the components.

water_phase The liquid-like phase with the highest water mole fraction, [-]

water_phase_index The liquid-like phase with the highest mole fraction of water, [-]

zs_calc

Methods

A()	Method to calculate and return the Helmholtz energy of the phase.
API([phase])	Method to calculate and return the API of the phase.
A_dep()	Method to calculate and return the departure Helmholtz energy of the phase.

continues on next page

Table 90 – continued from previous page

A_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas Helmholtz energy of formation of the phase (as if the phase was an ideal gas).
A_ideal_gas([phase])	Method to calculate and return the ideal-gas Helmholtz energy of the phase.
A_mass([phase])	Method to calculate and return mass Helmholtz energy of the phase.
A_reactive()	Method to calculate and return the Helmholtz free energy of the phase on a reactive basis.
Bvirial([phase])	Method to calculate and return the *B* virial coefficient of the phase at its current conditions.
Cp()	Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase.
Cp_Cv_ratio()	Method to calculate and return the Cp/Cv ratio of the phase.
Cp_Cv_ratio_ideal_gas([phase])	Method to calculate and return the ratio of the ideal-gas heat capacity to its constant-volume heat capacity.
Cp_dep([phase])	Method to calculate and return the difference between the actual *Cp* and the ideal-gas heat capacity C_p^{ig} of the phase.
Cp_ideal_gas([phase])	Method to calculate and return the ideal-gas heat capacity of the phase.
Cp_mass([phase])	Method to calculate and return mass constant pressure heat capacity of the phase.
Cv()	Method to calculate and return the constant-volume heat capacity *Cv* of the phase.
Cv_dep([phase])	Method to calculate and return the difference between the actual *Cv* and the ideal-gas constant volume heat capacity C_v^{ig} of the phase.
Cv_ideal_gas([phase])	Method to calculate and return the ideal-gas constant volume heat capacity of the phase.
Cv_mass([phase])	Method to calculate and return mass constant volume heat capacity of the phase.
G()	Method to calculate and return the Gibbs free energy of the phase.
G_dep()	Method to calculate and return the departure Gibbs free energy of the phase.
G_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas Gibbs free energy of formation of the phase (as if the phase was an ideal gas).
G_ideal_gas([phase])	Method to calculate and return the ideal-gas Gibbs free energy of the phase.
G_mass([phase])	Method to calculate and return mass Gibbs energy of the phase.
G_reactive()	Method to calculate and return the Gibbs free energy of the phase on a reactive basis.
H()	Method to calculate and return the constant-temperature and constant phase-fraction enthalpy of the bulk phase.

continues on next page

Table 90 – continued from previous page

`H_C_ratio`([phase])	Method to calculate and return the atomic ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.
`H_C_ratio_mass`([phase])	Method to calculate and return the mass ratio of hydrogen atoms to carbon atoms, based on the current composition of the phase.
`H_dep`([phase])	Method to calculate and return the difference between the actual H and the ideal-gas enthalpy of the phase.
`H_formation_ideal_gas`([phase])	Method to calculate and return the ideal-gas enthalpy of formation of the phase (as if the phase was an ideal gas).
`H_ideal_gas`([phase])	Method to calculate and return the ideal-gas enthalpy of the phase.
`H_mass`([phase])	Method to calculate and return mass enthalpy of the phase.
`H_reactive`()	Method to calculate and return the constant-temperature and constant phase-fraction reactive enthalpy of the bulk phase.
`Hc`([phase])	Method to calculate and return the molar ideal-gas higher heat of combustion of the object, [J/mol]
`Hc_lower`([phase])	Method to calculate and return the molar ideal-gas lower heat of combustion of the object, [J/mol]
`Hc_lower_mass`([phase])	Method to calculate and return the mass ideal-gas lower heat of combustion of the object, [J/mol]
`Hc_lower_normal`([phase])	Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the normal gas volume, [J/m^3]
`Hc_lower_standard`([phase])	Method to calculate and return the volumetric ideal-gas lower heat of combustion of the object using the standard gas volume, [J/m^3]
`Hc_mass`([phase])	Method to calculate and return the mass ideal-gas higher heat of combustion of the object, [J/mol]
`Hc_normal`([phase])	Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the normal gas volume, [J/m^3]
`Hc_standard`([phase])	Method to calculate and return the volumetric ideal-gas higher heat of combustion of the object using the standard gas volume, [J/m^3]
`Joule_Thomson`()	Method to calculate and return the Joule-Thomson coefficient of the bulk according to the selected calculation methodology.
`Ks`(phase[, phase_ref])	Method to calculate and return the K-values of each phase.
`MW`([phase])	Method to calculate and return the molecular weight of the phase.
`PIP`()	Method to calculate and return the phase identification parameter of the phase.
`Pmc`([phase])	Method to calculate and return the mechanical critical pressure of the phase.

continues on next page

Table 90 – continued from previous page

S()	Method to calculate and return the constant-temperature and constant phase-fraction entropy of the bulk phase.
SG([phase])	Method to calculate and return the standard liquid specific gravity of the phase, using constant liquid pure component densities not calculated by the phase object, at 60 °F.
SG_gas([phase])	Method to calculate and return the specific gravity of the phase with respect to a gas reference density.
S_dep([phase])	Method to calculate and return the difference between the actual *S* and the ideal-gas entropy of the phase.
S_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas entropy of formation of the phase (as if the phase was an ideal gas).
S_ideal_gas([phase])	Method to calculate and return the ideal-gas entropy of the phase.
S_mass([phase])	Method to calculate and return mass entropy of the phase.
S_reactive()	Method to calculate and return the constant-temperature and constant phase-fraction reactive entropy of the bulk phase.
StreamArgs()	Goal to create a StreamArgs instance, with the user specified variables always being here.
Tmc([phase])	Method to calculate and return the mechanical critical temperature of the phase.
U()	Method to calculate and return the internal energy of the phase.
U_dep()	Method to calculate and return the departure internal energy of the phase.
U_formation_ideal_gas([phase])	Method to calculate and return the ideal-gas internal energy of formation of the phase (as if the phase was an ideal gas).
U_ideal_gas([phase])	Method to calculate and return the ideal-gas internal energy of the phase.
U_mass([phase])	Method to calculate and return mass internal energy of the phase.
U_reactive()	Method to calculate and return the internal energy of the phase on a reactive basis.
V()	Method to calculate and return the molar volume of the bulk phase.
V_dep()	Method to calculate and return the departure (from ideal gas behavior) molar volume of the phase.
V_gas([phase])	Method to calculate and return the ideal-gas molar volume of the phase at the chosen reference temperature and pressure, according to the temperature variable *T_gas_ref* and pressure variable *P_gas_ref* of the *thermo.bulk.BulkSettings*.

continues on next page

Table 90 – continued from previous page

`V_gas_normal`([phase])	Method to calculate and return the ideal-gas molar volume of the phase at the normal temperature and pressure, according to the temperature variable *T_normal* and pressure variable *P_normal* of the `thermo.bulk.BulkSettings`.
`V_gas_standard`([phase])	Method to calculate and return the ideal-gas molar volume of the phase at the standard temperature and pressure, according to the temperature variable *T_standard* and pressure variable *P_standard* of the `thermo.bulk.BulkSettings`.
`V_ideal_gas`([phase])	Method to calculate and return the ideal-gas molar volume of the phase.
`V_iter`([phase, force])	Method to calculate and return the volume of the phase in a way suitable for a TV resolution to converge on the same pressure.
`V_liquid_ref`([phase])	Method to calculate and return the liquid reference molar volume according to the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings` and the composition of the phase.
`V_liquids_ref`()	Method to calculate and return the liquid reference molar volumes according to the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings`.
`V_mass`([phase])	Method to calculate and return the specific volume of the phase.
`Vfgs`([phase])	Method to calculate and return the ideal-gas volume fractions of the components of the phase.
`Vfls`([phase])	Method to calculate and return the ideal-liquid volume fractions of the components of the phase, using the standard liquid densities at the temperature variable *T_liquid_volume_ref* of `thermo.bulk.BulkSettings` and the composition of the phase.
`Vmc`([phase])	Method to calculate and return the mechanical critical volume of the phase.
`Wobbe_index`([phase])	Method to calculate and return the molar Wobbe index of the object, [J/mol].
`Wobbe_index_lower`([phase])	Method to calculate and return the molar lower Wobbe index of the
`Wobbe_index_lower_mass`([phase])	Method to calculate and return the lower mass Wobbe index of the object, [J/kg].
`Wobbe_index_lower_normal`([phase])	Method to calculate and return the volumetric normal lower Wobbe index of the object, [J/m^3].
`Wobbe_index_lower_standard`([phase])	Method to calculate and return the volumetric standard lower Wobbe index of the object, [J/m^3].
`Wobbe_index_mass`([phase])	Method to calculate and return the mass Wobbe index of the object, [J/kg].
`Wobbe_index_normal`([phase])	Method to calculate and return the volumetric normal Wobbe index of the object, [J/m^3].
`Wobbe_index_standard`([phase])	Method to calculate and return the volumetric standard Wobbe index of the object, [J/m^3].

continues on next page

Table 90 – continued from previous page

Z()	Method to calculate and return the compressibility factor of the phase.
Zmc([phase])	Method to calculate and return the mechanical critical compressibility of the phase.
alpha([phase])	Method to calculate and return the thermal diffusivity of the equilibrium state.
atom_fractions([phase])	Method to calculate and return the atomic composition of the phase; returns a dictionary of atom fraction (by count), containing only those elements who are present.
atom_mass_fractions([phase])	Method to calculate and return the atomic mass fractions of the phase; returns a dictionary of atom fraction (by mass), containing only those elements who are present.
d2P_dT2()	Method to calculate and return the second temperature derivative of pressure of the bulk according to the selected calculation methodology.
d2P_dT2_frozen()	Method to calculate and return the second constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions.
d2P_dTdV()	Method to calculate and return the second derivative of pressure with respect to temperature and volume of the bulk according to the selected calculation methodology.
d2P_dTdV_frozen()	Method to calculate and return the second derivative of pressure with respect to volume and temperature of the bulk phase, at constant phase fractions and phase compositions.
d2P_dV2()	Method to calculate and return the second volume derivative of pressure of the bulk according to the selected calculation methodology.
d2P_dV2_frozen()	Method to calculate and return the constant-temperature second derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions.
dA_dP()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.
dA_dP_T()	Method to calculate and return the constant-temperature pressure derivative of Helmholtz energy.
dA_dP_V()	Method to calculate and return the constant-volume pressure derivative of Helmholtz energy.
dA_dT()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dA_dT_P()	Method to calculate and return the constant-pressure temperature derivative of Helmholtz energy.
dA_dT_V()	Method to calculate and return the constant-volume temperature derivative of Helmholtz energy.

continues on next page

Table 90 – continued from previous page

dA_dV_P()	Method to calculate and return the constant-pressure volume derivative of Helmholtz energy.
dA_dV_T()	Method to calculate and return the constant-temperature volume derivative of Helmholtz energy.
dA_mass_dP()	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.
dA_mass_dP_T()	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant temperature.
dA_mass_dP_V()	Method to calculate and return the pressure derivative of mass Helmholtz energy of the phase at constant volume.
dA_mass_dT()	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dT_P()	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dT_V()	Method to calculate and return the temperature derivative of mass Helmholtz energy of the phase at constant volume.
dA_mass_dV_P()	Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant pressure.
dA_mass_dV_T()	Method to calculate and return the volume derivative of mass Helmholtz energy of the phase at constant temperature.
dCv_dP_T()	Method to calculate the pressure derivative of Cv, constant volume heat capacity, at constant temperature.
dCv_dT_P()	Method to calculate the temperature derivative of Cv, constant volume heat capacity, at constant pressure.
dCv_mass_dP_T()	Method to calculate and return the pressure derivative of mass Constant-volume heat capacity of the phase at constant temperature.
dCv_mass_dT_P()	Method to calculate and return the temperature derivative of mass Constant-volume heat capacity of the phase at constant pressure.
dG_dP()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dP_T()	Method to calculate and return the constant-temperature pressure derivative of Gibbs free energy.
dG_dP_V()	Method to calculate and return the constant-volume pressure derivative of Gibbs free energy.
dG_dT()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.
dG_dT_P()	Method to calculate and return the constant-pressure temperature derivative of Gibbs free energy.

continues on next page

Table 90 – continued from previous page

dG_dT_V()	Method to calculate and return the constant-volume temperature derivative of Gibbs free energy.
dG_dV_P()	Method to calculate and return the constant-pressure volume derivative of Gibbs free energy.
dG_dV_T()	Method to calculate and return the constant-temperature volume derivative of Gibbs free energy.
dG_mass_dP()	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.
dG_mass_dP_T()	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant temperature.
dG_mass_dP_V()	Method to calculate and return the pressure derivative of mass Gibbs free energy of the phase at constant volume.
dG_mass_dT()	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dT_P()	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dT_V()	Method to calculate and return the temperature derivative of mass Gibbs free energy of the phase at constant volume.
dG_mass_dV_P()	Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant pressure.
dG_mass_dV_T()	Method to calculate and return the volume derivative of mass Gibbs free energy of the phase at constant temperature.
dH_dP()	Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.
dH_dP_T()	Method to calculate and return the pressure derivative of enthalpy of the phase at constant pressure.
dH_dT()	Method to calculate and return the constant-temperature and constant phase-fraction heat capacity of the bulk phase.
dH_dT_P()	Method to calculate and return the temperature derivative of enthalpy of the phase at constant pressure.
dH_mass_dP()	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.
dH_mass_dP_T()	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant temperature.
dH_mass_dP_V()	Method to calculate and return the pressure derivative of mass enthalpy of the phase at constant volume.
dH_mass_dT()	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.

continues on next page

Table 90 – continued from previous page

dH_mass_dT_P()	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dT_V()	Method to calculate and return the temperature derivative of mass enthalpy of the phase at constant volume.
dH_mass_dV_P()	Method to calculate and return the volume derivative of mass enthalpy of the phase at constant pressure.
dH_mass_dV_T()	Method to calculate and return the volume derivative of mass enthalpy of the phase at constant temperature.
dP_dP_A()	Method to calculate and return the pressure derivative of pressure of the phase at constant Helmholtz energy.
dP_dP_G()	Method to calculate and return the pressure derivative of pressure of the phase at constant Gibbs energy.
dP_dP_H()	Method to calculate and return the pressure derivative of pressure of the phase at constant enthalpy.
dP_dP_S()	Method to calculate and return the pressure derivative of pressure of the phase at constant entropy.
dP_dP_U()	Method to calculate and return the pressure derivative of pressure of the phase at constant internal energy.
dP_dT()	Method to calculate and return the first temperature derivative of pressure of the bulk according to the selected calculation methodology.
dP_dT_A()	Method to calculate and return the temperature derivative of pressure of the phase at constant Helmholtz energy.
dP_dT_G()	Method to calculate and return the temperature derivative of pressure of the phase at constant Gibbs energy.
dP_dT_H()	Method to calculate and return the temperature derivative of pressure of the phase at constant enthalpy.
dP_dT_S()	Method to calculate and return the temperature derivative of pressure of the phase at constant entropy.
dP_dT_U()	Method to calculate and return the temperature derivative of pressure of the phase at constant internal energy.
dP_dT_frozen()	Method to calculate and return the constant-volume derivative of pressure with respect to temperature of the bulk phase, at constant phase fractions and phase compositions.
dP_dV()	Method to calculate and return the first volume derivative of pressure of the bulk according to the selected calculation methodology.
dP_dV_A()	Method to calculate and return the volume derivative of pressure of the phase at constant Helmholtz energy.
dP_dV_G()	Method to calculate and return the volume derivative of pressure of the phase at constant Gibbs energy.

continues on next page

Table 90 – continued from previous page

dP_dV_H()	Method to calculate and return the volume derivative of pressure of the phase at constant enthalpy.
dP_dV_S()	Method to calculate and return the volume derivative of pressure of the phase at constant entropy.
dP_dV_U()	Method to calculate and return the volume derivative of pressure of the phase at constant internal energy.
dP_dV_frozen()	Method to calculate and return the constant-temperature derivative of pressure with respect to volume of the bulk phase, at constant phase fractions and phase compositions.
dP_drho_A()	Method to calculate and return the density derivative of pressure of the phase at constant Helmholtz energy.
dP_drho_G()	Method to calculate and return the density derivative of pressure of the phase at constant Gibbs energy.
dP_drho_H()	Method to calculate and return the density derivative of pressure of the phase at constant enthalpy.
dP_drho_S()	Method to calculate and return the density derivative of pressure of the phase at constant entropy.
dP_drho_U()	Method to calculate and return the density derivative of pressure of the phase at constant internal energy.
dS_dP()	Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.
dS_dP_T()	Method to calculate and return the pressure derivative of entropy of the phase at constant pressure.
dS_dV_P()	Method to calculate and return the volume derivative of entropy of the phase at constant pressure.
dS_dV_T()	Method to calculate and return the volume derivative of entropy of the phase at constant temperature.
dS_mass_dP()	Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.
dS_mass_dP_T()	Method to calculate and return the pressure derivative of mass entropy of the phase at constant temperature.
dS_mass_dP_V()	Method to calculate and return the pressure derivative of mass entropy of the phase at constant volume.
dS_mass_dT()	Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.
dS_mass_dT_P()	Method to calculate and return the temperature derivative of mass entropy of the phase at constant pressure.
dS_mass_dT_V()	Method to calculate and return the temperature derivative of mass entropy of the phase at constant volume.
dS_mass_dV_P()	Method to calculate and return the volume derivative of mass entropy of the phase at constant pressure.
dS_mass_dV_T()	Method to calculate and return the volume derivative of mass entropy of the phase at constant temperature.
dT_dP_A()	Method to calculate and return the pressure derivative of temperature of the phase at constant Helmholtz energy.

continues on next page

Table 90 – continued from previous page

dT_dP_G()	Method to calculate and return the pressure derivative of temperature of the phase at constant Gibbs energy.
dT_dP_H()	Method to calculate and return the pressure derivative of temperature of the phase at constant enthalpy.
dT_dP_S()	Method to calculate and return the pressure derivative of temperature of the phase at constant entropy.
dT_dP_U()	Method to calculate and return the pressure derivative of temperature of the phase at constant internal energy.
dT_dT_A()	Method to calculate and return the temperature derivative of temperature of the phase at constant Helmholtz energy.
dT_dT_G()	Method to calculate and return the temperature derivative of temperature of the phase at constant Gibbs energy.
dT_dT_H()	Method to calculate and return the temperature derivative of temperature of the phase at constant enthalpy.
dT_dT_S()	Method to calculate and return the temperature derivative of temperature of the phase at constant entropy.
dT_dT_U()	Method to calculate and return the temperature derivative of temperature of the phase at constant internal energy.
dT_dV_A()	Method to calculate and return the volume derivative of temperature of the phase at constant Helmholtz energy.
dT_dV_G()	Method to calculate and return the volume derivative of temperature of the phase at constant Gibbs energy.
dT_dV_H()	Method to calculate and return the volume derivative of temperature of the phase at constant enthalpy.
dT_dV_S()	Method to calculate and return the volume derivative of temperature of the phase at constant entropy.
dT_dV_U()	Method to calculate and return the volume derivative of temperature of the phase at constant internal energy.
dT_drho_A()	Method to calculate and return the density derivative of temperature of the phase at constant Helmholtz energy.
dT_drho_G()	Method to calculate and return the density derivative of temperature of the phase at constant Gibbs energy.
dT_drho_H()	Method to calculate and return the density derivative of temperature of the phase at constant enthalpy.
dT_drho_S()	Method to calculate and return the density derivative of temperature of the phase at constant entropy.
dT_drho_U()	Method to calculate and return the density derivative of temperature of the phase at constant internal energy.
dU_dP()	Method to calculate and return the constant-temperature pressure derivative of internal energy.

continues on next page

Table 90 – continued from previous page

dU_dP_T()	Method to calculate and return the constant-temperature pressure derivative of internal energy.
dU_dP_V()	Method to calculate and return the constant-volume pressure derivative of internal energy.
dU_dT()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
dU_dT_P()	Method to calculate and return the constant-pressure temperature derivative of internal energy.
dU_dT_V()	Method to calculate and return the constant-volume temperature derivative of internal energy.
dU_dV_P()	Method to calculate and return the constant-pressure volume derivative of internal energy.
dU_dV_T()	Method to calculate and return the constant-temperature volume derivative of internal energy.
dU_mass_dP()	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.
dU_mass_dP_T()	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant temperature.
dU_mass_dP_V()	Method to calculate and return the pressure derivative of mass internal energy of the phase at constant volume.
dU_mass_dT()	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.
dU_mass_dT_P()	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant pressure.
dU_mass_dT_V()	Method to calculate and return the temperature derivative of mass internal energy of the phase at constant volume.
dU_mass_dV_P()	Method to calculate and return the volume derivative of mass internal energy of the phase at constant pressure.
dU_mass_dV_T()	Method to calculate and return the volume derivative of mass internal energy of the phase at constant temperature.
dV_dP_A()	Method to calculate and return the pressure derivative of volume of the phase at constant Helmholtz energy.
dV_dP_G()	Method to calculate and return the pressure derivative of volume of the phase at constant Gibbs energy.
dV_dP_H()	Method to calculate and return the pressure derivative of volume of the phase at constant enthalpy.
dV_dP_S()	Method to calculate and return the pressure derivative of volume of the phase at constant entropy.
dV_dP_U()	Method to calculate and return the pressure derivative of volume of the phase at constant internal energy.
dV_dT_A()	Method to calculate and return the temperature derivative of volume of the phase at constant Helmholtz energy.

continues on next page

Table 90 – continued from previous page

dV_dT_G()	Method to calculate and return the temperature derivative of volume of the phase at constant Gibbs energy.
dV_dT_H()	Method to calculate and return the temperature derivative of volume of the phase at constant enthalpy.
dV_dT_S()	Method to calculate and return the temperature derivative of volume of the phase at constant entropy.
dV_dT_U()	Method to calculate and return the temperature derivative of volume of the phase at constant internal energy.
dV_dV_A()	Method to calculate and return the volume derivative of volume of the phase at constant Helmholtz energy.
dV_dV_G()	Method to calculate and return the volume derivative of volume of the phase at constant Gibbs energy.
dV_dV_H()	Method to calculate and return the volume derivative of volume of the phase at constant enthalpy.
dV_dV_S()	Method to calculate and return the volume derivative of volume of the phase at constant entropy.
dV_dV_U()	Method to calculate and return the volume derivative of volume of the phase at constant internal energy.
dV_drho_A()	Method to calculate and return the density derivative of volume of the phase at constant Helmholtz energy.
dV_drho_G()	Method to calculate and return the density derivative of volume of the phase at constant Gibbs energy.
dV_drho_H()	Method to calculate and return the density derivative of volume of the phase at constant enthalpy.
dV_drho_S()	Method to calculate and return the density derivative of volume of the phase at constant entropy.
dV_drho_U()	Method to calculate and return the density derivative of volume of the phase at constant internal energy.
drho_dP_A()	Method to calculate and return the pressure derivative of density of the phase at constant Helmholtz energy.
drho_dP_G()	Method to calculate and return the pressure derivative of density of the phase at constant Gibbs energy.
drho_dP_H()	Method to calculate and return the pressure derivative of density of the phase at constant enthalpy.
drho_dP_S()	Method to calculate and return the pressure derivative of density of the phase at constant entropy.
drho_dP_U()	Method to calculate and return the pressure derivative of density of the phase at constant internal energy.
drho_dT_A()	Method to calculate and return the temperature derivative of density of the phase at constant Helmholtz energy.
drho_dT_G()	Method to calculate and return the temperature derivative of density of the phase at constant Gibbs energy.
drho_dT_H()	Method to calculate and return the temperature derivative of density of the phase at constant enthalpy.

continues on next page

Table 90 – continued from previous page

drho_dT_S()	Method to calculate and return the temperature derivative of density of the phase at constant entropy.
drho_dT_U()	Method to calculate and return the temperature derivative of density of the phase at constant internal energy.
drho_dV_A()	Method to calculate and return the volume derivative of density of the phase at constant Helmholtz energy.
drho_dV_G()	Method to calculate and return the volume derivative of density of the phase at constant Gibbs energy.
drho_dV_H()	Method to calculate and return the volume derivative of density of the phase at constant enthalpy.
drho_dV_S()	Method to calculate and return the volume derivative of density of the phase at constant entropy.
drho_dV_U()	Method to calculate and return the volume derivative of density of the phase at constant internal energy.
drho_drho_A()	Method to calculate and return the density derivative of density of the phase at constant Helmholtz energy.
drho_drho_G()	Method to calculate and return the density derivative of density of the phase at constant Gibbs energy.
drho_drho_H()	Method to calculate and return the density derivative of density of the phase at constant enthalpy.
drho_drho_S()	Method to calculate and return the density derivative of density of the phase at constant entropy.
drho_drho_U()	Method to calculate and return the density derivative of density of the phase at constant internal energy.
isentropic_exponent()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.
isentropic_exponent_PT()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $P^{(1-k)}T^k = \text{const}$.
isentropic_exponent_PV()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $PV^k = \text{const}$.
isentropic_exponent_TV()	Method to calculate and return the real gas isentropic exponent of the phase, which satisfies the relationship $TV^{k-1} = \text{const}$.
isobaric_expansion()	Method to calculate and return the isobatic expansion coefficient of the bulk according to the selected calculation methodology.
isothermal_bulk_modulus()	Method to calculate and return the isothermal bulk modulus of the phase.
k()	Calculate and return the thermal conductivity of the bulk according to the selected thermal conductivity settings in BulkSettings, the settings in *ThermalConductivityGasMixture* and *ThermalConductivityLiquidMixture*, and the configured pure-component settings in *ThermalConductivityGas* and *ThermalConductivityLiquid*.

continues on next page

Table 90 – continued from previous page

kappa()	Method to calculate and return the isothermal compressibility of the bulk according to the selected calculation methodology.
log_zs()	Method to calculate and return the log of mole fractions specified.
molar_water_content([phase])	Method to calculate and return the molar water content; this is the g/mol of the fluid which is coming from water, [g/mol].
mu()	Calculate and return the viscosity of the bulk according to the selected viscosity settings in BulkSettings, the settings in *ViscosityGasMixture* and *ViscosityLiquidMixture*, and the configured pure-component settings in *ViscosityGas* and *ViscosityLiquid*.
nu([phase])	Method to calculate and return the kinematic viscosity of the equilibrium state.
pseudo_Pc([phase])	Method to calculate and return the pseudocritical pressure calculated using Kay's rule (linear mole fractions):
pseudo_Tc([phase])	Method to calculate and return the pseudocritical temperature calculated using Kay's rule (linear mole fractions):
pseudo_Vc([phase])	Method to calculate and return the pseudocritical volume calculated using Kay's rule (linear mole fractions):
pseudo_Zc([phase])	Method to calculate and return the pseudocritical compressibility calculated using Kay's rule (linear mole fractions):
rho()	Method to calculate and return the molar density of the phase.
rho_mass([phase])	Method to calculate and return mass density of the phase.
rho_mass_liquid_ref([phase])	Method to calculate and return the liquid reference mass density according to the temperature variable *T_liquid_volume_ref* of *thermo.bulk.BulkSettings* and the composition of the phase.
sigma()	Calculate and return the surface tension of the bulk according to the selected surface tension settings in BulkSettings, the settings in *SurfaceTensionMixture* and the configured pure-component settings in *SurfaceTension*.
speed_of_sound()	Method to calculate and return the molar speed of sound of the bulk according to the selected calculation methodology.
speed_of_sound_mass()	Method to calculate and return the speed of sound of the phase.
value(name[, phase])	Method to retrieve a property from a string.
ws([phase])	Method to calculate and return the mass fractions of the phase, [-]

continues on next page

Table 90 – continued from previous page

ws_no_water([phase])	Method to calculate and return the mass fractions of all species in the phase, normalized to a water-free basis (the mass fraction of water returned is zero).
zs_no_water([phase])	Method to calculate and return the mole fractions of all species in the phase, normalized to a water-free basis (the mole fraction of water returned is zero).

dH_dP_V	
dH_dT_V	
dH_dV_P	
dH_dV_T	
dS_dP_V	
dS_dT	
dS_dT_P	
dS_dT_V	

property P_calc

property Q

property Q_calc

property Qgs

property Qgs_calc

property Qls

property Qls_calc

StreamArgs()

Goal to create a StreamArgs instance, with the user specified variables always being here.

The state variables are currently correctly tracked. The flow rate and composition variable needs to be tracked as a function of what was specified as the input variables.

The flow rate needs to be changed wen the stream flow rate is changed. Note this stores unnormalized specs, but that this is OK.

property T_calc

property VF_calc

property composition_specified

Always needs a composition

property energy

property energy_calc

property energy_reactive

property energy_reactive_calc

flashed = True

property flow_specified

Always needs a flow specified

property ms_calc

property n_calc

property non_pressure_spec_specified
Cannot have a stream without an energy-type spec.

property ns_calc

property property_package

property specified_composition_vars
number of composition variables

property specified_flow_vars
Always needs only one flow specified

property specified_state_vars
Always needs two states

property state_specified
Always needs a state

property state_specs
Returns a list of tuples of (state_variable, state_value) representing the thermodynamic state of the system.

property zs_calc

class thermo.stream.Stream(*IDs=None*, *zs=None*, *ws=None*, *Vfls=None*, *Vfgs=None*, *ns=None*, *ms=None*, *Qls=None*, *Qgs=None*, *n=None*, *m=None*, *Q=None*, *T=None*, *P=None*, *VF=None*, *H=None*, *Hm=None*, *S=None*, *Sm=None*, *energy=None*, *pkg=None*, *Vf_TP=(None, None)*, *Q_TP=(None, None, '')*)
Bases: `thermo.mixture.Mixture`

Creates a Stream object which is useful for modeling mass and energy balances.

Streams have five variables. The flow rate, composition, and components are mandatory; and two of the variables temperature, pressure, vapor fraction, enthalpy, or entropy are required. Entropy and enthalpy may also be provided in a molar basis; energy can also be provided, which when combined with either a flow rate or enthalpy will calculate the other variable.

The composition and flow rate may be specified together or separately. The options for specifying them are:

- Mole fractions *zs*
- Mass fractions *ws*
- Liquid standard volume fractions *Vfls*
- Gas standard volume fractions *Vfgs*
- Mole flow rates *ns*
- Mass flow rates *ms*
- Liquid flow rates *Qls* (based on pure component volumes at the T and P specified by *Q_TP*)
- Gas flow rates *Qgs* (based on pure component volumes at the T and P specified by *Q_TP*)

If only the composition is specified by providing any of *zs*, *ws*, *Vfls* or *Vfgs*, the flow rate must be specified by providing one of these:

- Mole flow rate *n*
- Mass flow rate *m*
- Volumetric flow rate *Q* at the provided T and P or if specified, *Q_TP*
- Energy *energy*

The state variables must be two of the following. Not all combinations result in a supported flash.

- Tempetarure *T*
- Pressure *P*
- Vapor fraction *VF*
- Enthalpy *H*
- Molar enthalpy *Hm*
- Entropy *S*
- Molar entropy *Sm*
- Energy *energy*

Parameters

IDs [list, optional] List of chemical identifiers - names, CAS numbers, SMILES or InChi strings can all be recognized and may be mixed [-]

zs [list or dict, optional] Mole fractions of all components in the stream [-]

ws [list or dict, optional] Mass fractions of all components in the stream [-]

Vfls [list or dict, optional] Volume fractions of all components as a hypothetical liquid phase based on pure component densities [-]

Vfgs [list or dict, optional] Volume fractions of all components as a hypothetical gas phase based on pure component densities [-]

ns [list or dict, optional] Mole flow rates of each component [mol/s]

ms [list or dict, optional] Mass flow rates of each component [kg/s]

Qls [list or dict, optional] Liquid flow rates of all components as a hypothetical liquid phase based on pure component densities [m^3/s]

Qgs [list or dict, optional] Gas flow rates of all components as a hypothetical gas phase based on pure component densities [m^3/s]

n [float, optional] Total mole flow rate of all components in the stream [mol/s]

m [float, optional] Total mass flow rate of all components in the stream [kg/s]

Q [float, optional] Total volumetric flow rate of all components in the stream based on the temperature and pressure specified by *T* and *P* [m^3/s]

T [float, optional] Temperature of the stream (default 298.15 K), [K]

P [float, optional] Pressure of the stream (default 101325 Pa) [Pa]

VF [float, optional] Vapor fraction (mole basis) of the stream, [-]

H [float, optional] Mass enthalpy of the stream, [J]

Hm [float, optional] Molar enthalpy of the stream, [J/mol]

S [float, optional] Mass entropy of the stream, [J/kg/K]

Sm [float, optional] Molar entropy of the stream, [J/mol/K]

energy [float, optional] Flowing energy of the stream (*H`*`m*), [W]

pkg [object] The thermodynamic property package to use for flash calculations; one of the caloric packages in `thermo.property_package`; defaults to the ideal model [-]

Vf_TP [tuple(2, float), optional] The (T, P) at which the volume fractions are specified to be at, [K] and [Pa]

Q_TP [tuple(3, float, float, str), optional] The (T, P, phase) at which the volumetric flow rate is specified to be at, [K] and [Pa]

Notes

> **Warning:** The Stream class is not designed for high-performance or the ability to use different thermodynamic models. It is especially limited in its multiphase support and the ability to solve with specifications other than temperature and pressure. It is impossible to change constant properties such as a compound's critical temperature in this interface.
>
> It is recommended to switch over to the *thermo.flash* and *EquilibriumStream* interfaces which solves those problems and are better positioned to grow. That interface also requires users to be responsible for their chemical constants and pure component correlations; while default values can easily be loaded for most compounds, the user is ultimately responsible for them.

Examples

Creating Stream objects:

A stream of vodka with volume fractions 60% water, 40% ethanol, 1 kg/s:

```
>>> from thermo import Stream
>>> Stream(['water', 'ethanol'], Vfls=[.6, .4], T=300, P=1E5, m=1)
<Stream, components=['water', 'ethanol'], mole fractions=[0.8299, 0.1701], mass␣
↪flow=1.0 kg/s, mole flow=43.883974 mol/s, T=300.00 K, P=100000 Pa>
```

A stream of air at 400 K and 1 bar, flow rate of 1 mol/s:

```
>>> Stream('air', T=400, P=1e5, n=1)
<Stream, components=['nitrogen', 'argon', 'oxygen'], mole fractions=[0.7812, 0.0092,
↪ 0.2096], mass flow=0.028958 kg/s, mole flow=1 mol/s, T=400.00 K, P=100000 Pa>
```

A flow of 1 L/s of 10 wt% phosphoric acid at 320 K:

```
>>> Stream(['water', 'phosphoric acid'], ws=[.9, .1], T=320, P=1E5, Q=0.001)
<Stream, components=['water', 'phosphoric acid'], mole fractions=[0.98, 0.02], mole␣
↪flow=53.2136286991 mol/s, T=320.00 K, P=100000 Pa>
```

Instead of specifying the composition and flow rate separately, they can be specified as a list of flow rates in the appropriate units.

80 kg/s of furfuryl alcohol/water solution:

```
>>> Stream(['furfuryl alcohol', 'water'], ms=[50, 30])
<Stream, components=['furfuryl alcohol', 'water'], mole fractions=[0.2343, 0.7657],␣
↪mole flow=2174.93735951 mol/s, T=298.15 K, P=101325 Pa>
```

A stream of 100 mol/s of 400 K, 1 MPa argon:

```
>>> Stream(['argon'], ns=[100], T=400, P=1E6)
<Stream, components=['argon'], mole fractions=[1.0], mole flow=100 mol/s, T=400.00␣
↪K, P=1000000 Pa>
```

(continues on next page)

(continued from previous page)

A large stream of vinegar, 8 volume %:

```
>>> Stream(['Acetic acid', 'water'], Qls=[1, 1/.088])
<Stream, components=['acetic acid', 'water'], mole fractions=[0.0269, 0.9731], mole␣
↪flow=646268.518749 mol/s, T=298.15 K, P=101325 Pa>
```

A very large stream of 100 m^3/s of steam at 500 K and 2 MPa:

```
>>> Stream(['water'], Qls=[100], T=500, P=2E6)
<Stream, components=['water'], mole fractions=[1.0], mole flow=4617174.33613 mol/s,␣
↪T=500.00 K, P=2000000 Pa>
```

A real example of a stream from a pulp mill:

```
>>> Stream(['Methanol', 'Sulphuric acid', 'sodium chlorate', 'Water', 'Chlorine␣
↪dioxide', 'Sodium chloride', 'Carbon dioxide', 'Formic Acid', 'sodium sulfate',
↪'Chlorine'], T=365.2, P=70900, ns=[0.3361749, 11.5068909, 16.8895876, 7135.
↪9902928, 1.8538332, 0.0480655, 0.0000000, 2.9135162, 205.7106922, 0.0012694])
<Stream, components=['methanol', 'sulfuric acid', 'sodium chlorate', 'water',
↪'chlorine dioxide', 'sodium chloride', 'carbon dioxide', 'formic acid', 'sodium␣
↪sulfate', 'chlorine'], mole fractions=[0.0, 0.0016, 0.0023, 0.9676, 0.0003, 0.0,␣
↪0.0, 0.0004, 0.0279, 0.0], mole flow=7375.2503227 mol/s, T=365.20 K, P=70900 Pa>
```

For streams with large numbers of components, it may be confusing to enter the composition separate from the names of the chemicals. For that case, the syntax using dictionaries as follows is supported with any composition specification:

```
>>> comp = OrderedDict([('methane', 0.96522),
...                     ('nitrogen', 0.00259),
...                     ('carbon dioxide', 0.00596),
...                     ('ethane', 0.01819),
...                     ('propane', 0.0046),
...                     ('isobutane', 0.00098),
...                     ('butane', 0.00101),
...                     ('2-methylbutane', 0.00047),
...                     ('pentane', 0.00032),
...                     ('hexane', 0.00066)])
>>> m = Stream(ws=comp, m=33)
```

Attributes

A Helmholtz energy of the mixture at its current state, in units of [J/kg].

API API gravity of the hypothetical liquid phase of the mixture, [degrees].

Am Helmholtz energy of the mixture at its current state, in units of [J/mol].

Bvirial Second virial coefficient of the gas phase of the mixture at its current temperature, pressure, and composition in units of [mol/m^3].

Cp Mass heat capacity of the mixture at its current phase and temperature, in units of [J/kg/K].

Cpg Gas-phase heat capacity of the mixture at its current temperature , and composition in units of [J/kg/K].

Cpgm Gas-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cpgms Gas-phase ideal gas heat capacity of the chemicals at its current temperature, in units of [J/mol/K].

Cpgs Gas-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Cpl Liquid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/kg/K].

Cplm Liquid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cplms Liquid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Cpls Liquid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Cpm Molar heat capacity of the mixture at its current phase and temperature, in units of [J/mol/K].

Cps Solid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/kg/K].

Cpsm Solid-phase heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cpsms Solid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Cpss Solid-phase pure component heat capacity of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

Cvg Gas-phase ideal-gas contant-volume heat capacity of the mixture at its current temperature, in units of [J/kg/K].

Cvgm Gas-phase ideal-gas contant-volume heat capacity of the mixture at its current temperature and composition, in units of [J/mol/K].

Cvgms Gas-phase pure component ideal-gas contant-volume heat capacities of the chemicals in the mixture at its current temperature, in units of [J/mol/K].

Cvgs Gas-phase pure component ideal-gas contant-volume heat capacities of the chemicals in the mixture at its current temperature, in units of [J/kg/K].

H

Hc Standard higher heat of combustion of the mixture, in units of [J/kg].

Hc_lower Standard lower heat of combustion of the mixture, in units of [J/kg].

Hcm Standard higher molar heat of combustion of the mixture, in units of [J/mol].

Hcm_lower Standard lower molar heat of combustion of the mixture, in units of [J/mol].

Hm

Hvapms Pure component enthalpies of vaporization of the chemicals in the mixture at its current temperature, in units of [J/mol].

Hvaps Enthalpy of vaporization of the chemicals in the mixture at its current temperature, in units of [J/kg].

IUPAC_names IUPAC names for all chemicals in the mixture.

InChI_Keys InChI keys for all chemicals in the mixture.

InChIs InChI strings for all chemicals in the mixture.

JT Joule Thomson coefficient of the mixture at its current phase, temperature, and pressure in units of [K/Pa].

JTg Joule Thomson coefficient of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [K/Pa].

JTgs Pure component Joule Thomson coefficients of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [K/Pa].

JTl Joule Thomson coefficient of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [K/Pa].

JTls Pure component Joule Thomson coefficients of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [K/Pa].

PSRK_groups List of dictionaries of PSRK subgroup: count groups for each chemical in the mixture.

Parachor Parachor of the mixture at its current temperature and pressure, in units of [N^0.25*m^2.75/mol].

Parachors Pure component Parachor parameters of the chemicals in the mixture at its current temperature and pressure, in units of [N^0.25*m^2.75/mol].

Pbubble Bubble point pressure of the mixture at its current temperature and composition, in units of [Pa].

Pdew Dew point pressure of the mixture at its current temperature and composition, in units of [Pa].

Pr Prandtl number of the mixture at its current temperature, pressure, and phase; [dimensionless].

Prg Prandtl number of the gas phase of the mixture if one exists at its current temperature and pressure, [dimensionless].

Prgs Pure component Prandtl numbers of the gas phase of the chemicals in the mixture at its current temperature and pressure, [dimensionless].

Prl Prandtl number of the liquid phase of the mixture if one exists at its current temperature and pressure, [dimensionless].

Prls Pure component Prandtl numbers of the liquid phase of the chemicals in the mixture at its current temperature and pressure, [dimensionless].

Psats Pure component vapor pressures of the chemicals in the mixture at its current temperature, in units of [Pa].

PubChems PubChem Component ID numbers for all chemicals in the mixture.

R_specific Specific gas constant of the mixture, in units of [J/kg/K].

SG Specific gravity of the mixture, [dimensionless].

SGg Specific gravity of a hypothetical gas phase of the mixture, .

SGl Specific gravity of a hypothetical liquid phase of the mixture at the specified temperature and pressure, [dimensionless].

SGs Specific gravity of a hypothetical solid phase of the mixture at the specified temperature and pressure, [dimensionless].

`Tbubble` Bubble point temperature of the mixture at its current pressure and composition, in units of [K].

`Tdew` Dew point temperature of the mixture at its current pressure and composition, in units of [K].

`U` Internal energy of the mixture at its current state, in units of [J/kg].

`UNIFAC_Dortmund_groups` List of dictionaries of Dortmund UNIFAC subgroup: count groups for each chemcial in the mixture.

`UNIFAC_Qs` UNIFAC *Q* (normalized Van der Waals area) values, dimensionless.

`UNIFAC_Rs` UNIFAC *R* (normalized Van der Waals volume) values, dimensionless.

`UNIFAC_groups` List of dictionaries of UNIFAC subgroup: count groups for each chemical in the mixture.

`Um` Internal energy of the mixture at its current state, in units of [J/mol].

V_over_F

`Van_der_Waals_areas` List of unnormalized Van der Waals areas of all the chemicals in the mixture, in units of [m^2/mol].

`Van_der_Waals_volumes` List of unnormalized Van der Waals volumes of all the chemicals in the mixture, in units of [m^3/mol].

`Vm` Molar volume of the mixture at its current phase and temperature and pressure, in units of [m^3/mol].

`Vmg` Gas-phase molar volume of the mixture at its current temperature, pressure, and composition in units of [m^3/mol].

`Vmg_STP` Gas-phase molar volume of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [m^3/mol].

`Vmgs` Pure component gas-phase molar volumes of the chemicals in the mixture at its current temperature and pressure, in units of [m^3/mol].

`Vml` Liquid-phase molar volume of the mixture at its current temperature, pressure, and composition in units of [m^3/mol].

`Vml_STP` Liquid-phase molar volume of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [m^3/mol].

`Vmls` Pure component liquid-phase molar volumes of the chemicals in the mixture at its current temperature and pressure, in units of [m^3/mol].

Vms

`Vmss` Pure component solid-phase molar volumes of the chemicals in the mixture at its current temperature, in units of [m^3/mol].

`Z` Compressibility factor of the mixture at its current phase and temperature and pressure, [dimensionless].

`Zg` Compressibility factor of the mixture in the gas phase at the current temperature, pressure, and composition, [dimensionless].

`Zg_STP` Gas-phase compressibility factor of the mixture at 298.15 K and 101.325 kPa, and the current composition, [dimensionless].

`Zgs` Pure component compressibility factors of the chemicals in the mixture in the gas phase at the current temperature and pressure, [dimensionless].

Zl Compressibility factor of the mixture in the liquid phase at the current temperature, pressure, and composition, [dimensionless].

Zl_STP Liquid-phase compressibility factor of the mixture at 298.15 K and 101.325 kPa, and the current composition, [dimensionless].

Zls Pure component compressibility factors of the chemicals in the liquid phase at the current temperature and pressure, [dimensionless].

Zss Pure component compressibility factors of the chemicals in the mixture in the solid phase at the current temperature and pressure, [dimensionless].

alpha Thermal diffusivity of the mixture at its current temperature, pressure, and phase in units of [m^2/s].

alphag Thermal diffusivity of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

alphags Pure component thermal diffusivities of the chemicals in the mixture in the gas phase at the current temperature and pressure, in units of [m^2/s].

alphal Thermal diffusivity of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

alphals Pure component thermal diffusivities of the chemicals in the mixture in the liquid phase at the current temperature and pressure, in units of [m^2/s].

atom_fractions Dictionary of atomic fractions for each atom in the mixture.

atom_fractionss List of dictionaries of atomic fractions for all chemicals in the mixture.

atoms Mole-averaged dictionary of atom counts for all atoms of the chemicals in the mixture.

atomss List of dictionaries of atom counts for all chemicals in the mixture.

charge_balance Charge imbalance of the mixture, in units of [faraday].

charges Charges for all chemicals in the mixture, [faraday].

composition_specified Always needs a composition

conductivity

constants Returns a :obj:`thermo.chemical_package.ChemicalConstantsPackage instance with constants from the mixture, [-].

economic_statuses List of dictionaries of the economic status for all chemicals in the mixture.

eos Equation of state object held by the mixture.

flow_specified Always needs a flow specified

formulas Chemical formulas for all chemicals in the mixture.

isentropic_exponent Gas-phase ideal-gas isentropic exponent of the mixture at its current temperature, [dimensionless].

isentropic_exponents Gas-phase pure component ideal-gas isentropic exponent of the chemicals in the mixture at its current temperature, [dimensionless].

isobaric_expansion Isobaric (constant-pressure) expansion of the mixture at its current phase, temperature, and pressure in units of [1/K].

isobaric_expansion_g Isobaric (constant-pressure) expansion of the gas phase of the mixture at its current temperature and pressure, in units of [1/K].

isobaric_expansion_gs Pure component isobaric (constant-pressure) expansions of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [1/K].

isobaric_expansion_l Isobaric (constant-pressure) expansion of the liquid phase of the mixture at its current temperature and pressure, in units of [1/K].

isobaric_expansion_ls Pure component isobaric (constant-pressure) expansions of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [1/K].

k Thermal conductivity of the mixture at its current phase, temperature, and pressure in units of [W/m/K].

kg Thermal conductivity of the mixture in the gas phase at its current temperature, pressure, and composition in units of [Pa*s].

kgs Pure component thermal conductivies of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [W/m/K].

kl Thermal conductivity of the mixture in the liquid phase at its current temperature, pressure, and composition in units of [Pa*s].

kls Pure component thermal conductivities of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [W/m/K].

ks

legal_statuses List of dictionaries of the legal status for all chemicals in the mixture.

mass_fractions Dictionary of mass fractions for each atom in the mixture.

mass_fractionss List of dictionaries of mass fractions for all chemicals in the mixture.

mu Viscosity of the mixture at its current phase, temperature, and pressure in units of [Pa*s].

mug Viscosity of the mixture in the gas phase at its current temperature, pressure, and composition in units of [Pa*s].

mugs Pure component viscosities of the chemicals in the mixture in the gas phase at its current temperature and pressure, in units of [Pa*s].

mul Viscosity of the mixture in the liquid phase at its current temperature, pressure, and composition in units of [Pa*s].

muls Pure component viscosities of the chemicals in the mixture in the liquid phase at its current temperature and pressure, in units of [Pa*s].

non_pressure_spec_specified Cannot have a stream without an energy-type spec.

nu Kinematic viscosity of the the mixture at its current temperature, pressure, and phase in units of [m^2/s].

nug Kinematic viscosity of the gas phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

nugs Pure component kinematic viscosities of the gas phase of the chemicals in the mixture at its current temperature and pressure, in units of [m^2/s].

nul Kinematic viscosity of the liquid phase of the mixture if one exists at its current temperature and pressure, in units of [m^2/s].

nuls Pure component kinematic viscosities of the liquid phase of the chemicals in the mixture at its current temperature and pressure, in units of [m^2/s].

permittivites Pure component relative permittivities of the chemicals in the mixture at its current temperature, [dimensionless].

phase

property_package_constants

rho Mass density of the mixture at its current phase and temperature and pressure, in units of [kg/m^3].

rhog Gas-phase mass density of the mixture at its current temperature, pressure, and composition in units of [kg/m^3].

rhog_STP Gas-phase mass density of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [kg/m^3].

rhogm Molar density of the mixture in the gas phase at the current temperature, pressure, and composition in units of [mol/m^3].

rhogm_STP Molar density of the mixture in the gas phase at 298.15 K and 101.325 kPa, and the current composition, in units of [mol/m^3].

rhogms Pure component molar densities of the chemicals in the gas phase at the current temperature and pressure, in units of [mol/m^3].

rhogs Pure-component gas-phase mass densities of the chemicals in the mixture at its current temperature and pressure, in units of [kg/m^3].

rhol Liquid-phase mass density of the mixture at its current temperature, pressure, and composition in units of [kg/m^3].

rhol_STP Liquid-phase mass density of the mixture at 298.15 K and 101.325 kPa, and the current composition in units of [kg/m^3].

rholm Molar density of the mixture in the liquid phase at the current temperature, pressure, and composition in units of [mol/m^3].

rholm_STP Molar density of the mixture in the liquid phase at 298.15 K and 101.325 kPa, and the current composition, in units of [mol/m^3].

rholms Pure component molar densities of the chemicals in the mixture in the liquid phase at the current temperature and pressure, in units of [mol/m^3].

rhols Pure-component liquid-phase mass density of the chemicals in the mixture at its current temperature and pressure, in units of [kg/m^3].

rhom Molar density of the mixture at its current phase and temperature and pressure, in units of [mol/m^3].

rhos

rhosms Pure component molar densities of the chemicals in the solid phase at the current temperature and pressure, in units of [mol/m^3].

rhoss Pure component solid-phase mass density of the chemicals in the mixture at its current temperature, in units of [kg/m^3].

ringss List of ring counts for all chemicals in the mixture.

sigma Surface tension of the mixture at its current temperature and composition, in units of [N/m].

sigmas Pure component surface tensions of the chemicals in the mixture at its current temperature, in units of [N/m].

similarity_variables Similarity variables for all chemicals in the mixture, see

smiless SMILES strings for all chemicals in the mixture.

solubility_parameters Pure component solubility parameters of the chemicals in the mixture at its current temperature and pressure, in units of [Pa^0.5].

specified_composition_vars number of composition variables

specified_flow_vars Always needs only one flow specified

specified_state_vars Always needs two states

speed_of_sound Bulk speed of sound of the mixture at its current temperature, [m/s].

speed_of_sound_g Gas-phase speed of sound of the mixture at its current temperature, [m/s].

speed_of_sound_l Liquid-phase speed of sound of the mixture at its current temperature, [m/s].

state_specified Always needs a state

state_specs Returns a list of tuples of (state_variable, state_value) representing the thermodynamic state of the system.

synonymss Lists of synonyms for all chemicals in the mixture.

xs

ys

Methods

Hc_volumetric_g([T, P])	Standard higher molar heat of combustion of the mixture, in units of [J/m^3] at the specified T and P in the gas phase.
Hc_volumetric_g_lower([T, P])	Standard lower molar heat of combustion of the mixture, in units of [J/m^3] at the specified T and P in the gas phase.
StreamArgs()	Goal to create a StreamArgs instance, with the user specified variables always being here.
Vfgs([T, P])	Volume fractions of all species in a hypothetical pure-gas phase at the current or specified temperature and pressure.
Vfls([T, P])	Volume fractions of all species in a hypothetical pure-liquid phase at the current or specified temperature and pressure.
draw_2d([Hs])	Interface for drawing a 2D image of all the molecules in the mixture.
set_chemical_TP([T, P])	Basic method to change all chemical instances to be at the T and P specified.
set_chemical_constants()	Basic method which retrieves and sets constants of chemicals to be accessible as lists from a Mixture object.

Bond	
Capillary	
Grashof	
Jakob	
Peclet_heat	
Reynolds	
Weber	
calculate	
compound_index	
eos_pures	
flash	
flash_caloric	
properties	
set_Chemical_property_objects	
set_TP_sources	
set_constant_sources	
set_constants	
set_eos	
set_extensive_flow	
set_extensive_properties	
set_property_package	

`StreamArgs()`

Goal to create a StreamArgs instance, with the user specified variables always being here.

The state variables are currently correctly tracked. The flow rate and composition variable needs to be tracked as a function of what was specified as the input variables.

The flow rate needs to be changed wen the stream flow rate is changed. Note this stores unnormalized specs, but that this is OK.

calculate(*T=None*, *P=None*)

property composition_specified

Always needs a composition

flash(*T=None*, *P=None*, *VF=None*, *H=None*, *Hm=None*, *S=None*, *Sm=None*, *energy=None*)

flashed = True

property flow_specified

Always needs a flow specified

property non_pressure_spec_specified

Cannot have a stream without an energy-type spec.

set_extensive_flow(*n=None*)

set_extensive_properties()

property specified_composition_vars

number of composition variables

property specified_flow_vars
Always needs only one flow specified

property specified_state_vars
Always needs two states

property state_specified
Always needs a state

property state_specs
Returns a list of tuples of (state_variable, state_value) representing the thermodynamic state of the system.

class thermo.stream.StreamArgs(*IDs=None*, *zs=None*, *ws=None*, *Vfls=None*, *Vfgs=None*, *T=None*, *P=None*, *VF=None*, *H=None*, *Hm=None*, *S=None*, *Sm=None*, *ns=None*, *ms=None*, *Qls=None*, *Qgs=None*, *m=None*, *n=None*, *Q=None*, *energy=None*, *Vf_TP=(None, None)*, *Q_TP=(None, None, '')*, *pkg=None*, *single_composition_basis=True*)

Bases: object

Attributes

H

Hm

Hm_calc

IDs

MW

P

P_calc

Q

Qgs

Qls

S

Sm

T

T_calc

VF

VF_calc

Vfgs

Vfls

clean If no variables (other than IDs) have been specified, return True, otherwise return False.

composition_spec

composition_specified

energy

energy_calc

flow_spec

flow_specified

m

m_calc

mixture

ms

n

n_calc

non_pressure_spec_specified

ns

ns_calc

specified_composition_vars

specified_flow_vars

specified_state_vars

state_specified

state_specs

stream

ws

zs

zs_calc

Methods

copy	
flash	
flash_state	
reconcile_flows	
update	

`property H`

`property Hm`

`property Hm_calc`

`property IDs`

`property MW`

`property P`

`property P_calc`

`property Q`

`property Qgs`

`property Qls`

property S

property Sm

property T

property T_calc

property VF

property VF_calc

property Vfgs

property Vfls

property clean

If no variables (other than IDs) have been specified, return True, otherwise return False.

property composition_spec

property composition_specified

copy()

property energy

property energy_calc

flash(*hot_start=None*, *existing_flash=None*)

flash_state(*hot_start=None*)

flashed = False

property flow_spec

property flow_specified

property m

property m_calc

property mixture

property ms

property n

property n_calc

property non_pressure_spec_specified

property ns

property ns_calc

reconcile_flows(*n_tol=2e-15*, *m_tol=2e-15*)

property specified_composition_vars

property specified_flow_vars

property specified_state_vars

property state_specified

property state_specs

property stream

update(***kwargs*)

property ws

property zs

property zs_calc

thermo.stream.**energy_balance**(*inlets*, *outlets*)

thermo.stream.**mole_balance**(*inlets*, *outlets*, *compounds*)

7.28 Thermal Conductivity (thermo.thermal_conductivity)

This module contains implementations of *TPDependentProperty* representing liquid and vapor thermal conductivity. A variety of estimation and data methods are available as included in the *chemicals* library. Additionally liquid and vapor mixture thermal conductivity predictor objects are implemented subclassing *MixtureProperty*.

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

7.28.1 Pure Liquid Thermal Conductivity

class thermo.thermal_conductivity.**ThermalConductivityLiquid**(*CASRN=''*, *MW=None*, *Tm=None*, *Tb=None*, *Tc=None*, *Pc=None*, *omega=None*, *Hfus=None*, *extrapolation='linear'*, *extrapolation_min=0.0001*, ***kwargs*)

Bases: *thermo.utils.tp_dependent_property.TPDependentProperty*

Class for dealing with liquid thermal conductivity as a function of temperature and pressure.

For low-pressure (at 1 atm while under the vapor pressure; along the saturation line otherwise) liquids, there is one source of tabular information, one polynomial-based method, 7 corresponding-states estimators, and the external library CoolProp.

For high-pressure liquids (also, <1 atm liquids), there are two corresponding-states estimator, and the external library CoolProp.

Parameters

CAS [str, optional] The CAS number of the compound, [-]

MW [float, optional] Molecular weight, [g/mol]

Tm [float, optional] Melting point, [K]

Tb [float, optional] Boiling point, [K]

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

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

omega [float, optional] Acentric factor, [-]

Hfus [float, optional] Heat of fusion, [J/mol]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.thermal_conductivity.Sheffy_Johnson`

`chemicals.thermal_conductivity.Sato_Riedel`

`chemicals.thermal_conductivity.Lakshmi_Prasad`

`chemicals.thermal_conductivity.Gharagheizi_liquid`

`chemicals.thermal_conductivity.Nicola_original`

`chemicals.thermal_conductivity.Nicola`

`chemicals.thermal_conductivity.Bahadori_liquid`

`chemicals.thermal_conductivity.DIPPR9G`

`chemicals.thermal_conductivity.Missenard`

Notes

To iterate over all methods, use the lists stored in *thermal_conductivity_liquid_methods* and *thermal_conductivity_liquid_methods_P* for low and high pressure methods respectively.

Low pressure methods:

GHARAGHEIZI_L: CSP method, described in `Gharagheizi_liquid`.

SATO_RIEDEL: CSP method, described in `Sato_Riedel`.

NICOLA: CSP method, described in `Nicola`.

NICOLA_ORIGINAL: CSP method, described in `Nicola_original`.

SHEFFY_JOHNSON: CSP method, described in `Sheffy_Johnson`.

BAHADORI_L: CSP method, described in `Bahadori_liquid`.

LAKSHMI_PRASAD: CSP method, described in `Lakshmi_Prasad`.

DIPPR_PERRY_8E: A collection of 340 coefficient sets from the DIPPR database published openly in [3]. Provides temperature limits for all its fluids. `EQ100` is used for its fluids.

VDI_PPDS: Coefficients for a equation form developed by the PPDS, published openly in [2]. Covers a large temperature range, but does not extrapolate well at very high or very low temperatures. 271 compounds.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [1]. Very slow.

VDI_TABULAR: Tabular data in [2] along the saturation curve; interpolation is as set by the user or the default.

High pressure methods:

DIPPR_9G: CSP method, described in `DIPPR9G`. Calculates a low-pressure thermal conductivity first from the low-pressure method.

MISSENARD: CSP method, described in `Missenard`. Calculates a low-pressure thermal conductivity first from the low-pressure method.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [1]. Very slow, but unparalled in accuracy for pressure dependence.

References

[1], [2], [3]

Attributes

Tmax Maximum temperature (K) at which the current method can calculate the property.

Tmin Minimum temperature (K) at which the current method can calculate the property.

Methods

calculate(T, method)	Method to calculate low-pressure liquid thermal conductivity at tempearture T with a given method.
calculate_P(T, P, method)	Method to calculate pressure-dependent liquid thermal conductivity at temperature T and pressure P with a given method.
test_method_validity(T, method)	Method to check the validity of a temperature-dependent low-pressure method.
test_method_validity_P(T, P, method)	Method to check the validity of a high-pressure method.

property Tmax
Maximum temperature (K) at which the current method can calculate the property.

property Tmin
Minimum temperature (K) at which the current method can calculate the property.

calculate(*T*, *method*)
Method to calculate low-pressure liquid thermal conductivity at tempearture T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature of the liquid, [K]

method [str] Name of the method to use

Returns

kl [float] Thermal conductivity of the liquid at T and a low pressure, [W/m/K]

`calculate_P`(*T*, *P*, *method*)

Method to calculate pressure-dependent liquid thermal conductivity at temperature *T* and pressure *P* with a given method.

This method has no exception handling; see `TP_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate liquid thermal conductivity, [K]

P [float] Pressure at which to calculate liquid thermal conductivity, [K]

method [str] Name of the method to use

Returns

kl [float] Thermal conductivity of the liquid at T and P, [W/m/K]

name = 'liquid thermal conductivity'

property_max = 10.0

Maximum valid value of liquid thermal conductivity. Generous limit.

property_min = 0.0

Mimimum valid value of liquid thermal conductivity.

ranked_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'VDI_TABULAR', 'GHARAGHEIZI_L', 'SHEFFY_JOHNSON', 'SATO_RIEDEL', 'LAKSHMI_PRASAD', 'BAHADORI_L', 'NICOLA', 'NICOLA_ORIGINAL']

Default rankings of the low-pressure methods.

ranked_methods_P = ['COOLPROP', 'DIPPR_9G', 'MISSENARD']

Default rankings of the high-pressure methods.

`test_method_validity`(*T*, *method*)

Method to check the validity of a temperature-dependent low-pressure method. For CSP methods, the models **BAHADORI_L**, **LAKSHMI_PRASAD**, and **SHEFFY_JOHNSON** are considered valid for all temperatures. For methods **GHARAGHEIZI_L**, **NICOLA**, and **NICOLA_ORIGINAL**, the methods are considered valid up to 1.5Tc and down to 0 K. Method **SATO_RIEDEL** does not work above the critical point, so it is valid from 0 K to the critical point.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`test_method_validity_P`(*T*, *P*, *method*)

Method to check the validity of a high-pressure method. For **COOLPROP**, the fluid must be both a liquid and under the maximum pressure of the fluid's EOS. **MISSENARD** has defined limits; between 0.5Tc and 0.8Tc, and below 200Pc. The CSP method **DIPPR_9G** is considered valid for all temperatures and pressures.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures and pressures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

P [float] Pressure at which to test the method, [Pa]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`units = 'W/m/K'`

The following variables are available to specify which method to use.

`thermo.thermal_conductivity.COOLPROP`

`thermo.thermal_conductivity.DIPPR_PERRY_8E`

`thermo.thermal_conductivity.VDI_PPDS`

`thermo.thermal_conductivity.VDI_TABULAR`

`thermo.thermal_conductivity.GHARAGHEIZI_L`

`thermo.thermal_conductivity.SHEFFY_JOHNSON`

`thermo.thermal_conductivity.SATO_RIEDEL`

`thermo.thermal_conductivity.LAKSHMI_PRASAD`

`thermo.thermal_conductivity.BAHADORI_L`

`thermo.thermal_conductivity.NICOLA`

`thermo.thermal_conductivity.NICOLA_ORIGINAL`

The following variables contain lists of available methods.

`thermo.thermal_conductivity.thermal_conductivity_liquid_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'VDI_TABULAR', 'GHARAGHEIZI_L', 'SHEFFY_JOHNSON', 'SATO_RIEDEL', 'LAKSHMI_PRASAD', 'BAHADORI_L', 'NICOLA', 'NICOLA_ORIGINAL']`

Holds all low-pressure methods available for the *ThermalConductivityLiquid* class, for use in iterating over them.

`thermo.thermal_conductivity.thermal_conductivity_liquid_methods_P = ['COOLPROP', 'DIPPR_9G', 'MISSENARD']`

Holds all high-pressure methods available for the *ThermalConductivityLiquid* class, for use in iterating over them.

7.28.2 Pure Gas Thermal Conductivity

class thermo.thermal_conductivity.**ThermalConductivityGas**(*CASRN=''*, *MW=None*, *Tb=None*, *Tc=None*, *Pc=None*, *Vc=None*, *Zc=None*, *omega=None*, *dipole=None*, *Vmg=None*, *Cpgm=None*, *mug=None*, *extrapolation='linear'*, *extrapolation_min=0.0001*, ***kwargs*)

Bases: `thermo.utils.tp_dependent_property.TPDependentProperty`

Class for dealing with gas thermal conductivity as a function of temperature and pressure.

For gases at atmospheric pressure, there are 7 corresponding-states estimators, one source of tabular information, and the external library CoolProp.

For gases under the fluid's boiling point (at sub-atmospheric pressures), and high-pressure gases above the boiling point, there are three corresponding-states estimators, and the external library CoolProp.

Parameters

CAS [str, optional] The CAS number of the compound, [-]

MW [float, optional] Molecular weight, [g/mol]

Tb [float, optional] Boiling point, [K]

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, [-]

dipole [float, optional] Dipole moment of the fluid, [debye]

Vmg [float or callable, optional] Molar volume of the fluid at a pressure and temperature or callable for the same, [m^3/mol]

Cpgm [float or callable, optional] Molar constant-pressure heat capacity of the fluid at a pressure and temperature or callable for the same, [J/mol/K]

mug [float or callable, optional] Gas viscosity of the fluid at a pressure and temperature or callable for the same, [Pa*s]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see `TDependentProperty` for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.thermal_conductivity.Bahadori_gas`

`chemicals.thermal_conductivity.Gharagheizi_gas`

`chemicals.thermal_conductivity.Eli_Hanley`

`chemicals.thermal_conductivity.Chung`

chemicals.thermal_conductivity.DIPPR9B

chemicals.thermal_conductivity.Eucken_modified

chemicals.thermal_conductivity.Eucken

chemicals.thermal_conductivity.Stiel_Thodos_dense

chemicals.thermal_conductivity.Eli_Hanley_dense

chemicals.thermal_conductivity.Chung_dense

Notes

To iterate over all methods, use the lists stored in *thermal_conductivity_gas_methods* and *thermal_conductivity_gas_methods_P* for low and high pressure methods respectively.

Low pressure methods:

GHARAGHEIZI_G: CSP method, described in Gharagheizi_gas.

DIPPR_9B: CSP method, described in DIPPR9B.

CHUNG: CSP method, described in Chung.

ELI_HANLEY: CSP method, described in Eli_Hanley.

EUCKEN_MOD: CSP method, described in Eucken_modified.

EUCKEN: CSP method, described in Eucken.

BAHADORI_G: CSP method, described in Bahadori_gas.

DIPPR_PERRY_8E: A collection of 345 coefficient sets from the DIPPR database published openly in [3]. Provides temperature limits for all its fluids. chemicals.dippr.EQ102 is used for its fluids.

VDI_PPDS: Coefficients for a equation form developed by the PPDS, published openly in [2]. Covers a large temperature range, but does not extrapolate well at very high or very low temperatures. 275 compounds.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [1]. Very slow.

VDI_TABULAR: Tabular data in [2] along the saturation curve; interpolation is as set by the user or the default.

High pressure methods:

STIEL_THODOS_DENSE: CSP method, described in Stiel_Thodos_dense. Calculates a low-pressure thermal conductivity first.

ELI_HANLEY_DENSE: CSP method, described in Eli_Hanley_dense. Calculates a low-pressure thermal conductivity first.

CHUNG_DENSE: CSP method, described in Chung_dense. Calculates a low-pressure thermal conductivity first.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [1]. Very slow, but unparalled in accuracy for pressure dependence.

References

[1], [2], [3]

Methods

`calculate`(T, method)	Method to calculate low-pressure gas thermal conductivity at tempearture T with a given method.
`calculate_P`(T, P, method)	Method to calculate pressure-dependent gas thermal conductivity at temperature T and pressure P with a given method.
`test_method_validity`(T, method)	Method to check the validity of a temperature-dependent low-pressure method.
`test_method_validity_P`(T, P, method)	Method to check the validity of a high-pressure method.

calculate(*T*, *method*)

Method to calculate low-pressure gas thermal conductivity at tempearture T with a given method.

This method has no exception handling; see `T_dependent_property` for that.

Parameters

T [float] Temperature of the gas, [K]

method [str] Name of the method to use

Returns

kg [float] Thermal conductivity of the gas at T and a low pressure, [W/m/K]

calculate_P(*T*, *P*, *method*)

Method to calculate pressure-dependent gas thermal conductivity at temperature T and pressure P with a given method.

This method has no exception handling; see `TP_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate gas thermal conductivity, [K]

P [float] Pressure at which to calculate gas thermal conductivity, [K]

method [str] Name of the method to use

Returns

kg [float] Thermal conductivity of the gas at T and P, [W/m/K]

name = 'gas thermal conductivity'

property_max = 10

Maximum valid value of gas thermal conductivity. Generous limit.

property_min = 0

Mimimum valid value of gas thermal conductivity.

ranked_methods = ['COOLPROP', 'VDI_PPDS', 'DIPPR_PERRY_8E', 'VDI_TABULAR', 'GHARAGHEIZI_G', 'DIPPR_9B', 'CHUNG', 'ELI_HANLEY', 'EUCKEN_MOD', 'EUCKEN', 'BAHADORI_G']

Default rankings of the low-pressure methods.

ranked_methods_P = ['COOLPROP', 'ELI_HANLEY_DENSE', 'CHUNG_DENSE', 'STIEL_THODOS_DENSE']
Default rankings of the high-pressure methods.

test_method_validity(*T*, *method*)
Method to check the validity of a temperature-dependent low-pressure method. For CSP methods, the all methods are considered valid from 0 K and up.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid. **GHARAGHEIZI_G** and **BAHADORI_G** are known to sometimes produce negative results.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

test_method_validity_P(*T*, *P*, *method*)
Method to check the validity of a high-pressure method. For **COOLPROP**, the fluid must be both a gas and under the maximum pressure of the fluid's EOS. The CSP method **ELI_HANLEY_DENSE**, **CHUNG_DENSE**, and **STIEL_THODOS_DENSE** are considered valid for all temperatures and pressures.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures and pressures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

P [float] Pressure at which to test the method, [Pa]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'W/m/K'

thermo.thermal_conductivity.**thermal_conductivity_gas_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'VDI_TABULAR', 'GHARAGHEIZI_G', 'DIPPR_9B', 'CHUNG', 'ELI_HANLEY', 'EUCKEN_MOD', 'EUCKEN', 'BAHADORI_G']**
Holds all low-pressure methods available for the *ThermalConductivityGas* class, for use in iterating over them.

thermo.thermal_conductivity.**thermal_conductivity_gas_methods_P = ['COOLPROP', 'ELI_HANLEY_DENSE', 'CHUNG_DENSE', 'STIEL_THODOS_DENSE']**
Holds all high-pressure methods available for the *ThermalConductivityGas* class, for use in iterating over them.

7.28.3 Mixture Liquid Thermal Conductivity

class thermo.thermal_conductivity.**ThermalConductivityLiquidMixture**(*CASs=[], ThermalConductivityLiquids=[], MWs=[], **kwargs*)

Bases: *thermo.utils.mixture_property.MixtureProperty*

Class for dealing with thermal conductivity of a liquid mixture as a function of temperature, pressure, and composition. Consists of two mixing rule specific to liquid thremal conductivity, one coefficient-based method for aqueous electrolytes, and mole weighted averaging. Most but not all methods are shown in [1].

Prefered method is DIPPR_9H which requires mass fractions, and pure component liquid thermal conductivities. This is substantially better than the ideal mixing rule based on mole fractions, **LINEAR**. Filippov is of similar accuracy but applicable to binary systems only.

Parameters

CASs [str, optional] The CAS numbers of all species in the mixture, [-]

ThermalConductivityLiquids [list[ThermalConductivityLiquid], optional] ThermalConductivityLiquid objects created for all species in the mixture, [-]

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

correct_pressure_pure [bool, optional] Whether to try to use the better pressure-corrected pure component models or to use only the T-only dependent pure species models, [-]

See also:

chemicals.thermal_conductivity.DIPPR9H

chemicals.thermal_conductivity.Filippov

chemicals.thermal_conductivity.thermal_conductivity_Magomedov

Notes

To iterate over all methods, use the list stored in *thermal_conductivity_liquid_mixture_methods*.

DIPPR_9H: Mixing rule described in DIPPR9H.

FILIPPOV: Mixing rule described in Filippov; for two binary systems only.

MAGOMEDOV: Coefficient-based method for aqueous electrolytes only, described in *thermo.electrochem.thermal_conductivity_Magomedov*.

LINEAR: Mixing rule described in mixing_simple.

References

[1]

Methods

`calculate`(T, P, zs, ws, method)	Method to calculate thermal conductivity of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
`test_method_validity`(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

calculate(*T*, *P*, *zs*, *ws*, *method*)

Method to calculate thermal conductivity of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see `mixture_property` for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

k [float] Thermal conductivity of the liquid mixture, [W/m/K]

name = 'liquid thermal conductivity'

property_max = 10

Maximum valid value of liquid thermal conductivity. Generous limit.

property_min = 0

Mimimum valid value of liquid thermal conductivity.

ranked_methods = ['MAGOMEDOV', 'DIPPR_9H', 'LINEAR', 'FILIPPOV']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)

Method to test the validity of a specified method for the given conditions. If **MAGOMEDOV** is applicable (electrolyte system), no other methods are considered viable. Otherwise, there are no easy checks that can be performed here.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'W/m/K'

thermo.thermal_conductivity.**thermal_conductivity_liquid_mixture_methods = ['MAGOMEDOV', 'DIPPR_9H', 'FILIPPOV', 'LINEAR']**

Holds all mixing rules available for the *ThermalConductivityLiquidMixture* class, for use in iterating over them.

7.28.4 Mixture Gas Thermal Conductivity

class thermo.thermal_conductivity.**ThermalConductivityGasMixture**(*MWs=[]*, *Tbs=[]*, *CASs=[]*, *ThermalConductivityGases=[]*, *ViscosityGases=[]*, ***kwargs*)

Bases: *thermo.utils.mixture_property.MixtureProperty*

Class for dealing with thermal conductivity of a gas mixture as a function of temperature, pressure, and composition. Consists of one mixing rule specific to gas thremal conductivity, and mole weighted averaging.

Prefered method is Lindsay_Bromley which requires mole fractions, pure component viscosities and thermal conductivities, and the boiling point and molecular weight of each pure component. This is substantially better than the ideal mixing rule based on mole fractions, **LINEAR** which is also available. More information on this topic can be found in [1].

Parameters

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

Tbs [list[float], optional] Boiling points of all species in the mixture, [K]

CASs [str, optional] The CAS numbers of all species in the mixture

ThermalConductivityGases [list[ThermalConductivityGas], optional] ThermalConductivityGas objects created for all species in the mixture, [-]

ViscosityGases [list[ViscosityGas], optional] ViscosityGas objects created for all species in the mixture, [-]

correct_pressure_pure [bool, optional] Whether to try to use the better pressure-corrected pure component models or to use only the T-only dependent pure species models, [-]

See also:

chemicals.thermal_conductivity.Lindsay_Bromley

Notes

To iterate over all methods, use the list stored in *thermal_conductivity_gas_methods*.

LINDSAY_BROMLEY: Mixing rule described in Lindsay_Bromley.

LINEAR: Mixing rule described in mixing_simple.

References

[1]

Methods

`calculate`(T, P, zs, ws, method)	Method to calculate thermal conductivity of a gas mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
`test_method_validity`(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate thermal conductivity of a gas mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see `mixture_property` for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

kg [float] Thermal conductivity of gas mixture, [W/m/K]

name = 'gas thermal conductivity'

property_max = 10.0
Maximum valid value of gas thermal conductivity. Generous limit.

property_min = 0.0
Mimimum valid value of gas thermal conductivity.

ranked_methods = ['LINDSAY_BROMLEY', 'LINEAR']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

`units = 'W/m/K'`

`thermo.thermal_conductivity.thermal_conductivity_gas_mixture_methods = ['LINDSAY_BROMLEY', 'LINEAR']`

Holds all mixing rules available for the *ThermalConductivityGasMixture* class, for use in iterating over them.

7.29 UNIFAC Gibbs Excess Model (thermo.unifac)

This module contains functions and classes related to the UNIFAC and its many variants. The bulk of the code relates to calculating derivativies, or is tables of data.

For reporting bugs, adding feature requests, or submitting pull requests, please use the GitHub issue tracker or contact the author at Caleb.Andrew.Bell@gmail.com.

7.29.1 Main Model (Object-Oriented)

class `thermo.unifac.UNIFAC`(*T*, *xs*, *rs*, *qs*, *Qs*, *vs*, *psi_coeffs=None*, *psi_abc=None*, *version=0*)

Class for representing an a liquid with excess gibbs energy represented by the UNIFAC equation. This model is capable of representing VL and LL behavior, provided the correct interaction parameters are used. [1] and [2] are good references on this model.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

rs [list[float]] *r* parameters $r_i = \sum_{k=1}^{n} \nu_k R_k$, [-]

qs [list[float]] q parameters $q_i = \sum_{k=1}^{n} \nu_k Q_k$, [-]

Qs [list[float]] Q parameter for each subgroup; subgroups are not required to but are suggested to be sorted from lowest number to highest number, [-]

vs [list[list[float]]] Indexed by [subgroup][count], this variable is the count of each subgroups in each compound, [-]

psi_abc [tuple(list[list[float]], 3), optional] *psi* interaction parameters between each subgroup; indexed [subgroup][subgroup], not symmetrical; first arg is the matrix for a, then b, and then c. Only one of *psi_abc* or *psi_coeffs* is required, [-]

psi_coeffs [list[list[tuple(float, 3)]], optional] *psi* interaction parameters between each subgroup; indexed [subgroup][subgroup][letter], not symmetrical. Only one of *psi_abc* or *psi_coeffs* is required, [-]

version [int, optional] Which version of the model to use [-]

- 0 - original UNIFAC, OR UNIFAC LLE
- 1 - Dortmund UNIFAC (adds T dept, 3/4 power)
- 2 - PSRK (original with T dept function)
- 3 - VTPR (drops combinatorial term, Dortmund UNIFAC otherwise)
- 4 - Lyngby/Larsen has different combinatorial, 2/3 power
- 5 - UNIFAC KT (2 params for psi, Lyngby/Larsen formulation; otherwise same as original)

Notes

In addition to the methods presented here, the methods of its base class `thermo.activity.GibbsExcess` are available as well.

References

[1], [2]

Examples

The DDBST has published numerous sample problems using UNIFAC; a simple binary system from example P05.22a in [2] with n-hexane and butanone-2 is shown below:

```
>>> from thermo.unifac import UFIP, UFSG
>>> GE = UNIFAC.from_subgroups(chemgroups=[{1:2, 2:4}, {1:1, 2:1, 18:1}], T=60+273.
↪15, xs=[0.5, 0.5], version=0, interaction_data=UFIP, subgroups=UFSG)
>>> GE.gammas()
[1.4276025835, 1.3646545010]
>>> GE.GE(), GE.dGE_dT(), GE.d2GE_dT2()
(923.641197, 0.206721488, -0.00380070204)
>>> GE.HE(), GE.SE(), GE.dHE_dT(), GE.dSE_dT()
(854.77193363, -0.2067214889, 1.266203886, 0.0038007020460)
```

The solution given by the DDBST has the same values [1.428, 1.365], and can be found here: http://chemthermo.ddbst.com/Problems_Solutions/Mathcad_Files/05.22a%20VLE%20of%20Hexane-Butanone-2%20Via%20UNIFAC%20-%20Step%20by%20Step.xps

Attributes

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Methods

CpE()	Calculate and return the first temperature derivative of excess enthalpy of a liquid phase using an activity coefficient model.
Fis()	Calculate the F_i terms used in calculating the combinatorial part.
GE()	Calculate the excess Gibbs energy with the UNIFAC model.
HE()	Calculate and return the excess entropy of a liquid phase using an activity coefficient model.
SE()	Calculates the excess entropy of a liquid phase using an activity coefficient model.
Thetas()	Calculate the Θ_m parameters used in calculating the residual part.
Thetas_pure()	Calculate the Θ_m parameters for each chemical in the mixture as a pure species, used in calculating the residual part.
Vis()	Calculate the V_i terms used in calculating the combinatorial part.
Vis_modified()	Calculate the V_i' terms used in calculating the combinatorial part.
Xs()	Calculate the X_m parameters used in calculating the residual part.
Xs_pure()	Calculate the X_m parameters for each chemical in the mixture as a pure species, used in calculating the residual part.
as_json()	Method to create a JSON-friendly representation of the Gibbs Excess model which can be stored, and reloaded later.
d2Fis_dxixjs()	Calculate the second mole fraction derivative of the F_i terms used in calculating the combinatorial part.
d2GE_dT2()	Calculate the second temperature derivative of excess Gibbs energy with the UNIFAC model.
d2GE_dTdns()	Calculate and return the mole number derivative of the first temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d2GE_dTdxs()	Calculate the first composition derivative and temperature derivative of excess Gibbs energy with the UNIFAC model.
d2GE_dxixjs()	Calculate the second composition derivative of excess Gibbs energy with the UNIFAC model.
d2Thetas_dxixjs()	Calculate the mole fraction derivatives of the Θ_m parameters.

continues on next page

Table 96 – continued from previous page

d2Vis_dxixjs()	Calculate the second mole fraction derivative of the V_i terms used in calculating the combinatorial part.
d2Vis_modified_dxixjs()	Calculate the second mole fraction derivative of the V_i' terms used in calculating the combinatorial part.
d2lnGammas_subgroups_dT2()	Calculate the second temperature derivative of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
d2lnGammas_subgroups_dTdxs()	Calculate the temperature and mole fraction derivatives of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
d2lnGammas_subgroups_dxixjs()	Calculate the second mole fraction derivatives of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
d2lnGammas_subgroups_pure_dT2()	Calculate the second temperature derivative of $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.
d2lngammas_c_dT2()	Second temperature derivatives of the combinatorial part of the UNIFAC model.
d2lngammas_c_dTdx()	Second temperature derivative and first mole fraction derivative of the combinatorial part of the UNIFAC model.
d2lngammas_c_dxixjs()	Second composition derivative of the combinatorial part of the UNIFAC model.
d2lngammas_dT2()	Calculates the second temperature derivative of the residual part of the UNIFAC model.
d2lngammas_r_dT2()	Calculates the second temperature derivative of the residual part of the UNIFAC model.
d2lngammas_r_dTdxs()	Calculates the first mole fraction derivative of the temperature derivative of the residual part of the UNIFAC model.
d2lngammas_r_dxixjs()	Calculates the second mole fraction derivative of the residual part of the UNIFAC model.
d2nGE_dTdns()	Calculate and return the partial mole number derivative of the first temperature derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d2nGE_dninjs()	Calculate and return the second partial mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
d2psis_dT2()	Calculate the Ψ term second temperature derivative matrix for all groups interacting with all other groups.
d3Fis_dxixjxks()	Calculate the third mole fraction derivative of the F_i terms used in calculating the combinatorial part.
d3GE_dT3()	Calculate the third temperature derivative of excess Gibbs energy with the UNIFAC model.
d3Vis_dxixjxks()	Calculate the third mole fraction derivative of the V_i terms used in calculating the combinatorial part.
d3Vis_modified_dxixjxks()	Calculate the third mole fraction derivative of the V_i' terms used in calculating the combinatorial part.

continues on next page

Table 96 – continued from previous page

d3lnGammas_subgroups_dT3()	Calculate the third temperature derivative of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
d3lnGammas_subgroups_pure_dT3()	Calculate the third temperature derivative of $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.
d3lngammas_c_dT3()	Third temperature derivatives of the combinatorial part of the UNIFAC model.
d3lngammas_c_dxixjxks()	Third composition derivative of the combinatorial part of the UNIFAC model.
d3lngammas_dT3()	Calculates the third temperature derivative of the residual part of the UNIFAC model.
d3lngammas_r_dT3()	Calculates the third temperature derivative of the residual part of the UNIFAC model.
d3psis_dT3()	Calculate the Ψ term third temperature derivative matrix for all groups interacting with all other groups.
dFis_dxs()	Calculate the mole fraction derivative of the F_i terms used in calculating the combinatorial part.
dGE_dT()	Calculate the first temperature derivative of excess Gibbs energy with the UNIFAC model.
dGE_dns()	Calculate and return the mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
dGE_dxs()	Calculate the first composition derivative of excess Gibbs energy with the UNIFAC model.
dHE_dT()	Calculate and return the first temperature derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dHE_dns()	Calculate and return the mole number derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dHE_dxs()	Calculate and return the mole fraction derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dSE_dT()	Calculate and return the first temperature derivative of excess entropy of a liquid phase using an activity coefficient model.
dSE_dns()	Calculate and return the mole number derivative of excess entropy of a liquid phase using an activity coefficient model.
dSE_dxs()	Calculate and return the mole fraction derivative of excess entropy of a liquid phase using an activity coefficient model.
dThetas_dxs()	Calculate the mole fraction derivatives of the Θ_m parameters.
dVis_dxs()	Calculate the mole fraction derivative of the V_i terms used in calculating the combinatorial part.
dVis_modified_dxs()	Calculate the mole fraction derivative of the V_i' terms used in calculating the combinatorial part.
dgammas_dT()	Calculates the first temperature derivative of activity coefficients with the UNIFAC model.

continues on next page

Table 96 – continued from previous page

dgammas_dns()	Calculate and return the mole number derivative of activity coefficients of a liquid phase using an activity coefficient model.
dgammas_dxs()	Calculates the first mole fraction derivative of activity coefficients with the UNIFAC model.
dlnGammas_subgroups_dT()	Calculate the first temperature derivative of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
dlnGammas_subgroups_dxs()	Calculate the mole fraction derivatives of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
dlnGammas_subgroups_pure_dT()	Calculate the first temperature derivative of $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.
dlngammas_c_dT()	Temperature derivatives of the combinatorial part of the UNIFAC model.
dlngammas_c_dxs()	First composition derivative of the combinatorial part of the UNIFAC model.
dlngammas_dT()	Calculates the first temperature derivative of the residual part of the UNIFAC model.
dlngammas_r_dT()	Calculates the first temperature derivative of the residual part of the UNIFAC model.
dlngammas_r_dxs()	Calculates the first mole fraction derivative of the residual part of the UNIFAC model.
dnGE_dns()	Calculate and return the partial mole number derivative of excess Gibbs energy of a liquid phase using an activity coefficient model.
dnHE_dns()	Calculate and return the partial mole number derivative of excess enthalpy of a liquid phase using an activity coefficient model.
dnSE_dns()	Calculate and return the partial mole number derivative of excess entropy of a liquid phase using an activity coefficient model.
dpsis_dT()	Calculate the Ψ term first temperature derivative matrix for all groups interacting with all other groups.
from_json(json_repr)	Method to create a Gibbs Excess model from a JSON-friendly serialization of another Gibbs Excess model.
from_subgroups(T, xs, chemgroups[, ...])	Method to construct a UNIFAC object from a dictionary of interaction parameters parameters and a list of dictionaries of UNIFAC keys.
gammas()	Calculates the activity coefficients with the UNIFAC model.
gammas_infinite_dilution()	Calculate and return the infinite dilution activity coefficients of each component.
lnGammas_subgroups()	Calculate the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.
lnGammas_subgroups_pure()	Calculate the $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.
lngammas_c()	Calculates the combinatorial part of the UNIFAC model.
lngammas_r()	Calculates the residual part of the UNIFAC model.

continues on next page

Table 96 – continued from previous page

model_hash()	Basic method to calculate a hash of the non-state parts of the model This is useful for comparing to models to determine if they are the same, i.e. in a VLL flash it is important to know if both liquids have the same model.
psis()	Calculate the Ψ term matrix for all groups interacting with all other groups.
state_hash()	Basic method to calculate a hash of the state of the model and its model parameters.
to_T_xs(T, xs)	Method to construct a new UNIFAC instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

lnphis_args	

Fis()

Calculate the F_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters q only.

$$F_i = \frac{q_i}{\sum_j q_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

Fis [list[float]] F terms size number of components, [-]

GE()

Calculate the excess Gibbs energy with the UNIFAC model.

$$G^E = RT \sum_i x_i \left(\ln \gamma_i^c + \ln \gamma_i^r\right)$$

For the VTPR model, the combinatorial component is set to zero.

Returns

GE [float] Excess Gibbs energy, [J/mol]

Thetas()

Calculate the Θ_m parameters used in calculating the residual part. A function of mole fractions and group counts only.

$$\Theta_m = \frac{Q_m X_m}{\sum_n Q_n X_n}$$

Returns

Thetas [list[float]] Θ_m terms, size number of subgroups, [-]

Thetas_pure()

Calculate the Θ_m parameters for each chemical in the mixture as a pure species, used in calculating the residual part. A function of group counts only.

$$\Theta_m = \frac{Q_m X_m}{\sum_n Q_n X_n}$$

Returns

Thetas_pure [list[list[float]]] Θ_m terms, size number of components by number of subgroups and indexed in that order, [-]

Vis()

Calculate the V_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$V_i = \frac{r_i}{\sum_j r_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

Vis [list[float]] *V* terms size number of components, [-]

Vis_modified()

Calculate the V_i' terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$V_i' = \frac{r_i^n}{\sum_j r_j^n x_j}$$

This is used in the UNIFAC Dortmund and UNIFAC-NIST model with n=0.75, and the Lyngby model with n=2/3.

Returns

Vis_modified [list[float]] Modified *V* terms size number of components, [-]

Xs()

Calculate the X_m parameters used in calculating the residual part. A function of mole fractions and group counts only.

$$X_m = \frac{\sum_j \nu_m^j x_j}{\sum_j \sum_n \nu_n^j x_j}$$

Returns

Xs [list[float]] X_m terms, size number of subgroups, [-]

Xs_pure()

Calculate the X_m parameters for each chemical in the mixture as a pure species, used in calculating the residual part. A function of group counts only, not even mole fractions or temperature.

$$X_m = \frac{\nu_m}{\sum_n^{gr} \nu_n}$$

Returns

Xs_pure [list[list[float]]] X_m terms, size number of subgroups by number of components and indexed in that order, [-]

d2Fis_dxixjs()

Calculate the second mole fraction derivative of the F_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters q only.

$$\frac{\partial F_i}{\partial x_j \partial x_k} = 2 q_i q_j q_k G_{sum}^3$$

$$G_{sum} = \frac{1}{\sum_j q_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

d2Fis_dxixjs [list[list[list[float]]]] F terms size number of components by number of components by number of components, [-]

d2GE_dT2()

Calculate the second temperature derivative of excess Gibbs energy with the UNIFAC model.

$$\frac{\partial^2 G^E}{\partial T^2} = RT\sum_i x_i \frac{\partial^2 \ln \gamma_i^r}{\partial T^2} + 2R\sum_i x_i \frac{\partial \ln \gamma_i^r}{\partial T}$$

Returns

d2GE_dT2 [float] Second temperature derivative of excess Gibbs energy, [J/mol/K^2]

d2GE_dTdxs()

Calculate the first composition derivative and temperature derivative of excess Gibbs energy with the UNIFAC model.

$$\frac{\partial^2 G^E}{\partial T \partial x_i} = RT\left(\frac{\partial \ln \gamma_i^r}{\partial T} + \sum_j x_j \frac{\partial \ln \gamma_j^r}{\partial x_i}\right) + R\left[\frac{\partial \ln \gamma_i^c}{\partial x_i} + \frac{\partial \ln \gamma_i^r}{\partial x_i} + \sum_j x_j\left(\frac{\partial \ln \gamma_j^c}{\partial x_i} + \frac{\partial \ln \gamma_j^r}{\partial x_i}\right)\right]$$

Returns

dGE_dxs [list[float]] First composition derivative and first temperature derivative of excess Gibbs energy, [J/mol/K]

d2GE_dxixjs()

Calculate the second composition derivative of excess Gibbs energy with the UNIFAC model.

$$\frac{\partial^2 G^E}{\partial x_j \partial x_k} = RT\left[\sum_i\left(\frac{\partial \ln \gamma_i^c}{\partial x_j \partial x_k} + \frac{\partial \ln \gamma_i^r}{\partial x_j \partial x_k}\right) + \frac{\partial \ln \gamma_j^c}{\partial x_k} + \frac{\partial \ln \gamma_j^r}{\partial x_k} + \frac{\partial \ln \gamma_k^c}{\partial x_j} + \frac{\partial \ln \gamma_k^r}{\partial x_j}\right]$$

Returns

d2GE_dxixjs [list[list[float]]] Second composition derivative of excess Gibbs energy, [J/mol]

d2Thetas_dxixjs()

Calculate the mole fraction derivatives of the Θ_m parameters. A function of mole fractions and group counts only.

$$\frac{\partial^2 \Theta_i}{\partial x_j \partial x_k} = \frac{Q_i}{\sum_n Q_n (\nu x)_{sum,n}}\left[-F(\nu)_{sum,j}\nu_{i,k} - F(\nu)_{sum,k}\nu_{i,j} + 2F^2(\nu)_{sum,j}(\nu)_{sum,k}(\nu x)_{sum,i} + \frac{F(\nu x)_{sum,i}\left[\sum_n(\right.}{}\right.$$

$$G = \frac{1}{\sum_j Q_j X_j}$$

$$F = \frac{1}{\sum_j \sum_n \nu_n^j x_j}$$

$$(\nu)_{sum,i} = \sum_j \nu_{j,i}$$

$$(\nu x)_{sum,i} = \sum_j \nu_{i,j} x_j$$

Returns

d2Thetas_dxixjs [list[list[list[float]]]] Θ_m terms, size number of subgroups by mole fractions and indexed in that order, [-]

`d2Vis_dxixjs()`

Calculate the second mole fraction derivative of the V_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$\frac{\partial V_i}{\partial x_j \partial x_k} = 2 r_i r_j r_k V_{sum}^3$$

$$V_{sum} = \frac{1}{\sum_j r_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

d2Vis_dxixjs [list[list[list[float]]]] *V* terms size number of components by number of components by number of components, [-]

`d2Vis_modified_dxixjs()`

Calculate the second mole fraction derivative of the V_i' terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$\frac{\partial V_i'}{\partial x_j \partial x_k} = 2 r_i^n r_j^n r_k^n V_{sum}^3$$

$$V_{sum} = \frac{1}{\sum_j r_j^n x_j}$$

This is used in the UNIFAC Dortmund and UNIFAC-NIST model with n=0.75, and the Lyngby model with n=2/3.

Returns

d2Vis_modified_dxixjs [list[list[list[float]]]] *V'* terms size number of components by number of components by number of components, [-]

`d2lnGammas_subgroups_dT2()`

Calculate the second temperature derivative of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\frac{\partial^2 \ln \Gamma_i}{\partial T^2} = -Q_i \left[Z(i)G(i) - F(i)^2 Z(i)^2 + \sum_j \left(\theta_j Z(j) \frac{\partial^2 \psi_{i,j}}{\partial T} - Z(j)^2 \left(G(j)\theta_j \psi_{i,j} + 2 F_j \theta_j \frac{\partial \psi_{i,j}}{\partial T} \right) + 2 Z(j)^3 F(j)^2 \theta \right.\right.$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

$$G(k) = \sum_m^{gr} \theta_m \frac{\partial^2 \psi_{m,k}}{\partial T^2}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{m,k}}$$

Returns

d2lnGammas_subgroups_dT2 [list[float]] Second temperature derivative of ln Gamma parameters for each subgroup, size number of subgroups, [1/K^2]

d2lnGammas_subgroups_dTdxs()

Calculate the temperature and mole fraction derivatives of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\frac{\partial^2 \ln \Gamma_k}{\partial x_i \partial T} = -Q_k \left(D(k,i)Z(k) - B(k)W(k,i)Z(k)^2 + \sum_m^{gr}(Z(m)\frac{\partial \theta_m}{\partial x_i}\frac{\partial \psi_{k,m}}{\partial T}) - \sum_m^{gr}(B(m)Z(m)^2\psi_{k,m}\frac{\partial \theta_m}{\partial x_i}) - \sum_m^{gr}(D$$

The following groups are used as follows to simplfy the number of evaluations:

$$W(k,i) = \sum_m^{gr} \psi_{m,k}\frac{\partial \theta_m}{\partial x_i}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{mk}}$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

In the below expression, k` refers to a group, and *i* refers to a component.

$$D(k,i) = \sum_m^{gr} \frac{\partial \theta_m}{\partial x_i}\frac{\partial \psi_{m,k}}{\partial T}$$

Returns

d2lnGammas_subgroups_dTdxs [list[list[float]]] Temperature and mole fraction derivatives of Gamma parameters for each subgroup, size number of subgroups by number of components and indexed in that order, [1/K]

d2lnGammas_subgroups_dxixjs()

Calculate the second mole fraction derivatives of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\frac{\partial^2 \ln \Gamma_k}{\partial x_i \partial x_j} = -Q_k \left(-Z(k)K(k,i,j) - \sum_m^{gr} Z(m)^2 K(m,i,j)\theta_m \psi_{k,m} - W(k,i)W(k,j)Z(k)^2 + \sum_m^{gr} Z_m \psi_{k,m}\frac{\partial^2 \theta_m}{\partial x_i \partial x_j} - \sum\right.$$

The following groups are used as follows to simplfy the number of evaluations:

$$W(k,i) = \sum_m^{gr} \psi_{m,k}\frac{\partial \theta_m}{\partial x_i}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{mk}}$$

$$K(k,i,j) = \sum_m^{gr} \psi_{m,k}\frac{\partial^2 \theta_m}{\partial x_i \partial x_j}$$

Returns

d2lnGammas_subgroups_dxixjs [list[list[list[float]]]] Second mole fraction derivatives of Gamma parameters for each subgroup, size number of components by number of components by number of subgroups and indexed in that order, [-]

d2lnGammas_subgroups_pure_dT2()

Calculate the second temperature derivative of $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.

$$\frac{\partial^2 \ln \Gamma_i}{\partial T^2} = -Q_i \left[Z(i)G(i) - F(i)^2 Z(i)^2 + \sum_j \left(\theta_j Z(j)\frac{\partial^2 \psi_{i,j}}{\partial T} - Z(j)^2 \left(G(j)\theta_j \psi_{i,j} + 2F_j \theta_j \frac{\partial \psi_{i,j}}{\partial T}\right) + 2Z(j)^3 F(j)^2 \theta\right.\right.$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

$$G(k) = \sum_m^{gr} \theta_m \frac{\partial^2 \psi_{m,k}}{\partial T^2}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{m,k}}$$

In this model, the Θ values come from the *UNIFAC.Thetas_pure* method, where each compound is assumed to be pure.

Returns

d2lnGammas_subgroups_pure_dT2 [list[list[float]]] Second temperature derivative of ln Gamma parameters for each subgroup, size number of subgroups by number of components and indexed in that order, [1/K^2]

`d2lngammas_c_dT2()`

Second temperature derivatives of the combinatorial part of the UNIFAC model. Zero in all variations.

$$\frac{\partial^2 \ln \gamma_i^c}{\partial T^2} = 0$$

Returns

d2lngammas_c_dT2 [list[float]] Combinatorial lngammas term second temperature derivatives, size number of components, [-]

`d2lngammas_c_dTdx()`

Second temperature derivative and first mole fraction derivative of the combinatorial part of the UNIFAC model. Zero in all variations.

$$\frac{\partial^3 \ln \gamma_i^c}{\partial T^2 \partial x_j} = 0$$

Returns

d2lngammas_c_dTdx [list[list[float]]] Combinatorial lngammas term second temperature derivatives, size number of components by number of components, [-]

`d2lngammas_c_dxixjs()`

Second composition derivative of the combinatorial part of the UNIFAC model. For the modified UNIFAC model, the equation is as follows; for the original UNIFAC and UNIFAC LLE, replace V_i' with V_i.

$$\frac{\partial \ln \gamma_i^c}{\partial x_j \partial x_k} = 5q_i \left(\frac{- \frac{d^2}{dx_k dx_j} V_i + \frac{V_i \frac{d^2}{dx_k dx_j} F_i}{F_i} + \frac{\frac{d}{dx_j} F_i \frac{d}{dx_k} V_i}{F_i} + \frac{\frac{d}{dx_k} F_i \frac{d}{dx_j} V_i}{F_i} - \frac{2V_i \frac{d}{dx_j} F_i \frac{d}{dx_k} F_i}{F_i^2}}{V_i} + \frac{\left(\frac{d}{dx_j} V_i - \frac{V_i \frac{d}{dx_j} F_i}{F_i}\right) \frac{d}{dx_k} V_i}{V_i^2} +$$

For the Lyngby model, the following equations are used:

$$\frac{\partial^2 \ln \gamma_i^c}{\partial x_j \partial x_k} = -\frac{\partial^2 V_i'}{\partial x_j \partial x_k} + \frac{1}{V_i'} \frac{\partial^2 V_i'}{\partial x_j \partial x_k} - \frac{1}{(V_i')^2} \frac{\partial V_i'}{\partial x_j} \frac{\partial V_i'}{\partial x_k}$$

Returns

d2lngammas_c_dxixjs [list[list[list[float]]]] Combinatorial lngammas term second composition derivative, size number of components by number of components by number of components, [-] `d2lngammas_dT2()`

Calculates the second temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial^2 \ln \gamma_i^r}{\partial T^2} = \sum_k^{gr} \nu_k^{(i)} \left[\frac{\partial^2 \ln \Gamma_k}{\partial T^2} - \frac{\partial^2 \ln \Gamma_k^{(i)}}{\partial T^2}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

d2lngammas_r_dT2 [list[float]] Residual lngammas terms second temperature derivative, size number of components [1/K^2]

`d2lngammas_r_dT2()`

Calculates the second temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial^2 \ln \gamma_i^r}{\partial T^2} = \sum_k^{gr} \nu_k^{(i)} \left[\frac{\partial^2 \ln \Gamma_k}{\partial T^2} - \frac{\partial^2 \ln \Gamma_k^{(i)}}{\partial T^2}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

d2lngammas_r_dT2 [list[float]] Residual lngammas terms second temperature derivative, size number of components [1/K^2]

`d2lngammas_r_dTdxs()`

Calculates the first mole fraction derivative of the temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial^2 \ln \gamma_i^r}{\partial x_j \partial T} = \sum_m^{gr} \nu_m^{(i)} \frac{\partial^2 \ln \Gamma_m}{\partial x_j \partial T}$$

Returns

d2lngammas_r_dTdxs [list[list[float]]] First mole fraction derivative and temperature derivative of residual lngammas terms, size number of components by number of components [-]

`d2lngammas_r_dxixjs()`

Calculates the second mole fraction derivative of the residual part of the UNIFAC model.

$$\frac{\partial^2 \ln \gamma_i^r}{\partial x_j^2} = \sum_m^{gr} \nu_m^{(i)} \frac{\partial^2 \ln \Gamma_m}{\partial x_j^2}$$

Returns

d2lngammas_r_dxixjs [list[list[list[float]]]] Second mole fraction derivative of the residual lngammas terms, size number of components by number of components by number of components [-]

`d2psis_dT2()`

Calculate the Ψ term second temperature derivative matrix for all groups interacting with all other groups.

The main model calculates the derivative as a function of three coefficients;

$$\frac{\partial^2 \Psi_{mn}}{\partial T^2} = \frac{\left(-2c_{mn} + \frac{2(2Tc_{mn} + b_{mn})}{T} + \frac{\left(2Tc_{mn} + b_{mn} - \frac{T^2 c_{mn} + Tb_{mn} + a_{mn}}{T}\right)^2}{T} - \frac{2\left(T^2 c_{mn} + Tb_{mn} + a_{mn}\right)}{T^2}\right) e^{-\frac{T^2 c_{mn} + Tb_{mn} + a_{mn}}{T}}}{T}$$

Only the first, *a* coefficient, is used in the original UNIFAC model as well as the UNIFAC-LLE model, so the expression simplifies to:

$$\frac{\partial^2 \Psi_{mn}}{\partial T^2} = \frac{a_{mn}\left(-2 + \frac{a_{mn}}{T}\right) e^{-\frac{a_{mn}}{T}}}{T^3}$$

For the Lyngby model, the second temperature derivative is:

$$\frac{\partial^2 \Psi_{mk}}{\partial T^2} = \frac{\left(2a_2 + 2a_3\ln\left(\frac{T_0}{T}\right) + a_3 + \left(a_2 + a_3\ln\left(\frac{T_0}{T}\right) - \frac{a_1 + a_2(T-T_0) + a_3\left(T\ln\left(\frac{T_0}{T}\right) + T - T_0\right)}{T}\right)^2 - \frac{2\left(a_1 + a_2(T-T_0) + a_3\left(T\ln \cdots\right.\right.}{T}\right.}{T^2}$$

with $T_0 = 298.15$ K and the *a* coefficients are specific to each pair of main groups, and they are asymmetric, so $a_{0,mk} \neq a_{0,km}$.

Returns

d2psis_dT2 [list[list[float]]] Second temperature derivative of`psi` terms, size subgroups x subgroups [-]

d3Fis_dxixjxks()

Calculate the third mole fraction derivative of the F_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters *q* only.

$$\frac{\partial F_i}{\partial x_j \partial x_k \partial x_m} = -6 q_i q_j q_k q_m G_{sum}^4$$

$$G_{sum} = \frac{1}{\sum_j q_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

d3Fis_dxixjxks [list[list[list[list[float]]]]] *F* terms size number of components by number of components by number of components by number of components, [-]

d3GE_dT3()

Calculate the third temperature derivative of excess Gibbs energy with the UNIFAC model.

$$\frac{\partial^3 G^E}{\partial T^3} = RT\sum_i x_i \frac{\partial^3 \ln \gamma_i^r}{\partial T^3} + 3R\sum_i x_i \frac{\partial^2 \ln \gamma_i^r}{\partial T^2}$$

Returns

d3GE_dT3 [float] Third temperature derivative of excess Gibbs energy, [J/mol/K^3]

d3Vis_dxixjxks()

Calculate the third mole fraction derivative of the V_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$\frac{\partial V_i}{\partial x_j \partial x_k \partial x_m} = -6 r_i r_j r_k r_m V_{sum}^4$$

$$V_{sum} = \frac{1}{\sum_j r_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

d3Vis_dxixjxks [list[list[list[list[float]]]]] *V* terms size number of components by number of components by number of components by number of components, [-]

d3Vis_modified_dxixjxks()

Calculate the third mole fraction derivative of the V_i' terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$\frac{\partial V_i'}{\partial x_j \partial x_k \partial x_m} = -6 r_i^n r_j^n r_k^n r_m^n V_{sum}^4$$

$$V_{sum} = \frac{1}{\sum_j r_j x_j}$$

This is used in the UNIFAC Dortmund and UNIFAC-NIST model with n=0.75, and the Lyngby model with n=2/3.

Returns

d3Vis_modified_dxixjxks [list[list[list[list[float]]]]] *V'* terms size number of components by number of components by number of components by number of components, [-]

d3lnGammas_subgroups_dT3()

Calculate the third temperature derivative of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\frac{\partial^3 \ln \Gamma_i}{\partial T^3} = Q_i \left[-H(i)Z(i) - 2F(i)^3 Z(i)^3 + 3F(i)G(i)Z(i)^2 + \left(-\theta_j Z(j) \frac{\partial^3 \psi}{\partial T^3} + H(j)Z(j)^2 \theta(j) \psi_{i,j} - 6F(j)^2 Z(j)^3 \theta \right. \right.$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

$$G(k) = \sum_m^{gr} \theta_m \frac{\partial^2 \psi_{m,k}}{\partial T^2}$$

$$H(k) = \sum_m^{gr} \theta_m \frac{\partial^3 \psi_{m,k}}{\partial T^3}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{m,k}}$$

Returns

d3lnGammas_subgroups_dT3 [list[float]] Third temperature derivative of ln Gamma parameters for each subgroup, size number of subgroups, [1/K^3]

d3lnGammas_subgroups_pure_dT3()

Calculate the third temperature derivative of $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.

$$\frac{\partial^3 \ln \Gamma_i}{\partial T^3} = Q_i \left[-H(i)Z(i) - 2F(i)^3 Z(i)^3 + 3F(i)G(i)Z(i)^2 + \left(-\theta_j Z(j) \frac{\partial^3 \psi}{\partial T^3} + H(j)Z(j)^2 \theta(j) \psi_{i,j} - 6F(j)^2 Z(j)^3 \theta \right. \right.$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

$$G(k) = \sum_m^{gr} \theta_m \frac{\partial^2 \psi_{m,k}}{\partial T^2}$$

$$H(k) = \sum_m^{gr} \theta_m \frac{\partial^3 \psi_{m,k}}{\partial T^3}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{m,k}}$$

In this model, the Θ values come from the *UNIFAC.Thetas_pure* method, where each compound is assumed to be pure.

Returns

d3lnGammas_subgroups_pure_dT3 [list[list[float]]] Third temperature derivative of ln Gamma parameters for each subgroup, size number of subgroups by number of components and indexed in that order, [1/K^3]

`d3lngammas_c_dT3()`

Third temperature derivatives of the combinatorial part of the UNIFAC model. Zero in all variations.

$$\frac{\partial^3 \ln \gamma_i^c}{\partial T^3} = 0$$

Returns

d3lngammas_c_dT3 [list[float]] Combinatorial lngammas term second temperature derivatives, size number of components, [-]

`d3lngammas_c_dxixjxks()`

Third composition derivative of the combinatorial part of the UNIFAC model. For the modified UNIFAC model, the equation is as follows; for the original UNIFAC and UNIFAC LLE, replace V_i' with V_i.

$$\frac{\partial \ln \gamma_i^c}{\partial x_j \partial x_k \partial x_m} = -\frac{d^3}{dx_m dx_k dx_j} Vi' + \frac{\frac{d^3}{dx_m dx_k dx_j} Vi'}{Vi'} - \frac{\frac{d}{dx_j} Vi' \frac{d^2}{dx_m dx_k} Vi'}{Vi'^2} - \frac{\frac{d}{dx_k} Vi' \frac{d^2}{dx_m dx_j} Vi'}{Vi'^2} - \frac{\frac{d}{dx_m} Vi' \frac{d^2}{dx_k dx_j} Vi'}{Vi'^2} + $$

For the Lyngby model, the following equations are used:

$$\frac{\partial^3 \ln \gamma_i^c}{\partial x_j \partial x_k \partial x_m} = \frac{\partial^3 V_i'}{\partial x_j \partial x_k \partial x_m}\left(\frac{1}{V_i'} - 1\right) - \frac{1}{(V_i')^2}\left(\frac{\partial V_i'}{\partial x_j}\frac{\partial V_i'}{\partial x_k \partial x_m} + \frac{\partial V_i'}{\partial x_k}\frac{\partial V_i'}{\partial x_j \partial x_m} + \frac{\partial V_i'}{\partial x_m}\frac{\partial V_i'}{\partial x_j \partial x_k}\right) + \frac{2}{(V_i')^3}\frac{\partial V_i'}{\partial x_j}\frac{\partial V_i'}{\partial x_k}$$

Returns

d3lngammas_c_dxixjxks [list[list[list[list[float]]]]] Combinatorial lngammas term third composition derivative, size number of components by number of components by number of components by number of components, [-]

`d3lngammas_dT3()`

Calculates the third temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial^3 \ln \gamma_i^r}{\partial T^3} = \sum_k^{gr} \nu_k^{(i)} \left[\frac{\partial^2 3 \ln \Gamma_k}{\partial T^3} - \frac{\partial^3 \ln \Gamma_k^{(i)}}{\partial T^3}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

d3lngammas_r_dT3 [list[float]] Residual lngammas terms third temperature derivative, size number of components [1/K^3]

`d3lngammas_r_dT3()`

Calculates the third temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial^3 \ln \gamma_i^r}{\partial T^3} = \sum_k^{gr} \nu_k^{(i)} \left[\frac{\partial^2 3 \ln \Gamma_k}{\partial T^3} - \frac{\partial^3 \ln \Gamma_k^{(i)}}{\partial T^3}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

d3lngammas_r_dT3 [list[float]] Residual lngammas terms third temperature derivative, size number of components [1/K^3] `d3psis_dT3()`

Calculate the Ψ term third temperature derivative matrix for all groups interacting with all other groups.

The main model calculates the derivative as a function of three coefficients;

$$\frac{\partial^3 \Psi_{mn}}{\partial T^3} = \frac{\Bigg(6c_{mn} + 6\bigg(c_{mn} - \frac{2Tc_{mn}+b_{mn}}{T} + \frac{T^2c_{mn}+Tb_{mn}+a_{mn}}{T^2}\bigg)\bigg(2Tc_{mn} + b_{mn} - \frac{T^2c_{mn}+Tb_{mn}+a_{mn}}{T}\bigg) - \frac{6(2Tc_{mn}+b_{mn}}{T}}{T^2}$$

Only the first, *a* coefficient, is used in the original UNIFAC model as well as the UNIFAC-LLE model, so the expression simplifies to:

$$\frac{\partial^3 \Psi_{mn}}{\partial T^3} = \frac{a_{mn}\left(6 - \frac{6a_{mn}}{T} + \frac{a_{mn}^2}{T^2}\right)e^{-\frac{a_{mn}}{T}}}{T^4}$$

For the Lyngby model, the third temperature derivative is:

$$\frac{\partial^3 \Psi_{mk}}{\partial T^3} = -\frac{\Bigg(6a_2 + 6a_3\ln\left(\frac{T_0}{T}\right) + 4a_3 + \bigg(a_2 + a_3\ln\left(\frac{T_0}{T}\right) - \frac{a_1+a_2(T-T_0)+a_3\left(T\ln\left(\frac{T_0}{T}\right)+T-T_0\right)}{T}\bigg)^3 + 3\bigg(a_2 + a_3\ln\left(\frac{T_0}{T}\right)}{}$$

with $T_0 = 298.15$ K and the *a* coefficients are specific to each pair of main groups, and they are asymmetric, so $a_{0,mk} \neq a_{0,km}$.

Returns

d3psis_dT3 [list[list[float]]] Third temperature derivative of`psi` terms, size subgroups x subgroups [-]

`dFis_dxs()`

Calculate the mole fraction derivative of the F_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters *q* only.

$$\frac{\partial F_i}{\partial x_j} = -q_i q_j G_{sum}^2$$

$$G_{sum} = \frac{1}{\sum_j q_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

dFis_dxs [list[list[float]]] *F* terms size number of components by number of components, [-]

`dGE_dT()`

Calculate the first temperature derivative of excess Gibbs energy with the UNIFAC model.

$$\frac{\partial G^E}{\partial T} = RT\sum_i x_i \frac{\partial \ln \gamma_i^r}{\partial T} + \frac{G^E}{T}$$

Returns

dGE_dT [float] First temperature derivative of excess Gibbs energy, [J/mol/K]

`dGE_dxs()`

Calculate the first composition derivative of excess Gibbs energy with the UNIFAC model.

$$\frac{\partial G^E}{\partial x_i} = RT(\ln \gamma_i^c + \ln \gamma_i^r) + RT\sum_j x_j \left(\frac{\partial \ln \gamma_j^c}{\partial x_i} + \frac{\partial \ln \gamma_j^r}{\partial x_i}\right)$$

Returns

dGE_dxs [list[float]] First composition derivative of excess Gibbs energy, [J/mol]

dThetas_dxs()

Calculate the mole fraction derivatives of the Θ_m parameters. A function of mole fractions and group counts only.

$$\frac{\partial \Theta_i}{\partial x_j} = FGQ_i\left[FG(\nu x)_{sum,i}\left(\sum_k^{gr} FQ_k(\nu)_{sum,j}(\nu x)_{sum,k} - \sum_k^{gr} Q_k\nu_{k,j}\right) - F(\nu)_{sum,j}(\nu x)_{sum,i} + \nu_{ij}\right]$$

$$G = \frac{1}{\sum_j Q_j X_j}$$

$$F = \frac{1}{\sum_j \sum_n \nu_n^j x_j}$$

$$(\nu)_{sum,i} = \sum_j \nu_{j,i}$$

$$(\nu x)_{sum,i} = \sum_j \nu_{i,j} x_j$$

Returns

dThetas_dxs [list[list[float]]] Mole fraction derivatives of Θ_m terms, size number of subgroups by mole fractions and indexed in that order, [-]

dVis_dxs()

Calculate the mole fraction derivative of the V_i terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$\frac{\partial V_i}{\partial x_j} = -r_i r_j V_{sum}^2$$

$$V_{sum} = \frac{1}{\sum_j r_j x_j}$$

This is used in the UNIFAC, UNIFAC-LLE, UNIFAC Dortmund, UNIFAC-NIST, and PSRK models.

Returns

dVis_dxs [list[list[float]]] *V* terms size number of components by number of components, [-]

dVis_modified_dxs()

Calculate the mole fraction derivative of the V_i' terms used in calculating the combinatorial part. A function of mole fractions and the parameters *r* only.

$$\frac{\partial V_i'}{\partial x_j} = -r_i^n r_j^n V_{sum}^2$$

$$V_{sum} = \frac{1}{\sum_j r_j^n x_j}$$

This is used in the UNIFAC Dortmund and UNIFAC-NIST model with n=0.75, and the Lyngby model with n=2/3.

Returns

dVis_modified_dxs [list[list[float]]] *V'* terms size number of components by number of components, [-]

dgammas_dT()
Calculates the first temperature derivative of activity coefficients with the UNIFAC model.

$$\frac{\partial \gamma_i}{\partial T} = \gamma_i \frac{\partial \ln \gamma_i^r}{\partial T}$$

Returns

dgammas_dT [list[float]] First temperature derivative of activity coefficients, size number of components [1/K]

dgammas_dns()
Calculate and return the mole number derivative of activity coefficients of a liquid phase using an activity coefficient model.

$$\frac{\partial \gamma_i}{\partial n_i} = \gamma_i \left(\frac{\frac{\partial^2 G^E}{\partial x_i \partial x_j}}{RT}\right)$$

Returns

dgammas_dns [list[list[float]]] Mole number derivatives of activity coefficients, [1/mol]

dgammas_dxs()
Calculates the first mole fraction derivative of activity coefficients with the UNIFAC model.

$$\frac{\partial \gamma_i}{\partial x_j} = \gamma_i \left(\frac{\partial \ln \gamma_i^r}{\partial x_j} + \frac{\partial \ln \gamma_i^c}{\partial x_j}\right)$$

For the VTPR variant, the combinatorial part is skipped:

$$\frac{\partial \gamma_i}{\partial x_j} = \gamma_i \left(\frac{\partial \ln \gamma_i^r}{\partial x_j}\right)$$

Returns

dgammas_dxs [list[list[float]]] First mole fraction derivative of activity coefficients, size number of components by number of components [-]

dlnGammas_subgroups_dT()
Calculate the first temperature derivative of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\frac{\partial \ln \Gamma_i}{\partial T} = Q_i \left(\sum_j^{gr} Z(j) \left[\theta_j \frac{\partial \psi_{i,j}}{\partial T} + \theta_j \psi_{i,j} F(j) Z(j)\right] - F(i)Z(i)\right)$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{m,k}}$$

Returns

dlnGammas_subgroups_dT [list[float]] First temperature derivative of ln Gamma parameters for each subgroup, size number of subgroups, [1/K]

dlnGammas_subgroups_dxs()
Calculate the mole fraction derivatives of the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\frac{\partial \ln \Gamma_k}{\partial x_i} = Q_k \left(-\frac{\sum_m^{gr} \psi_{m,k} \frac{\partial \theta_m}{\partial x_i}}{\sum_m^{gr} \theta_m \psi_{m,k}} - \sum_m^{gr} \frac{\psi_{k,m} \frac{\partial \theta_m}{\partial x_i}}{\sum_n^{gr} \theta_n \psi_{n,m}} + \sum_m^{gr} \frac{(\sum_n^{gr} \psi_{n,m} \frac{\partial \theta_n}{\partial x_i})\theta_m \psi_{k,m}}{(\sum_n^{gr} \theta_n \psi_{n,m})^2}\right)$$

The group W is used internally as follows to simplfy the number of evaluations.

$$W(k,i) = \sum_m^{gr} \psi_{m,k} \frac{\partial \theta_m}{\partial x_i}$$

Returns

dlnGammas_subgroups_dxs [list[list[float]]] Mole fraction derivatives of Gamma parameters for each subgroup, size number of subgroups by number of components and indexed in that order, [-]

`dlnGammas_subgroups_pure_dT()`

Calculate the first temperature derivative of $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.

$$\frac{\partial \ln \Gamma_i}{\partial T} = Q_i\left(\sum_j^{gr} Z(j)\left[\theta_j \frac{\partial \psi_{i,j}}{\partial T} + \theta_j \psi_{i,j} F(j) Z(j)\right] - F(i)Z(i)\right)$$

$$F(k) = \sum_m^{gr} \theta_m \frac{\partial \psi_{m,k}}{\partial T}$$

$$Z(k) = \frac{1}{\sum_m \Theta_m \Psi_{m,k}}$$

In this model, the Θ values come from the *UNIFAC.Thetas_pure* method, where each compound is assumed to be pure.

Returns

dlnGammas_subgroups_pure_dT [list[list[float]]] First temperature derivative of ln Gamma parameters for each subgroup, size number of subgroups by number of components and indexed in that order, [1/K]

`dlngammas_c_dT()`

Temperature derivatives of the combinatorial part of the UNIFAC model. Zero in all variations.

$$\frac{\partial \ln \gamma_i^c}{\partial T} = 0$$

Returns

dlngammas_c_dT [list[float]] Combinatorial lngammas term temperature derivatives, size number of components, [-]

`dlngammas_c_dxs()`

First composition derivative of the combinatorial part of the UNIFAC model. For the modified UNIFAC model, the equation is as follows; for the original UNIFAC and UNIFAC LLE, replace V_i' with V_i.

$$\frac{\partial \ln \gamma_i^c}{\partial x_j} = -5q_i\left[\left(\frac{\frac{\partial V_i}{\partial x_j}}{F_i} - \frac{V_i \frac{\partial F_i}{\partial x_j}}{F_i^2}\right)\frac{F_i}{V_i} - \frac{\frac{\partial V_i}{\partial x_j}}{F_i} + \frac{V_i \frac{\partial F_i}{\partial x_j}}{F_i^2}\right] - \frac{\partial V_i'}{\partial x_j} + \frac{\frac{\partial V_i'}{\partial x_j}}{V_i'}$$

For the Lyngby model, the following equations are used:

$$\frac{\partial \ln \gamma_i^c}{\partial x_j} = \frac{-\partial V_i'}{\partial x_j} + \frac{1}{V_i'}\frac{\partial V_i'}{\partial x_j}$$

Returns

dlngammas_c_dxs [list[list[float]]] Combinatorial lngammas term first composition derivative, size number of components by number of components, [-]

dlngammas_dT()

Calculates the first temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial \ln \gamma_i^r}{\partial T} = \sum_k^{gr} \nu_k^{(i)} \left[\frac{\partial \ln \Gamma_k}{\partial T} - \frac{\partial \ln \Gamma_k^{(i)}}{\partial T}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

dlngammas_r_dT [list[float]] Residual lngammas terms first temperature derivative, size number of components [1/K]

dlngammas_r_dT()

Calculates the first temperature derivative of the residual part of the UNIFAC model.

$$\frac{\partial \ln \gamma_i^r}{\partial T} = \sum_k^{gr} \nu_k^{(i)} \left[\frac{\partial \ln \Gamma_k}{\partial T} - \frac{\partial \ln \Gamma_k^{(i)}}{\partial T}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

dlngammas_r_dT [list[float]] Residual lngammas terms first temperature derivative, size number of components [1/K]

dlngammas_r_dxs()

Calculates the first mole fraction derivative of the residual part of the UNIFAC model.

$$\frac{\partial \ln \gamma_i^r}{\partial x_j} = \sum_m^{gr} \nu_m^{(i)} \frac{\partial \ln \Gamma_m}{\partial x_j}$$

Returns

dlngammas_r_dxs [list[list[float]]] First mole fraction derivative of residual lngammas terms, size number of components by number of components [-]

dpsis_dT()

Calculate the Ψ term first temperature derivative matrix for all groups interacting with all other groups.

The main model calculates the derivative as a function of three coefficients;

$$\frac{\partial \Psi_{mn}}{\partial T} = \left(\frac{-2Tc_{mn} - b_{mn}}{T} - \frac{-T^2 c_{mn} - Tb_{mn} - a_{mn}}{T^2}\right) e^{\frac{-T^2 c_{mn} - Tb_{mn} - a_{mn}}{T}}$$

Only the first, *a* coefficient, is used in the original UNIFAC model as well as the UNIFAC-LLE model, so the expression simplifies to:

$$\frac{\partial \Psi_{mn}}{\partial T} = \frac{a_{mn} e^{-\frac{a_{mn}}{T}}}{T^2}$$

For the Lyngby model, the first temperature derivative is:

$$\frac{\partial \Psi_{mk}}{\partial T} = \left(\frac{-a_2 - a_3 \ln\left(\frac{T_0}{T}\right)}{T} - \frac{-a_1 - a_2 (T - T_0) - a_3 \left(T \ln\left(\frac{T_0}{T}\right) + T - T_0\right)}{T^2}\right) e^{\frac{-a_1 - a_2 (T - T_0) - a_3 \left(T \ln\left(\frac{T_0}{T}\right) + T - T_0\right)}{T}}$$

with $T_0 = 298.15$ K and the *a* coefficients are specific to each pair of main groups, and they are asymmetric, so $a_{0,mk} \neq a_{0,km}$.

Returns

dpsis_dT [list[list[float]]] First temperature derivative of`psi` terms, size subgroups x subgroups [-]

static from_subgroups(*T*, *xs*, *chemgroups*, *subgroups=None*, *interaction_data=None*, *version=0*)

Method to construct a UNIFAC object from a dictionary of interaction parameters parameters and a list of dictionaries of UNIFAC keys. As the actual implementation is matrix based not dictionary based, this method can be quite convenient.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

chemgroups [list[dict]] List of dictionaries of subgroup IDs and their counts for all species in the mixture, [-]

subgroups [dict[int: UNIFAC_subgroup], optional] UNIFAC subgroup data; available dictionaries in this module include UFSG (original), DOUFSG (Dortmund), or NISTUFSG. The default depends on the given *version*, [-]

interaction_data [dict[int: dict[int: tuple(a_mn, b_mn, c_mn)]], optional] UNIFAC interaction parameter data; available dictionaries in this module include UFIP (original), DOUFIP2006 (Dortmund parameters published in 2006), DOUFIP2016 (Dortmund parameters published in 2016), and NISTUFIP. The default depends on the given *version*, [-]

version [int, optional] Which version of the model to use. Defaults to 0, [-]

- 0 - original UNIFAC, OR UNIFAC LLE
- 1 - Dortmund UNIFAC (adds T dept, 3/4 power)
- 2 - PSRK (original with T dept function)
- 3 - VTPR (drops combinatorial term, Dortmund UNIFAC otherwise)
- 4 - Lyngby/Larsen has different combinatorial, 2/3 power
- 5 - UNIFAC KT (2 params for psi, Lyngby/Larsen formulation; otherwise same as original)

Returns

UNIFAC [UNIFAC] Object for performing calculations with the UNIFAC activity coefficient model, [-]

Notes

Warning: For version 0, the interaction data and subgroups default to the original UNIFAC model (not LLE).

For version 1, the interaction data defaults to the Dortmund parameters publshed in 2016 (not 2006).

Examples

Mixture of [‘benzene’, ‘cyclohexane’, ‘acetone’, ‘ethanol’] according to the Dortmund UNIFAC model:

```
>>> from thermo.unifac import DOUFIP2006, DOUFSG
>>> T = 373.15
>>> xs = [0.2, 0.3, 0.1, 0.4]
>>> chemgroups = [{9: 6}, {78: 6}, {1: 1, 18: 1}, {1: 1, 2: 1, 14: 1}]
>>> GE = UNIFAC.from_subgroups(T=T, xs=xs, chemgroups=chemgroups, version=1,␣
↪interaction_data=DOUFIP2006, subgroups=DOUFSG)
>>> GE
UNIFAC(T=373.15, xs=[0.2, 0.3, 0.1, 0.4], rs=[2.2578, 4.2816, 2.3373, 2.
↪4951999999999996], qs=[2.5926, 5.181, 2.7308, 2.6616], Qs=[1.0608, 0.7081, 0.
↪4321, 0.8927, 1.67, 0.8635], vs=[[0, 0, 1, 1], [0, 0, 0, 1], [6, 0, 0, 0], [0,
↪ 0, 0, 1], [0, 0, 1, 0], [0, 6, 0, 0]], psi_abc=([[0.0, 0.0, 114.2, 2777.0,␣
↪433.6, -117.1], [0.0, 0.0, 114.2, 2777.0, 433.6, -117.1], [16.07, 16.07, 0.0,␣
↪3972.0, 146.2, 134.6], [1606.0, 1606.0, 3049.0, 0.0, -250.0, 3121.0], [199.0,␣
↪199.0, -57.53, 653.3, 0.0, 168.2], [170.9, 170.9, -2.619, 2601.0, 464.5, 0.
↪0]], [[0.0, 0.0, 0.0933, -4.674, 0.1473, 0.5481], [0.0, 0.0, 0.0933, -4.674,␣
↪0.1473, 0.5481], [-0.2998, -0.2998, 0.0, -13.16, -1.237, -1.231], [-4.746, -4.
↪746, -12.77, 0.0, 2.857, -13.69], [-0.8709, -0.8709, 1.212, -1.412, 0.0, -0.
↪8197], [-0.8062, -0.8062, 1.094, -1.25, 0.1542, 0.0]], [[0.0, 0.0, 0.0, 0.
↪001551, 0.0, -0.00098], [0.0, 0.0, 0.0, 0.001551, 0.0, -0.00098], [0.0, 0.0,␣
↪0.0, 0.01208, 0.004237, 0.001488], [0.0009181, 0.0009181, 0.01435, 0.0, -0.
↪006022, 0.01446], [0.0, 0.0, -0.003715, 0.000954, 0.0, 0.0], [0.001291, 0.
↪001291, -0.001557, -0.006309, 0.0, 0.0]]), version=1)
```

gammas()

Calculates the activity coefficients with the UNIFAC model.

$$\gamma_i = \exp\left(\ln \gamma_i^c + \ln \gamma_i^r\right)$$

For the VTPR variant, the combinatorial part is skipped:

$$\gamma_i = \exp(\ln \gamma_i^r)$$

Returns

gammas [list[float]] Activity coefficients, size number of components [-]

lnGammas_subgroups()

Calculate the $\ln \Gamma_k$ parameters for the phase; depends on the phases's composition and temperature.

$$\ln \Gamma_k = Q_k \left[1 - \ln \sum_m \Theta_m \Psi_{mk} - \sum_m \frac{\Theta_m \Psi_{km}}{\sum_n \Theta_n \Psi_{nm}}\right]$$

Returns

lnGammas_subgroups [list[float]] Gamma parameters for each subgroup, size number of subgroups, [-]

lnGammas_subgroups_pure()

Calculate the $\ln \Gamma_k$ pure component parameters for the phase; depends on the phases's temperature only.

$$\ln \Gamma_k = Q_k \left[1 - \ln \sum_m \Theta_m \Psi_{mk} - \sum_m \frac{\Theta_m \Psi_{km}}{\sum_n \Theta_n \Psi_{nm}}\right]$$

In this model, the Θ values come from the *UNIFAC.Thetas_pure* method, where each compound is assumed to be pure.

Returns

lnGammas_subgroups_pure [list[list[float]]] Gamma parameters for each subgroup, size number of subgroups by number of components and indexed in that order, [-]

`lngammas_c()`

Calculates the combinatorial part of the UNIFAC model. For the modified UNIFAC model, the equation is as follows; for the original UNIFAC and UNIFAC LLE, replace V_i' with V_i.

$$\ln \gamma_i^c = 1 - {V'}_i + \ln({V'}_i) - 5q_i \left(1 - \frac{V_i}{F_i} + \ln\left(\frac{V_i}{F_i}\right)\right)$$

For the Lyngby model:

$$\ln \gamma_i^c = \ln \left(V_i'\right) + 1 - V_i'$$

Returns

lngammas_c [list[float]] Combinatorial lngammas terms, size number of components [-]

`lngammas_r()`

Calculates the residual part of the UNIFAC model.

$$\ln \gamma_i^r = \sum_{k}^{gr} \nu_k^{(i)} \left[\ln \Gamma_k - \ln \Gamma_k^{(i)}\right]$$

where the second Gamma is the pure-component Gamma of group *k* in component *i*.

Returns

lngammas_r [list[float]] Residual lngammas terms, size number of components [-]

`property model_id`

A unique numerical identifier refering to the thermodynamic model being implemented. For internal use.

`psis()`

Calculate the Ψ term matrix for all groups interacting with all other groups.

The main model calculates it as a function of three coefficients;

$$\Psi_{mn} = \exp\left(\frac{-a_{mn} - b_{mn}T - c_{mn}T^2}{T}\right)$$

Only the first, *a* coefficient, is used in the original UNIFAC model as well as the UNIFAC-LLE model, so the expression simplifies to:

$$\Psi_{mn} = \exp\left(\frac{-a_{mn}}{T}\right)$$

For the Lyngby model, the temperature dependence is modified slightly, as follows:

$$\Psi_{mk} = e^{\frac{-a_1 - a_2(T - T_0) - a_3\left(T\ln\left(\frac{T_0}{T}\right) + T - T_0\right)}{T}}$$

with $T_0 = 298.15$ K and the *a* coefficients are specific to each pair of main groups, and they are asymmetric, so $a_{0,mk} \neq a_{0,km}$.

Returns

psis [list[list[float]]] *psi* terms, size subgroups x subgroups [-]

`to_T_xs(T, xs)`

Method to construct a new `UNIFAC` instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions of each component, [-]

Returns

obj [UNIFAC] New `UNIFAC` object at the specified conditions [-]

Notes

If the new temperature is the same temperature as the existing temperature, if the *psi* terms or their derivatives have been calculated, they will be set to the new object as well. If the mole fractions are the same, various subgroup terms are also kept.

7.29.2 Main Model (Functional)

`thermo.unifac.UNIFAC_gammas`(*T*, *xs*, *chemgroups*, *cached=None*, *subgroup_data=None*, *interaction_data=None*, *modified=False*)

Calculates activity coefficients using the UNIFAC model (optionally modified), given a mixture's temperature, liquid mole fractions, and optionally the subgroup data and interaction parameter data of your choice. The default is to use the original UNIFAC model, with the latest parameters published by DDBST. The model supports modified forms (Dortmund, NIST) when the *modified* parameter is True.

Parameters

T [float] Temperature of the system, [K]

xs [list[float]] Mole fractions of all species in the system in the liquid phase, [-]

chemgroups [list[dict]] List of dictionaries of subgroup IDs and their counts for all species in the mixture, [-]

subgroup_data [dict[UNIFAC_subgroup]] UNIFAC subgroup data; available dictionaries in this module are UFSG (original), DOUFSG (Dortmund), or NISTUFSG ([4]).

interaction_data [dict[dict[tuple(a_mn, b_mn, c_mn)]]] UNIFAC interaction parameter data; available dictionaries in this module are UFIP (original), DOUFIP2006 (Dortmund parameters as published by 2006), DOUFIP2016 (Dortmund parameters as published by 2016), and NISTUFIP ([4]).

modified [bool] True if using the modified form and temperature dependence, otherwise False.

Returns

gammas [list[float]] Activity coefficients of all species in the mixture, [-]

Notes

The actual implementation of UNIFAC is formulated slightly different than the formulas above for computational efficiency. DDBST switched to using the more efficient forms in their publication, but the numerical results are identical.

The model is as follows:

$$\ln \gamma_i = \ln \gamma_i^c + \ln \gamma_i^r$$

Combinatorial component

$$\ln \gamma_i^c = \ln \frac{\phi_i}{x_i} + \frac{z}{2} q_i \ln\frac{\theta_i}{\phi_i} + L_i - \frac{\phi_i}{x_i}\sum_{j=1}^{n} x_j L_j$$

$$\theta_i = \frac{x_i q_i}{\sum_{j=1}^{n} x_j q_j}$$

$$\phi_i = \frac{x_i r_i}{\sum_{j=1}^{n} x_j r_j}$$

$$L_i = 5(r_i - q_i) - (r_i - 1)$$

Residual component

$$\ln \gamma_i^r = \sum_{k}^{n} \nu_k^{(i)} \left[\ln \Gamma_k - \ln \Gamma_k^{(i)} \right]$$

$$\ln \Gamma_k = Q_k \left[1 - \ln \sum_m \Theta_m \Psi_{mk} - \sum_m \frac{\Theta_m \Psi_{km}}{\sum_n \Theta_n \Psi_{nm}}\right]$$

$$\Theta_m = \frac{Q_m X_m}{\sum_n Q_n X_n}$$

$$X_m = \frac{\sum_j \nu^j_m x_j}{\sum_j \sum_n \nu_n^j x_j}$$

R and Q

$$r_i = \sum_{k=1}^{n} \nu_k R_k$$

$$q_i = \sum_{k=1}^{n} \nu_k Q_k$$

The newer forms of UNIFAC (Dortmund, NIST) calculate the combinatorial part slightly differently:

$$\ln \gamma_i^c = 1 - {V'}_i + \ln({V'}_i) - 5q_i \left(1 - \frac{V_i}{F_i} + \ln\left(\frac{V_i}{F_i}\right)\right)$$

$$V'_i = \frac{r_i^{3/4}}{\sum_j r_j^{3/4} x_j}$$

$$V_i = \frac{r_i}{\sum_j r_j x_j}$$

$$F_i = \frac{q_i}{\sum_j q_j x_j}$$

Although this form looks substantially different than the original, it infact reverts to the original form if only V'_i is replaced by V_i. This is more clear when looking at the full rearranged form as in [3].

In some publications such as [5], the nomenclature is such that θ_i and ϕ do not contain the top x_i, making $\theta_i = F_i$ and $\phi_i = V_i$. [5] is also notable for having supporting information containing very nice sets of analytical derivatives.

UNIFAC LLE uses the original formulation of UNIFAC, and otherwise only different interaction parameters.

References

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

Examples

```
>>> UNIFAC_gammas(T=333.15, xs=[0.5, 0.5], chemgroups=[{1:2, 2:4}, {1:1, 2:1, 18:1}
↪])
[1.427602583562, 1.364654501010]
```

```
>>> from thermo.unifac import DOUFIP2006
>>> UNIFAC_gammas(373.15, [0.2, 0.3, 0.2, 0.2],
... [{9:6}, {78:6}, {1:1, 18:1}, {1:1, 2:1, 14:1}],
... subgroup_data=DOUFSG, interaction_data=DOUFIP2006, modified=True)
[1.1864311137, 1.44028013391, 1.20447983349, 1.972070609029]
```

`thermo.unifac.UNIFAC_psi`(*T*, *subgroup1*, *subgroup2*, *subgroup_data*, *interaction_data*, *modified=False*)

Calculates the interaction parameter psi(m, n) for two UNIFAC subgroups, given the system temperature, the UNIFAC subgroups considered for the variant of UNIFAC used, the interaction parameters for the variant of UNIFAC used, and whether or not the temperature dependence is modified from the original form, as shown below.

Original temperature dependence:

$$\Psi_{mn} = \exp\left(\frac{-a_{mn}}{T}\right)$$

Modified temperature dependence:

$$\Psi_{mn} = \exp\left(\frac{-a_{mn} - b_{mn}T - c_{mn}T^2}{T}\right)$$

Parameters

T [float] Temperature of the system, [K]

subgroup1 [int] First UNIFAC subgroup for identifier, [-]

subgroup2 [int] Second UNIFAC subgroup for identifier, [-]

subgroup_data [dict[UNIFAC_subgroup]] Normally provided as inputs to *UNIFAC*.

interaction_data [dict[dict[tuple(a_mn, b_mn, c_mn)]]] Normally provided as inputs to *UNIFAC*.

modified [bool] True if the modified temperature dependence is used by the interaction parameters, otherwise False

Returns

psi [float] UNIFAC interaction parameter term, [-]

Notes

UNIFAC interaction parameters are asymmetric. No warning is raised if an interaction parameter is missing.

References

[1], [2]

Examples

```
>>> from thermo.unifac import UFSG, UFIP, DOUFSG, DOUFIP2006
```

```
>>> UNIFAC_psi(307, 18, 1, UFSG, UFIP)
0.9165248264184787
```

```
>>> UNIFAC_psi(373.15, 9, 78, DOUFSG, DOUFIP2006, modified=True)
1.3703140538273264
```

7.29.3 Misc Functions

thermo.unifac.**UNIFAC_RQ**(*groups*, *subgroup_data=None*)

Calculates UNIFAC parameters R and Q for a chemical, given a dictionary of its groups, as shown in [1]. Most UNIFAC methods use the same subgroup values; however, a dictionary of *UNIFAC_subgroup* instances may be specified as an optional second parameter.

$$r_i = \sum_{k=1}^{n} \nu_k R_k$$

$$q_i = \sum_{k=1}^{n} \nu_k Q_k$$

Parameters

groups [dict[count]] Dictionary of numeric subgroup IDs : their counts

subgroup_data [None or dict[UNIFAC_subgroup]] Optional replacement for standard subgroups; leave as None to use the original UNIFAC subgroup r and q values.

Returns

R [float] R UNIFAC parameter (normalized Van der Waals Volume) [-]

Q [float] Q UNIFAC parameter (normalized Van der Waals Area) [-]

Notes

These parameters have some predictive value for other chemical properties.

References

[1]

Examples

Hexane

```
>>> UNIFAC_RQ({1:2, 2:4})
(4.4998000000000005, 3.856)
```

thermo.unifac.**Van_der_Waals_volume**(*R*)

Calculates a species Van der Waals molar volume with the UNIFAC method, given a species's R parameter.

$$V_{wk} = 15.17R_k$$

Parameters

R [float] R UNIFAC parameter (normalized Van der Waals Volume) [-]

Returns

V_vdw [float] Unnormalized Van der Waals volume, [m^3/mol]

Notes

The volume was originally given in cm^3/mol, but is converted to SI here.

References

[1]

Examples

```
>>> Van_der_Waals_volume(4.4998)
6.826196599999999e-05
```

thermo.unifac.**Van_der_Waals_area**(*Q*)

Calculates a species Van der Waals molar surface area with the UNIFAC method, given a species's Q parameter.

$$A_{wk} = 2.5 \times 10^9 Q_k$$

Parameters

Q [float] Q UNIFAC parameter (normalized Van der Waals Area) [-]

Returns

A_vdw [float] Unnormalized Van der Waals surface area, [m^2/mol]

Notes

The volume was originally given in cm^2/mol, but is converted to SI here.

References

[1]

Examples

```
>>> Van_der_Waals_area(3.856)
964000.0
```

thermo.unifac.**chemgroups_to_matrix**(*chemgroups*)

Index by [group index][compound index]

```
>>> chemgroups_to_matrix([{9: 6}, {2: 6}, {1: 1, 18: 1}, {1: 1, 2: 1, 14: 1}])
[[0, 0, 1, 1], [0, 6, 0, 1], [6, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]
```

thermo.unifac.**load_group_assignments_DDBST**()

Data is stored in the format InChI key bool bool bool subgroup count … subgroup count subgroup count… where the bools refer to whether or not the original UNIFAC, modified UNIFAC, and PSRK group assignments were completed correctly. The subgroups and their count have an indefinite length.

7.29.4 Data for Original UNIFAC

thermo.unifac.**UFSG = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>, 7: <CH2=C>, 8: <CH=C>, 9: <ACH>, 10: <AC>, 11: <ACCH3>, 12: <ACCH2>, 13: <ACCH>, 14: <OH>, 15: <CH3OH>, 16: <H2O>, 17: <ACOH>, 18: <CH3CO>, 19: <CH2CO>, 20: <CHO>, 21: <CH3COO>, 22: <CH2COO>, 23: <HCOO>, 24: <CH3O>, 25: <CH2O>, 26: <CHO>, 27: <THF>, 28: <CH3NH2>, 29: <CH2NH2>, 30: <CHNH2>, 31: <CH3NH>, 32: <CH2NH>, 33: <CHNH>, 34: <CH3N>, 35: <CH2N>, 36: <ACNH2>, 37: <C5H5N>, 38: <C5H4N>, 39: <C5H3N>, 40: <CH3CN>, 41: <CH2CN>, 42: <COOH>, 43: <HCOOH>, 44: <CH2CL>, 45: <CHCL>, 46: <CCL>, 47: <CH2CL2>, 48: <CHCL2>, 49: <CCL2>, 50: <CHCL3>, 51: <CCL3>, 52: <CCL4>, 53: <ACCL>, 54: <CH3NO2>, 55: <CH2NO2>, 56: <CHNO2>, 57: <ACNO2>, 58: <CS2>, 59: <CH3SH>, 60: <CH2SH>, 61: <FURFURAL>, 62: <DOH>, 63: <I>, 64:
, 65: <CH=-C>, 66: <C=-C>, 67: <DMSO>, 68: <ACRY>, 69: <CL-(C=C)>, 70: <C=C>, 71: <ACF>, 72: <DMF>, 73: <HCON(CH2)2>, 74: <CF3>, 75: <CF2>, 76: <CF>, 77: <COO>, 78: <SIH3>, 79: <SIH2>, 80: <SIH>, 81: <SI>, 82: <SIH2O>, 83: <SIHO>, 84: <SIO>, 85: <NMP>, 86: <CCL3F>, 87: <CCL2F>, 88: <HCCL2F>, 89: <HCCLF>, 90: <CCLF2>, 91: <HCCLF2>, 92: <CCLF3>, 93: <CCL2F2>, 94: <AMH2>, 95: <AMHCH3>, 96: <AMHCH2>, 97: <AM(CH3)2>, 98: <AMCH3CH2>, 99: <AM(CH2)2>, 100: <C2H5O2>, 101: <C2H4O2>, 102: <CH3S>, 103: <CH2S>, 104: <CHS>, 105: <MORPH>, 106: <C4H4S>, 107: <C4H3S>, 108: <C4H2S>, 109: <NCO>, 118: <(CH2)2SU>, 119: <CH2CHSU>, 178: <IMIDAZOL>, 179: <BTI>}**

```
thermo.unifac.UFMG = {1: ('CH2', [1, 2, 3, 4]), 2: ('C=C', [5, 6, 7, 8, 70]), 3: ('ACH',
[9, 10]), 4: ('ACCH2', [11, 12, 13]), 5: ('OH', [14]), 6: ('CH3OH', [15]), 7: ('H2O',
[16]), 8: ('ACOH', [17]), 9: ('CH2CO', [18, 19]), 10: ('CHO', [20]), 11: ('CCOO', [21,
22]), 12: ('HCOO', [23]), 13: ('CH2O', [24, 25, 26, 27]), 14: ('CNH2', [28, 29, 30]), 15:
('CNH', [31, 32, 33]), 16: ('(C)3N', [34, 35]), 17: ('ACNH2', [36]), 18: ('PYRIDINE',
[37, 38, 39]), 19: ('CCN', [40, 41]), 20: ('COOH', [42, 43]), 21: ('CCL', [44, 45, 46]),
22: ('CCL2', [47, 48, 49]), 23: ('CCL3', [50, 51]), 24: ('CCL4', [52]), 25: ('ACCL',
[53]), 26: ('CNO2', [54, 55, 56]), 27: ('ACNO2', [57]), 28: ('CS2', [58]), 29: ('CH3SH',
[59, 60]), 30: ('FURFURAL', [61]), 31: ('DOH', [62]), 32: ('I', [63]), 33: ('BR', [64]),
34: ('C=-C', [65, 66]), 35: ('DMSO', [67]), 36: ('ACRY', [68]), 37: ('CLCC', [69]), 38:
('ACF', [71]), 39: ('DMF', [72, 73]), 40: ('CF2', [74, 75, 76]), 41: ('COO', [77]), 42:
('SIH2', [78, 79, 80, 81]), 43: ('SIO', [82, 83, 84]), 44: ('NMP', [85]), 45: ('CCLF',
[86, 87, 88, 89, 90, 91, 92, 93]), 46: ('CON(AM)', [94, 95, 96, 97, 98, 99]), 47:
('OCCOH', [100, 101]), 48: ('CH2S', [102, 103, 104]), 49: ('MORPH', [105]), 50:
('THIOPHEN', [106, 107, 108]), 51: ('NCO', [109]), 55: ('SULFONES', [118, 119]), 84:
('IMIDAZOL', [178]), 85: ('BTI', [179])}
```

```
thermo.unifac.UFIP
```

Interaction parameters for the original unifac model.

Type dict[int: dict[int: float]]

7.29.5 Data for Dortmund UNIFAC

```
thermo.unifac.DOUFSG = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>, 7:
<CH2=C>, 8: <CH=C>, 9: <ACH>, 10: <AC>, 11: <ACCH3>, 12: <ACCH2>, 13: <ACCH>, 14:
<OH(P)>, 15: <CH3OH>, 16: <H2O>, 17: <ACOH>, 18: <CH3CO>, 19: <CH2CO>, 20: <CHO>, 21:
<CH3COO>, 22: <CH2COO>, 23: <HCOO>, 24: <CH3O>, 25: <CH2O>, 26: <CHO>, 27: <THF>, 28:
<CH3NH2>, 29: <CH2NH2>, 30: <CHNH2>, 31: <CH3NH>, 32: <CH2NH>, 33: <CHNH>, 34: <CH3N>,
35: <CH2N>, 36: <ACNH2>, 37: <AC2H2N>, 38: <AC2HN>, 39: <AC2N>, 40: <CH3CN>, 41: <CH2CN>,
42: <COOH>, 43: <HCOOH>, 44: <CH2CL>, 45: <CHCL>, 46: <CCL>, 47: <CH2CL2>, 48: <CHCL2>,
49: <CCL2>, 50: <CHCL3>, 51: <CCL3>, 52: <CCL4>, 53: <ACCL>, 54: <CH3NO2>, 55: <CH2NO2>,
56: <CHNO2>, 57: <ACNO2>, 58: <CS2>, 59: <CH3SH>, 60: <CH2SH>, 61: <FURFURAL>, 62: <DOH>,
63: <I>, 64: <BR>, 65: <CH=-C>, 66: <C=-C>, 67: <DMSO>, 68: <ACRY>, 69: <CL-(C=C)>, 70:
<C=C>, 71: <ACF>, 72: <DMF>, 73: <HCON(CH2)2>, 74: <CF3>, 75: <CF2>, 76: <CF>, 77: <COO>,
78: <CY-CH2>, 79: <CY-CH>, 80: <CY-C>, 81: <OH(S)>, 82: <OH(T)>, 83: <CY-CH2O>, 84:
<TRIOXAN>, 85: <CNH2>, 86: <NMP>, 87: <NEP>, 88: <NIPP>, 89: <NTBP>, 91: <CONH2>, 92:
<CONHCH3>, 100: <CONHCH2>, 101: <AM(CH3)2>, 102: <AMCH3CH2>, 103: <AM(CH2)2>, 104:
<AC2H2S>, 105: <AC2HS>, 106: <AC2S>, 107: <H2COCH>, 108: <COCH>, 109: <HCOCH>, 110:
<(CH2)2SU>, 111: <CH2SUCH>, 112: <(CH3)2CB>, 113: <(CH2)2CB>, 114: <CH2CH3CB>, 119:
<H2COCH2>, 122: <CH3S>, 123: <CH2S>, 124: <CHS>, 153: <H2COC>, 178: <C3H2N2+>, 179:
<BTI->, 184: <C3H3N2+>, 189: <C4H8N+>, 195: <BF4->, 196: <C5H5N+>, 197: <OTF->, 201:
<-S-S->}
```

thermo.unifac.DOUFMG = {1: ('CH2', [1, 2, 3, 4]), 2: ('C=C', [5, 6, 7, 8, 70]), 3: ('ACH', [9, 10]), 4: ('ACCH2', [11, 12, 13]), 5: ('OH', [14, 81, 82]), 6: ('CH3OH', [15]), 7: ('H2O', [16]), 8: ('ACOH', [17]), 9: ('CH2CO', [18, 19]), 10: ('CHO', [20]), 11: ('CCOO', [21, 22]), 12: ('HCOO', [23]), 13: ('CH2O', [24, 25, 26]), 14: ('CH2NH2', [28, 29, 30, 85]), 15: ('CH2NH', [31, 32, 33]), 16: ('(C)3N', [34, 35]), 17: ('ACNH2', [36]), 18: ('PYRIDINE', [37, 38, 39]), 19: ('CH2CN', [40, 41]), 20: ('COOH', [42]), 21: ('CCL', [44, 45, 46]), 22: ('CCL2', [47, 48, 49]), 23: ('CCL3', [51]), 24: ('CCL4', [52]), 25: ('ACCL', [53]), 26: ('CNO2', [54, 55, 56]), 27: ('ACNO2', [57]), 28: ('CS2', [58]), 29: ('CH3SH', [59, 60]), 30: ('FURFURAL', [61]), 31: ('DOH', [62]), 32: ('I', [63]), 33: ('BR', [64]), 34: ('C=-C', [65, 66]), 35: ('DMSO', [67]), 36: ('ACRY', [68]), 37: ('CLCC', [69]), 38: ('ACF', [71]), 39: ('DMF', [72, 73]), 40: ('CF2', [74, 75, 76]), 41: ('COO', [77]), 42: ('CY-CH2', [78, 79, 80]), 43: ('CY-CH2O', [27, 83, 84]), 44: ('HCOOH', [43]), 45: ('CHCL3', [50]), 46: ('CY-CONC', [86, 87, 88, 89]), 47: ('CONR', [91, 92, 100]), 48: ('CONR2', [101, 102, 103]), 49: ('HCONR', [93, 94]), 52: ('ACS', [104, 105, 106]), 53: ('EPOXIDES', [107, 108, 109, 119, 153]), 55: ('CARBONAT', [112, 113, 114]), 56: ('SULFONE', [110, 111]), 61: ('SULFIDES', [122, 123, 124]), 84: ('IMIDAZOL', [178, 184]), 85: ('BTI', [179]), 87: ('PYRROL', [189]), 89: ('BF4', [195]), 90: ('PYRIDIN', [196]), 91: ('OTF', [197]), 93: ('DISULFIDES', [201])}

thermo.unifac.DOUFIP2016

Interaction parameters for the Dornmund unifac model.

Type dict[int: dict[int: tuple(float, 3)]]

7.29.6 Data for NIST UNIFAC (2015)

```
thermo.unifac.NISTUFSG = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>,
7: <CH2=C>, 8: <CH=C>, 9: <ACH>, 10: <AC>, 11: <ACCH3>, 12: <ACCH2>, 13: <ACCH>, 14: <OH
prim>, 15: <CH3OH>, 16: <H2O>, 17: <ACOH>, 18: <CH3CO>, 19: <CH2CO>, 20: <CHO>, 21:
<CH3COO>, 22: <CH2COO>, 23: <HCOO>, 24: <CH3O>, 25: <CH2O>, 26: <CHO>, 27: <CH2-O-CH2>,
28: <CH3NH2>, 29: <CH2NH2>, 30: <CHNH2>, 31: <CH3NH>, 32: <CH2NH>, 33: <CHNH>, 34:
<CH3N>, 35: <CH2N>, 36: <ACNH2>, 37: <AC2H2N>, 38: <AC2HN>, 39: <AC2N>, 40: <CH3CN>, 41:
<CH2CN>, 42: <COOH>, 43: <HCOOH>, 44: <CH2Cl>, 45: <CHCl>, 46: <CCl>, 47: <CH2Cl2>, 48:
<CHCl2>, 49: <CCl2>, 50: <CHCl3>, 51: <CCl3>, 52: <CCl4>, 53: <ACCl>, 54: <CH3NO2>, 55:
<CH2NO2>, 56: <CHNO2>, 57: <ACNO2>, 58: <CS2>, 59: <CH3SH>, 60: <CH2SH>, 61: <Furfural>,
62: <CH2(OH)-CH2(OH)>, 63: <I>, 64: <Br>, 65: <CH#C>, 66: <C#C>, 67: <DMSO>, 68:
<Acrylonitrile>, 69: <Cl-(C=C)>, 70: <C=C>, 71: <ACF>, 72: <DMF>, 73: <HCON(CH2)2>, 74:
<CF3>, 75: <CF2>, 76: <CF>, 77: <COO>, 78: <c-CH2>, 79: <c-CH>, 80: <c-C>, 81: <OH sec>,
82: <OH tert>, 83: <CH2-O-[CH2-O]1/2>, 84: <[O-CH2]1/2-O-[CH2-O]1/2>, 85: <CNH2>, 86:
<c-CON-CH3>, 87: <c-CON-CH2>, 88: <c-CON-CH>, 89: <c-CON-C>, 92: <CONHCH3>, 93:
<HCONHCH3>, 94: <HCONHCH2>, 100: <CONHCH2>, 101: <CON(CH3)2>, 102: <CON(CH3)CH2>, 103:
<CON(CH2)2>, 104: <AC2H2S>, 105: <AC2HS>, 106: <AC2S>, 107: <H2COCH>, 109: <HCOCH>, 110:
<CH2SuCH2>, 111: <CH2SuCH >, 112: <(CH3O)2CO>, 113: <(CH2O)2CO>, 114: <(CH3O)COOCH2>,
116: <ACCN>, 117: <CH3NCO>, 118: <CH2NCO>, 119: <CHNCO>, 120: <ACNCO>, 121: <COOCO>, 122:
<ACSO2>, 123: <ACCHO>, 124: <ACCOOH>, 125: <c-CO-NH>, 126: <c-CO-O>, 127: <AC-O-CO-CH3 >,
128: <AC-O-CO-CH2>, 129: <AC-O-CO-CH>, 130: <AC-O-CO-C>, 131: <-O-CH2-CH2-OH>, 132:
<-O-CH-CH2-OH>, 133: <-O-CH2-CH-OH>, 134: <CH3-S->, 135: <-CH2-S->, 136: <>CH-S->, 137:
<->C-S->, 138: <CH3O-(O)>, 139: <CH2O-(O)>, 140: <CHO-(O)>, 141: <CO-(O)>, 142:
<ACO-(O)>, 143: <CFH>, 144: <CFCl>, 145: <CFCl2>, 146: <CF2H>, 147: <CF2ClH>, 148:
<CF2Cl2>, 149: <CF3H>, 150: <CF3Cl>, 151: <CF4>, 152: <C(O)2>, 153: <ACN(CH3)2>, 154:
<ACN(CH3)CH2>, 155: <ACN(CH2)2>, 156: <ACNHCH3>, 157: <ACNHCH2>, 158: <ACNHCH>, 159:
<AC2H2O>, 160: <AC2HO>, 161: <AC2O>, 162: <c-CH-NH>, 163: <c-C-NH>, 164: <c-CH-NCH3>,
165: <c-CH-NCH2>, 166: <c-CH-NCH>, 170: <SiH3->, 171: <-SiH2->, 172: <>SiH->, 173:
<>Si<>, 174: <-SiH2-O->, 175: <>SiH-O->, 176: <->Si-O->, 177: <C=NOH>, 178: <ACCO>, 179:
<C2Cl4>, 180: <c-CHH2>, 186: <CH(O)2>, 187: <ACS>, 188: <c-CH2-NH>, 189: <c-CH2-NCH3>,
190: <c-CH2-NCH2>, 191: <c-CH2-NCH>, 192: <CHSH>, 193: <CSH>, 194: <ACSH>, 195: <ACC>,
196: <AC2H2NH>, 197: <AC2HNH>, 198: <AC2NH>, 199: <(ACO)COOCH2>, 200: <(ACO)CO(OAC)>,
201: <c-CH=CH>, 202: <c-CH=C>, 203: <c-C=C>, 204: <Glycerol>, 205: <-CH(OH)-CH2(OH)>,
206: <-CH(OH)-CH(OH)->, 207: <>C(OH)-CH2(OH)>, 208: <>C(OH)-CH(OH)->, 209:
<>C(OH)-C(OH)<>, 301: <CHCO>, 302: <CCO>, 303: <CHCN>, 304: <CCN>, 305: <CNO2>, 306:
<ACNH>, 307: <ACN>, 308: <HCHO>, 309: <CH=NOH>}
```

thermo.unifac.NISTUFMG = {1: ('CH2', [1, 2, 3, 4], 'Alkyl chains'), 2: ('C=C', [5, 6, 7, 8, 9], 'Double bonded alkyl chains'), 3: ('ACH', [15, 16, 17], 'Aromatic carbon'), 4: ('ACCH2', [18, 19, 20, 21], 'Aromatic carbon plus alkyl chain'), 5: ('OH', [34, 204, 205], 'Alcohols'), 6: ('CH3OH', [35], 'Methanol'), 7: ('H2O', [36], 'Water'), 8: ('ACOH', [37], 'Phenolic -OH groups '), 9: ('CH2CO', [42, 43, 44, 45], 'Ketones'), 10: ('CHO', [48], 'Aldehydes'), 11: ('CCOO', [51, 52, 53, 54], 'Esters'), 12: ('HCOO', [55], 'Formates'), 13: ('CH2O', [59, 60, 61, 62, 63], 'Ethers'), 14: ('CNH2', [66, 67, 68, 69], 'Amines with 1-alkyl group'), 15: ('(C)2NH', [71, 72, 73], 'Amines with 2-alkyl groups'), 16: ('(C)3N', [74, 75], 'Amines with 3-alkyl groups'), 17: ('ACNH2', [79, 80, 81], 'Anilines'), 18: ('PYRIDINE', [76, 77, 78], 'Pyridines'), 19: ('CCN', [85, 86, 87, 88], 'Nitriles'), 20: ('COOH', [94, 95], 'Acids'), 21: ('CCl', [99, 100, 101], 'Chlorocarbons'), 22: ('CCl2', [102, 103, 104], 'Dichlorocarbons'), 23: ('CCl3', [105, 106], 'Trichlorocarbons'), 24: ('CCl4', [107], 'Tetrachlorocarbons'), 25: ('ACCl', [109], 'Chloroaromatics'), 26: ('CNO2', [132, 133, 134, 135], 'Nitro alkanes'), 27: ('ACNO2', [136], 'Nitroaromatics'), 28: ('CS2', [146], 'Carbon disulfide'), 29: ('CH3SH', [138, 139, 140, 141], 'Mercaptans'), 30: ('FURFURAL', [50], 'Furfural'), 31: ('DOH', [38], 'Ethylene Glycol'), 32: ('I', [128], 'Iodides'), 33: ('BR', [130], 'Bromides'), 34: ('CC', [13, 14], 'Triplebonded alkyl chains'), 35: ('DMSO', [153], 'Dimethylsulfoxide'), 36: ('ACRY', [90], 'Acrylic'), 37: ('ClC=C', [108], 'Chlorine attached to double bonded alkyl chain'), 38: ('ACF', [118], 'Fluoroaromatics'), 39: ('DMF', [161, 162, 163, 164, 165], 'Amides'), 40: ('CF2', [111, 112, 113, 114, 115, 116, 117], 'Fluorines'), 41: ('COO', [58], 'Esters'), 42: ('SiH2', [197, 198, 199, 200], 'Silanes'), 43: ('SiO', [201, 202, 203], 'Siloxanes'), 44: ('NMP', [195], 'N-Methyl-2-pyrrolidone'), 45: ('CClF', [120, 121, 122, 123, 124, 125, 126, 127], 'Chloro-Fluorides'), 46: ('CONCH2', [166, 167, 168, 169], 'Amides'), 47: ('OCCOH', [39, 40, 41], 'Oxygenated Alcohols'), 48: ('CH2S', [142, 143, 144, 145], 'Sulfides'), 49: ('MORPHOLIN', [196], 'Morpholine'), 50: ('THIOPHENE', [147, 148, 149], 'Thiophene'), 51: ('CH2(cy)', [27, 28, 29], 'Cyclic hydrocarbon chains'), 52: ('C=C(cy)', [30, 31, 32], 'Cyclic unsaturated hydrocarbon chains')}

thermo.unifac.**NISTUFIP**

Interaction parameters for the NIST (2015) unifac model.

Type dict[int: dict[int: tuple(float, 3)]]

7.29.7 Data for NIST KT UNIFAC (2011)

thermo.unifac.**NISTKTUFSG** = {1: <CH3->, 2: <-CH2->, 3: <-CH<>, 4: <>C<>, 5: <CH2=CH->, 6: <-CH=CH->, 7: <CH2=C<>, 8: <-CH=C<>, 9: <>C=C<>, 13: <CHC->, 14: <-CC->, 15: <-ACH->, 16: <>AC- (link)>, 17: <>AC- (cond)>, 18: <>AC-CH3>, 19: <>AC-CH2->, 20: <>AC-CH<>, 21: <>AC-C<->, 27: <-CH2- (cy)>, 28: <>CH- (cy)>, 29: <>C< (cy)>, 30: <-CH=CH- (cy)>, 31: <CH2=C< (cy)>, 32: <-CH=C< (cy)>, 34: <-OH(primary)>, 35: <CH3OH>, 36: <H2O>, 37: <>AC-OH>, 38: <(CH2OH)2>, 39: <-O-CH2-CH2-OH>, 40: <-O-CH-CH2-OH>, 41: <-O-CH2-CH-OH>, 42: <CH3-CO->, 43: <-CH2-CO->, 44: <>CH-CO->, 45: <->C-CO->, 48: <-CHO>, 50: <C5H4O2>, 51: <CH3-COO->, 52: <-CH2-COO->, 53: <>CH-COO->, 54: <->C-COO->, 55: <HCOO->, 58: <-COO->, 59: <CH3-O->, 60: <-CH2-O->, 61: <>CH-O->, 62: <->CO->, 63: <-CH2-O- (cy)>, 66: <CH3-NH2>, 67: <-CH2-NH2>, 68: <>CH-NH2>, 69: <->C-NH2>, 71: <CH3-NH->, 72: <-CH2-NH->, 73: <>CH-NH->, 74: <CH3-N<>, 75: <-CH2-N<>, 76: <C5H5N>, 77: <C5H4N->, 78: <C5H3N<>, 79: <>AC-NH2>, 80: <>AC-NH->, 81: <>AC-N<>, 85: <CH3-CN>, 86: <-CH2-CN>, 87: <>CH-CN>, 88: <->C-CN>, 90: <CH2=CH-CN>, 94: <-COOH>, 95: <HCOOH>, 99: <-CH2-Cl>, 100: <>CH-Cl>, 101: <->CCl>, 102: <CH2Cl2>, 103: <-CHCl2>, 104: <>CCl2>, 105: <CHCl3>, 106: <-CCl3>, 107: <CCl4>, 108: <Cl(C=C)>, 109: <>AC-Cl>, 111: <CHF3>, 112: <-CF3>, 113: <-CHF2>, 114: <>CF2>, 115: <-CH2F>, 116: <>CH-F>, 117: <->CF>, 118: <>AC-F>, 120: <CCl3F>, 121: <-CCl2F>, 122: <HCCl2F>, 123: <-HCClF>, 124: <-CClF2>, 125: <HCClF2>, 126: <CClF3>, 127: <CCl2F2>, 128: <-I>, 130: <-Br>, 132: <CH3-NO2>, 133: <-CH2-NO2>, 134: <>CH-NO2>, 135: <->C-NO2>, 136: <>AC-NO2>, 138: <CH3-SH>, 139: <-CH2-SH>, 140: <>CH-SH>, 141: <->C-SH>, 142: <CH3-S->, 143: <-CH2-S->, 144: <>CH-S->, 145: <->C-S->, 146: <CS2>, 147: <THIOPHENE>, 148: <C4H3S->, 149: <C4H2S<>, 153: <DMSO>, 161: <DMF>, 162: <-CON(CH3)2>, 163: <-CON(CH2)(CH3)->, 164: <HCON(CH2)2<>, 165: <-CON(CH2)2<>, 166: <-CONH(CH3)>, 167: <HCONH(CH2)->, 168: <-CONH(CH2)->, 169: <-CONH2>, 195: <NMP>, 196: <MORPHOLIN>, 197: <SiH3->, 198: <-SiH2->, 199: <>SiH->, 200: <>Si<>, 201: <-SiH2-O->, 202: <>SiH-O->, 203: <->Si-O->, 204: <-OH(secondary)>, 205: <-OH(tertiary)>}

thermo.unifac.**NISTKTUFMG** = {1: ('C', [1, 2, 3, 4]), 2: ('C=C', [5, 6, 7, 8, 9]), 3: ('ACH', [15, 16, 17]), 4: ('ACCH2', [18, 19, 20, 21]), 5: ('OH', [34, 204, 205]), 6: ('CH2OH', [35]), 7: ('H2O', [36]), 8: ('ACOH', [37]), 9: ('CH2CO', [42, 43, 44, 45]), 10: ('CHO', [48]), 11: ('CCOO', [51, 52, 53, 54]), 12: ('HCOO', [55]), 13: ('CH2O', [59, 60, 61, 62]), 14: ('CNH2', [66, 67, 68, 69]), 15: ('(C)2NH', [71, 72, 73]), 16: ('(C)3N', [74, 75]), 17: ('ACNH2', [79, 80, 81]), 18: ('Pyridine', [76, 77, 78]), 19: ('CCN', [85, 86, 87, 88]), 20: ('COOH', [94, 95]), 21: ('CCl', [99, 100, 101]), 22: ('CCl2', [102, 103, 104]), 23: ('CCl3', [105, 106]), 24: ('CCl4', [107]), 25: ('ACCl', [109]), 26: ('CNO2', [132, 133, 134, 135]), 27: ('ACNO2', [136]), 28: ('CS2', [146]), 29: ('CH3SH', [138, 139, 140, 141]), 30: ('Furfural', [50]), 31: ('DOH', [38]), 32: ('I', [128]), 33: ('Br', [130]), 34: ('C=-C', [13, 14]), 35: ('DMSO', [153]), 36: ('ACRY', [90]), 37: ('Cl(C=C)', [108]), 38: ('ACF', [118]), 39: ('DMF', [161, 162, 163, 164, 165]), 40: ('CF2', [111, 112, 113, 114, 115, 116, 117]), 41: ('COO', [58]), 42: ('SiH2', [197, 198, 199, 200]), 43: ('SiO', [201, 202, 203]), 44: ('NMP', [195]), 45: ('CClF', [120, 121, 122, 123, 124, 125, 126, 127]), 46: ('CONCH2', [166, 167, 168, 169]), 47: ('OCCOH', [39, 40, 41]), 48: ('CH2S', [142, 143, 144, 145]), 49: ('Morpholin', [196]), 50: ('THIOPHENE', [147, 148, 149]), 51: ('CH2(cyc)', [27, 28, 29]), 52: ('C=C(cyc)', [30, 31, 32])}

Compared to storing the values in dict[(int1, int2)] = (values), the dict-in-dict structure is found emperically to take 111608 bytes vs. 79096 bytes, or 30% less memory.

thermo.unifac.**NISTKTUFIP**

Interaction parameters for the NIST KT UNIFAC (2011) model.

Type dict[int: dict[int: tuple(float, 3)]]

7.29.8 Data for UNIFAC LLE

thermo.unifac.**LLEUFSG** = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>, 7: <CH=C>, 8: <CH2=C>, 9: <ACH>, 10: <AC>, 11: <ACCH3>, 12: <ACCH2>, 13: <ACCH>, 14: <OH>, 15: <P1>, 16: <P2>, 17: <H2O>, 18: <ACOH>, 19: <CH3CO>, 20: <CH2CO>, 21: <CHO>, 22: <Furfural>, 23: <COOH>, 24: <HCOOH>, 25: <CH3COO>, 26: <CH2COO>, 27: <CH3O>, 28: <CH2O>, 29: <CHO>, 30: <FCH2O>, 31: <CH2CL>, 32: <CHCL>, 33: <CCL>, 34: <CH2CL2>, 35: <CHCL2>, 36: <CCL2>, 37: <CHCL3>, 38: <CCL3>, 39: <CCL4>, 40: <ACCL>, 41: <CH3CN>, 42: <CH2CN>, 43: <ACNH2>, 44: <CH3NO2>, 45: <CH2NO2>, 46: <CHNO2>, 47: <ACNO2>, 48: <DOH>, 49: <(HOCH2CH2)2O>, 50: <C5H5N>, 51: <C5H4N>, 52: <C5H3N>, 53: <CCl2=CHCl>, 54: <HCONHCH3>, 55: <DMF>, 56: <(CH2)4SO2>, 57: <DMSO>}

thermo.unifac.**LLEMG** = {1: ('CH2', [1, 2, 3, 4]), 2: ('C=C', [5, 6, 7, 8]), 3: ('ACH', [9, 10]), 4: ('ACCH2', [11, 12, 13]), 5: ('OH', [14]), 6: ('P1', [15]), 7: ('P2', [16]), 8: ('H2O', [17]), 9: ('ACOH', [18]), 10: ('CH2CO', [19, 20]), 11: ('CHO', [21]), 12: ('Furfural', [22]), 13: ('COOH', [23, 24]), 14: ('CCOO', [25, 26]), 15: ('CH2O', [27, 28, 29, 30]), 16: ('CCL', [31, 32, 33]), 17: ('CCL2', [34, 35, 36]), 18: ('CCL3', [37, 38]), 19: ('CCL4', [39]), 20: ('ACCL', [40]), 21: ('CCN', [41, 42]), 22: ('ACNH2', [43]), 23: ('CNO2', [44, 45, 46]), 24: ('ACNO2', [47]), 25: ('DOH', [48]), 26: ('DEOH', [49]), 27: ('PYRIDINE', [50, 51, 52]), 28: ('TCE', [53]), 29: ('MFA', [54]), 30: ('DMFA', [55]), 31: ('TMS', [56]), 32: ('DMSO', [57])}

Larsen, Bent L., Peter Rasmussen, and Aage Fredenslund. "A Modified UNIFAC Group-Contribution Model for Prediction of Phase Equilibria and Heats of Mixing." Industrial & Engineering Chemistry Research 26, no. 11 (November 1, 1987): 2274-86. https://doi.org/10.1021/ie00071a018.

thermo.unifac.**LLEUFIP**

Interaction parameters for the LLE unifac model.

Type dict[int: dict[int: float]]

7.29.9 Data for Lyngby UNIFAC

thermo.unifac.**LUFSG** = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>, 7: <CH2=C>, 8: <CH=C>, 9: <C=C>, 10: <ACH>, 11: <AC>, 12: <OH>, 13: <CH3OH>, 14: <H2O>, 15: <CH3CO>, 16: <CH2CO>, 17: <CHO>, 18: <CH3COO>, 19: <CH2COO>, 20: <CH3O>, 21: <CH2O>, 22: <CHO>, 23: <THF>, 24: <NH2>, 25: <CH3NH>, 26: <CH2NH>, 27: <CHNH>, 28: <CH3N>, 29: <CH2N>, 30: <ANH2>, 31: <C5H5N>, 32: <C5H4N>, 33: <C5H3N>, 34: <CH3CN>, 35: <CH2CN>, 36: <COOH>, 37: <CH2CL>, 38: <CHCL>, 39: <CCL>, 40: <CH2CL2>, 41: <CHCL2>, 42: <CCL2>, 43: <CHCL3>, 44: <CCL3>, 45: <CCL4>}

thermo.unifac.**LUFMG** = {1: ('CH2', [1, 2, 3, 4]), 2: ('C=C', [5, 6, 7, 8, 9]), 3: ('ACH', [10, 11]), 4: ('OH', [12]), 5: ('CH3OH', [13]), 6: ('H2O', [14]), 7: ('CH2CO', [15, 16]), 8: ('CHO', [17]), 9: ('CCOO', [18, 19]), 10: ('CH2O', [20, 21, 22, 23]), 11: ('NH2', [24]), 12: ('CNH2NG', [25, 26, 27]), 13: ('CH2N', [28, 29]), 14: ('ANH2', [30]), 15: ('PYRIDINE', [31, 32, 33]), 16: ('CCN', [34, 35]), 17: ('COOH', [36]), 18: ('CCL', [37, 38, 39]), 19: ('CCL2', [40, 41, 42]), 20: ('CCL3', [43, 44]), 21: ('CCL4', [45])}

thermo.unifac.**LUFIP**

Interaction parameters for the Lyngby UNIFAC model.

Type dict[int: dict[int: tuple(float, 3)]]

7.29.10 Data for PSRK UNIFAC

```
thermo.unifac.PSRKSG = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>, 7:
<CH2=C>, 8: <CH=C>, 9: <ACH>, 10: <AC>, 11: <ACCH3>, 12: <ACCH2>, 13: <ACCH>, 14: <OH>,
15: <CH3OH>, 16: <H2O>, 17: <ACOH>, 18: <CH3CO>, 19: <CH2CO>, 20: <CHO>, 21: <CH3COO>,
22: <CH2COO>, 23: <HCOO>, 24: <CH3O>, 25: <CH2O>, 26: <CHO>, 27: <THF>, 28: <CH3NH2>, 29:
<CH2NH2>, 30: <CHNH2>, 31: <CH3NH>, 32: <CH2NH>, 33: <CHNH>, 34: <CH3N>, 35: <CH2N>, 36:
<ACNH2>, 37: <C5H5N>, 38: <C5H4N>, 39: <C5H3N>, 40: <CH3CN>, 41: <CH2CN>, 42: <COOH>, 43:
<HCOOH>, 44: <CH2CL>, 45: <CHCL>, 46: <CCL>, 47: <CH2CL2>, 48: <CHCL2>, 49: <CCL2>, 50:
<CHCL3>, 51: <CCL3>, 52: <CCL4>, 53: <ACCL>, 54: <CH3NO2>, 55: <CH2NO2>, 56: <CHNO2>, 57:
<ACNO2>, 58: <CS2>, 59: <CH3SH>, 60: <CH2SH>, 61: <FURFURAL>, 62: <DOH>, 63: <I>, 64:
<BR>, 65: <CH=-C>, 66: <C=-C>, 67: <DMSO>, 68: <ACRY>, 69: <CL-(C=C)>, 70: <C=C>, 71:
<ACF>, 72: <DMF>, 73: <HCON(CH2)2>, 74: <CF3>, 75: <CF2>, 76: <CF>, 77: <COO>, 78:
<SIH3>, 79: <SIH2>, 80: <SIH>, 81: <SI>, 82: <SIH2O>, 83: <SIHO>, 84: <SIO>, 85: <NMP>,
86: <CCL3F>, 87: <CCL2F>, 88: <HCCL2F>, 89: <HCCLF>, 90: <CCLF2>, 91: <HCCLF2>, 92:
<CCLF3>, 93: <CCL2F2>, 94: <AMH2>, 95: <AMHCH3>, 96: <AMHCH2>, 97: <AM(CH3)2>, 98:
<AMCH3CH2>, 99: <AM(CH2)2>, 100: <C2H5O2>, 101: <C2H4O2>, 102: <CH3S>, 103: <CH2S>, 104:
<CHS>, 105: <MORPH>, 106: <C4H4S>, 107: <C4H3S>, 108: <C4H2S>, 109: <H2C=CH2>, 110:
<CH=-CH>, 111: <NH3>, 112: <CO>, 113: <H2>, 114: <H2S>, 115: <N2>, 116: <AR>, 117: <CO2>,
118: <CH4>, 119: <O2>, 120: <D2>, 121: <SO2>, 122: <NO>, 123: <N2O>, 124: <SF6>, 125:
<HE>, 126: <NE>, 127: <KR>, 128: <XE>, 129: <HF>, 130: <HCL>, 131: <HBR>, 132: <HI>, 133:
<COS>, 134: <CHSH>, 135: <CSH>, 136: <H2COCH>, 137: <HCOCH>, 138: <HCOC>, 139: <H2COCH2>,
140: <H2COC>, 141: <COC>, 142: <F2>, 143: <CL2>, 144: <BR2>, 145: <HCN>, 146: <NO2>, 147:
<CF4>, 148: <O3>, 149: <CLNO>, 152: <CNH2>}
```

```
thermo.unifac.PSRKMG = {1: ('CH2', [1, 2, 3, 4]), 2: ('C=C', [5, 6, 7, 8, 70, 109]), 3:
('ACH', [9, 10]), 4: ('ACCH2', [11, 12, 13]), 5: ('OH', [14]), 6: ('CH3OH', [15]), 7:
('H2O', [16]), 8: ('ACOH', [17]), 9: ('CH2CO', [18, 19]), 10: ('CHO', [20]), 11: ('CCOO',
[21, 22]), 12: ('HCOO', [23]), 13: ('CH2O', [24, 25, 26, 27]), 14: ('CNH2', [28, 29, 30,
152]), 15: ('CNH', [31, 32, 33]), 16: ('(C)3N', [34, 35]), 17: ('ACNH2', [36]), 18:
('PYRIDINE', [37, 38, 39]), 19: ('CCN', [40, 41]), 20: ('COOH', [42, 43]), 21: ('CCL',
[44, 45, 46]), 22: ('CCL2', [47, 48, 49]), 23: ('CCL3', [50, 51]), 24: ('CCL4', [52]),
25: ('ACCL', [53]), 26: ('CNO2', [54, 55, 56]), 27: ('ACNO2', [57]), 28: ('CS2', [58]),
29: ('CH3SH', [59, 60, 134, 135]), 30: ('FURFURAL', [61]), 31: ('DOH', [62]), 32: ('I',
[63]), 33: ('BR', [64]), 34: ('C=-C', [65, 66, 110]), 35: ('DMSO', [67]), 36: ('ACRY',
[68]), 37: ('CLCC', [69]), 38: ('ACF', [71]), 39: ('DMF', [72, 73]), 40: ('CF2', [74, 75,
76]), 41: ('COO', [77]), 42: ('SIH2', [78, 79, 80, 81]), 43: ('SIO', [82, 83, 84]), 44:
('NMP', [85]), 45: ('CCLF', [86, 87, 88, 89, 90, 91, 92, 93]), 46: ('CON (AM)', [94, 95,
96, 97, 98, 99]), 47: ('OCCOH', [100, 101]), 48: ('CH2S', [102, 103, 104]), 49: ('MORPH',
[105]), 50: ('THIOPHEN', [106, 107, 108]), 51: ('EPOXY', [136, 137, 138, 139, 140, 141]),
55: ('NH3', [111]), 56: ('CO2', [117]), 57: ('CH4', [118]), 58: ('O2', [119]), 59: ('AR',
[116]), 60: ('N2', [115]), 61: ('H2S', [114]), 62: ('H2', [113, 120]), 63: ('CO', [112]),
65: ('SO2', [121]), 66: ('NO', [122]), 67: ('N2O', [123]), 68: ('SF6', [124]), 69: ('HE',
[125]), 70: ('NE', [126]), 71: ('KR', [127]), 72: ('XE', [128]), 73: ('HF', [129]), 74:
('HCL', [130]), 75: ('HBR', [131]), 76: ('HI', [132]), 77: ('COS', [133]), 78: ('F2',
[142]), 79: ('CL2', [143]), 80: ('BR2', [144]), 81: ('HCN', [145]), 82: ('NO2', [146]),
83: ('CF4', [147]), 84: ('O3', [148]), 85: ('CLNO', [149])}
```

Magnussen, Thomas, Peter Rasmussen, and Aage Fredenslund. "UNIFAC Parameter Table for Prediction of Liquid-Liqu
Industrial & Engineering Chemistry Process Design and Development 20, no. 2 (April 1, 1981): 331-39. https://doi.org/10.1021/i200013a024.

`thermo.unifac.PSRKIP`

Interaction parameters for the PSRKIP UNIFAC model.

Type dict[int: dict[int: tuple(float, 3)]]

7.29.11 Data for VTPR UNIFAC

thermo.unifac.**VTPRSG** = {1: <CH3>, 2: <CH2>, 3: <CH>, 4: <C>, 5: <CH2=CH>, 6: <CH=CH>, 7: <CH2=C>, 8: <CH=C>, 9: <ACH>, 10: <AC>, 11: <ACCH3>, 12: <ACCH2>, 13: <ACCH>, 14: <OH(P)>, 15: <CH3OH>, 16: <H2O>, 17: <ACOH>, 18: <CH3CO>, 19: <CH2CO>, 20: <CHO>, 21: <CH3COO>, 22: <CH2COO>, 23: <HCOO>, 24: <CH3O>, 25: <CH2O>, 26: <CHO>, 27: <THF>, 28: <CH3NH2>, 29: <CH2NH2>, 30: <CHNH2>, 31: <CH3NH>, 32: <CH2NH>, 33: <CHNH>, 34: <CH3N>, 35: <CH2N>, 36: <ACNH2>, 40: <CH3CN>, 41: <CH2CN>, 44: <CH2CL>, 45: <CHCL>, 46: <CCL>, 47: <CH2CL2>, 48: <CHCL2>, 49: <CCL2>, 50: <CHCL3>, 51: <CCL3>, 52: <CCL4>, 53: <ACCL>, 54: <CH3NO2>, 55: <CH2NO2>, 56: <CHNO2>, 58: <CS2>, 59: <CH3SH>, 60: <CH2SH>, 61: <FURFURAL>, 62: <DOH>, 63: <I>, 64:
, 67: <DMSO>, 70: <C=C>, 72: <DMF>, 73: <HCON(..>, 78: <CY-CH2>, 79: <CY-CH>, 80: <CY-C>, 81: <OH(S)>, 82: <OH(T)>, 83: <CY-CH2O>, 84: <TRIOXAN>, 85: <CNH2>, 86: <NMP>, 87: <NEP>, 88: <NIPP>, 89: <NTBP>, 97: <Allene>, 98: <=CHCH=>, 99: <=CCH=>, 107: <H2COCH>, 108: <COCH>, 109: <HCOCH>, 116: <AC-CHO>, 119: <H2COCH2>, 129: <CHCOO>, 139: <CF2H>, 140: <CF2H2>, 142: <CF2Cl>, 143: <CF2Cl2>, 146: <CF4>, 148: <CF3Br>, 153: <H2COC>, 180: <CHCOO>, 250: <H2C=CH2>, 300: <NH3>, 301: <CO>, 302: <H2>, 303: <H2S>, 304: <N2>, 305: <Ar>, 306: <CO2>, 307: <CH4>, 308: <O2>, 309: <D2>, 310: <SO2>, 312: <N2O>, 314: <He>, 315: <Ne>, 319: <HCl>, 345: <Hg>}

thermo.unifac.**VTPRMG** = {1: ('CH2', [1, 2, 3, 4]), 2: ('H2C=CH2', [5, 6, 7, 8, 70, 97, 98, 99, 250]), 3: ('ACH', [9, 10]), 4: ('ACCH2', [11, 12, 13]), 5: ('OH', [14, 81, 82]), 6: ('CH3OH', [15]), 7: ('H2O', [16]), 8: ('ACOH', [17]), 9: ('CH2CO', [18, 19]), 10: ('CHO', [20]), 11: ('CCOO', [21, 22, 129, 180]), 12: ('HCOO', [23]), 13: ('CH2O', [24, 25, 26]), 14: ('CH2NH2', [28, 29, 30, 85]), 15: ('CH2NH', [31, 32, 33]), 16: ('(C)3N', [34, 35]), 17: ('ACNH2', [36]), 19: ('CH2CN', [40, 41]), 21: ('CCL', [44, 45, 46]), 22: ('CCL2', [47, 48, 49]), 23: ('CCL3', [51]), 24: ('CCL4', [52]), 25: ('ACCL', [53]), 26: ('CNO2', [54, 55, 56]), 28: ('CS2', [58]), 29: ('CH3SH', [59, 60]), 30: ('FURFURAL', [61]), 31: ('DOH', [62]), 32: ('I', [63]), 33: ('BR', [64]), 35: ('DMSO', [67]), 39: ('DMF', [72, 73]), 42: ('CY-CH2', [78, 79, 80]), 43: ('CY-CH2O', [27, 83, 84]), 45: ('CHCL3', [50]), 46: ('CY-CONC', [86, 87, 88, 89]), 53: ('EPOXIDES', [107, 108, 109, 119, 153]), 57: ('AC-CHO', [116]), 68: ('CF2H', [139, 140]), 70: ('CF2Cl2', [142, 143, 148]), 73: ('CF4', [146]), 150: ('NH3', [300]), 151: ('CO2', [306]), 152: ('CH4', [307]), 153: ('O2', [308]), 154: ('Ar', [305]), 155: ('N2', [304]), 156: ('H2S', [303]), 157: ('D2', [302, 309]), 158: ('CO', [301]), 160: ('SO2', [310]), 162: ('N2O', [312]), 164: ('He', [314]), 165: ('Ne', [315]), 169: ('HCl', [319]), 185: ('Hg', [345])}

thermo.unifac.**VTPRIP**

Interaction parameters for the VTPRIP UNIFAC model.

Type dict[int: dict[int: tuple(float, 3)]]

7.30 Support for pint Quantities (thermo.units)

Basic module which wraps some of thermo functions and classes to be compatible with the pint unit handling library. All other object - dicts, lists, etc - are not wrapped.

```
>>> from fluids.units import *
>>> import thermo
>>> thermo.units.PRMIX
<class 'fluids.units.PRMIX'>
```

```
>>> kwargs = dict(T=400.0*u.degC, P=30*u.psi, Tcs=[126.1, 190.6]*u.K, Pcs=[33.94E5, 46.
↪04E5]*u.Pa, omegas=[0.04, 0.011]*u.dimensionless, zs=[0.5, 0.5]*u.dimensionless,␣
↪kijs=[[0.0, 0.0289], [0.0289, 0.0]]*u.dimensionless)
>>> thermo.units.PRMIX(**kwargs)
PRMIX(Tcs=array([126.1, 190.6]), Pcs=array([3394000., 4604000.]), omegas=array([0.04 , 0.
↪011]), kijs=array([[0.    , 0.0289],
          [0.0289, 0.    ]]), zs=array([0.5, 0.5]), T=673.15, P=206842.7187950509)
```

Note that values which can normally be numpy arrays or python lists, are required to always be numpy arrays in this interface.

This is interface is powerful but not complex enough to handle many of the objects in Thermo. A list of the types of classes which are not supported is as follows:

- TDependentProperty, TPDependentProperty, MixtureProperty
- Phase objects
- Flash object
- ChemicalConstantsPackage
- PropertyCorrelationsPackage

For further information on this interface, please see the documentation of fluids.units which is built in the same way.

7.31 Utilities and Base Classes (thermo.utils)

This module contains base classes for temperature T, pressure P, and composition zs dependent properties. These power the various interfaces for each property.

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

7.31.1 Temperature Dependent

class thermo.utils.**TDependentProperty**(*extrapolation*, ***kwargs*)

Class for calculating temperature-dependent chemical properties.

On creation, a `TDependentProperty` examines all the possible methods implemented for calculating the property, loads whichever coefficients it needs (unless *load_data* is set to False), examines its input parameters, and selects the method it prefers. This method will continue to be used for all calculations until the method is changed by setting a new method to the to `method` attribute.

The default list of preferred method orderings is at `ranked_methods` for all properties; the order can be modified there in-place, and this will take effect on all new `TDependentProperty` instances created but NOT on existing instances.

All methods have defined criteria for determining if they are valid before calculation, i.e. a minimum and maximum temperature for coefficients to be valid. For constant property values used due to lack of temperature-dependent data, a short range is normally specified as valid.

It is not assumed that a specified method will succeed; for example many expressions are not mathematically valid past the critical point, and in some cases there is no easy way to determine the temperature where a property stops being reasonable.

Accordingly, all properties calculated are checked by a sanity function *test_property_validity*, which has basic sanity checks. If the property is not reasonable, None is returned.

This framework also supports tabular data, which is interpolated from if specified. Interpolation is cubic-spline based if 5 or more points are given, and linearly interpolated with if few points are given. A transform may be applied so that a property such as vapor pressure can be interpolated non-linearly. These are functions or lambda expressions which are set for the variables *interpolation_T*, *interpolation_property*, and *interpolation_property_inv*.

In order to calculate properties outside of the range of their correlations, a number of extrapolation method are available. Extrapolation is used by default on some properties but not all. The extrapolation methods available are as follows:

- 'constant' - returns the model values as calculated at the temperature limits
- 'linear' - fits the model at its temperature limits to a linear model
- 'nolimit' - attempt to evaluate the model outside of its limits; this will error in most cases and return None
- 'interp1d' - SciPy's `interp1d` is used to extrapolate
- 'AntoineAB' - fits the model to `Antoine`'s equation at the temperature limits using only the A and B coefficient
- 'DIPPR101_ABC' - fits the model at its temperature limits to the `EQ101` equation
- 'Watson' - fits the model to the Heat of Vaporization model `Watson`
- 'EXP_POLY_LN_TAU2' - uses the models's critical temperature and derivative to fit the model linearly in the equation $\text{prop} = \exp(a + b \cdot \ln \tau)$, so that it is always zero at the critical point; suitable for surface tension.
- 'DIPPR106_AB' - uses the models's critical temperature and derivative to fit the model linearly in the equation `EQ106`'s equation at the temperature limits using only the A and B coefficient
- 'DIPPR106_ABC' - uses the models's critical temperature and first two derivatives to fit the model quadratically in the equation `EQ106`'s equation at the temperature limits using only the A, B, and C coefficient.

It is possible to use different extrapolation methods for the low-temperature and the high-temperature region. Specify the extrapolation parameter with the '|' symbols between the two methods; the first method is used for low-temperature, and the second for the high-temperature.

Attributes

name [str] The name of the property being calculated, [-]

units [str] The units of the property, [-]

method [str] Method used to set a specific property method or to obtain the name of the method in use.

interpolation_T [callable or None] A function or lambda expression to transform the temperatures of tabular data for interpolation; e.g. 'lambda self, T: 1./T'

interpolation_T_inv [callable or None] A function or lambda expression to invert the transform of temperatures of tabular data for interpolation; e.g. 'lambda self, x: self.Tc*(1 - x)'

interpolation_property [callable or None] A function or lambda expression to transform tabular property values prior to interpolation; e.g. 'lambda self, P: log(P)'

interpolation_property_inv [callable or None] A function or property expression to transform interpolated property values from the transform performed by *interpolation_property* back to their actual form, e.g. 'lambda self, P: exp(P)'

Tmin [float] Minimum temperature (K) at which the current method can calculate the property.

Tmax [float] Maximum temperature (K) at which the current method can calculate the property.

property_min [float] Lowest value expected for a property while still being valid; this is a criteria used by *test_method_validity*.

property_max [float] Highest value expected for a property while still being valid; this is a criteria used by *test_method_validity*.

ranked_methods [list] Constant list of ranked methods by default

tabular_data [dict] Stores all user-supplied property data for interpolation in format {name: (Ts, properties)}, [-]

tabular_data_interpolators [dict] Stores all interpolation objects, idexed by name and property transform methods with the format {(name, interpolation_T, interpolation_property, interpolation_property_inv): (extrapolator, spline)}, [-]

all_methods [set] Set of all methods available for a given CASRN and set of properties, [-]

Methods

T_dependent_property(T)	Method to calculate the property with sanity checking and using the selected *method*.
T_dependent_property_derivative(T[, order])	Method to obtain a derivative of a property with respect to temperature, of a given order.
T_dependent_property_integral(T1, T2)	Method to calculate the integral of a property with respect to temperature, using the selected method.
T_dependent_property_integral_over_T(T1, T2)	Method to calculate the integral of a property over temperature with respect to temperature, using the selected method.
__call__(T)	Convenience method to calculate the property; calls *T_dependent_property*.
add_correlation(name, model, Tmin, Tmax, ...)	Method to add a new set of emperical fit equation coefficients to the object and select it for future property calculations.
add_method(f[, Tmin, Tmax, f_der, f_der2, ...])	Define a new method and select it for future property calculations.
add_tabular_data(Ts, properties[, name, ...])	Method to set tabular data to be used for interpolation.
as_json([references])	Method to create a JSON serialization of the property model which can be stored, and reloaded later.
calculate(T, method)	Method to calculate a property with a specified method, with no validity checking or error handling.
calculate_derivative(T, method[, order])	Method to calculate a derivative of a property with respect to temperature, of a given order using a specified method.
calculate_integral(T1, T2, method)	Method to calculate the integral of a property with respect to temperature, using a specified method.

continues on next page

Table 97 – continued from previous page

calculate_integral_over_T(T1, T2, method)	Method to calculate the integral of a property over temperature with respect to temperature, using a specified method.
extrapolate(T, method[, in_range])	Method to perform extrapolation on a given method according to the extrapolation setting.
fit_add_model(name, model, Ts, data, **kwargs)	Method to add a new emperical fit equation to the object by fitting its coefficients to specified data.
fit_data_to_model(Ts, data, model[, ...])	Method to fit T-dependent property data to one of the available model correlations.
from_json(json_repr)	Method to create a property model from a JSON serialization of another property model.
interpolate(T, name)	Method to perform interpolation on a given tabular data set previously added via add_tabular_data.
plot_T_dependent_property([Tmin, Tmax, ...])	Method to create a plot of the property vs temperature according to either a specified list of methods, or user methods (if set), or all methods.
polynomial_from_method(method[, n, start_n, ...])	Method to fit a T-dependent property to a polynomial.
solve_property(goal)	Method to solve for the temperature at which a property is at a specified value.
test_method_validity(T, method)	Method to test the validity of a specified method for a given temperature.
test_property_validity(prop)	Method to test the validity of a calculated property.
valid_methods([T])	Method to obtain a sorted list of methods that have data available to be used.

T_dependent_property(*T*)

Method to calculate the property with sanity checking and using the selected method.

In the unlikely event the calculation of the property fails, None is returned.

The calculated result is checked with test_property_validity and None is returned if the calculated value is nonsensical.

Parameters

T [float] Temperature at which to calculate the property, [K]

Returns

prop [float] Calculated property, [*units*]

T_dependent_property_derivative(*T*, *order=1*)

Method to obtain a derivative of a property with respect to temperature, of a given order.

Calls calculate_derivative internally to perform the actual calculation.

$$\text{derivative} = \frac{d(\text{property})}{dT}$$

Parameters

T [float] Temperature at which to calculate the derivative, [K]

order [int] Order of the derivative, >= 1

Returns

derivative [float] Calculated derivative property, [*units/K^order*]

T_dependent_property_integral(*T1*, *T2*)

Method to calculate the integral of a property with respect to temperature, using the selected method.

Calls *calculate_integral* internally to perform the actual calculation.

$$\text{integral} = \int_{T_1}^{T_2} \text{property } dT$$

Parameters

T1 [float] Lower limit of integration, [K]

T2 [float] Upper limit of integration, [K]

Returns

integral [float] Calculated integral of the property over the given range, [*units*K*]

T_dependent_property_integral_over_T(*T1*, *T2*)

Method to calculate the integral of a property over temperature with respect to temperature, using the selected method.

Calls *calculate_integral_over_T* internally to perform the actual calculation.

$$\text{integral} = \int_{T_1}^{T_2} \frac{\text{property}}{T} \, dT$$

Parameters

T1 [float] Lower limit of integration, [K]

T2 [float] Upper limit of integration, [K]

Returns

integral [float] Calculated integral of the property over the given range, [*units*]

T_limits = {}

Dictionary containing method: (Tmin, Tmax) pairs for all methods applicable to the chemical

__call__(*T*)

Convenience method to calculate the property; calls *T_dependent_property*. Caches previously calculated value, which is an overhead when calculating many different values of a property. See *T_dependent_property* for more details as to the calculation procedure.

Parameters

T [float] Temperature at which to calculate the property, [K]

Returns

prop [float] Calculated property, [*units*]

__repr__()

Create and return a string representation of the object. The design of the return string is such that it can be `eval`'d into itself. This is very convinient for creating tests. Note that several methods are not compatible with the `eval`'ing principle.

Returns

repr [str] String representation, [-]

add_correlation(*name*, *model*, *Tmin*, *Tmax*, ***kwargs*)

Method to add a new set of emperical fit equation coefficients to the object and select it for future property calculations.

A number of hardcoded *model* names are implemented; other models are not supported.

Parameters

name [str] The name of the coefficient set; user specified, [-]

model [str] A string representing the supported models, [-]

Tmin [float] Minimum temperature to use the method at, [K]

Tmax [float] Maximum temperature to use the method at, [K]

kwargs [dict] Various keyword arguments accepted by the model, [-]

Notes

The correlation models and links to their functions, describing their parameters, are as follows:

- "Antoine": Antoine, required parameters ('A', 'B', 'C'), optional parameters ('base',).
- "TRC_Antoine_extended": TRC_Antoine_extended, required parameters ('Tc', 'to', 'A', 'B', 'C', 'n', 'E', 'F').
- "Wagner_original": Wagner_original, required parameters ('Tc', 'Pc', 'a', 'b', 'c', 'd').
- "Wagner": Wagner, required parameters ('Tc', 'Pc', 'a', 'b', 'c', 'd').
- "Yaws_Psat": Yaws_Psat, required parameters ('A', 'B', 'C', 'D', 'E').
- "TDE_PVExpansion": TDE_PVExpansion, required parameters ('a1', 'a2', 'a3'), optional parameters ('a4', 'a5', 'a6', 'a7', 'a8').
- "Alibakhshi": Alibakhshi, required parameters ('Tc', 'C').
- "PPDS12": PPDS12, required parameters ('Tc', 'A', 'B', 'C', 'D', 'E').
- "Watson": Watson, required parameters ('Hvap_ref', 'T_ref', 'Tc'), optional parameters ('exponent',).
- "Viswanath_Natarajan_2": Viswanath_Natarajan_2, required parameters ('A', 'B').
- "Viswanath_Natarajan_2_exponential": Viswanath_Natarajan_2_exponential, required parameters ('C', 'D').
- "Viswanath_Natarajan_3": Viswanath_Natarajan_3, required parameters ('A', 'B', 'C').
- "PPDS5": PPDS5, required parameters ('Tc', 'a0', 'a1', 'a2').
- "mu_TDE": mu_TDE, required parameters ('A', 'B', 'C', 'D').
- "PPDS9": PPDS9, required parameters ('A', 'B', 'C', 'D', 'E').
- "mu_Yaws": mu_Yaws, required parameters ('A', 'B'), optional parameters ('C', 'D').
- "Poling": Poling, required parameters ('a', 'b', 'c', 'd', 'e').
- "TRCCp": TRCCp, required parameters ('a0', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7').
- "Zabransky_quasi_polynomial": Zabransky_quasi_polynomial, required parameters ('Tc', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6').
- "Zabransky_cubic": Zabransky_cubic, required parameters ('a1', 'a2', 'a3', 'a4').
- "REFPROP_sigma": REFPROP_sigma, required parameters ('Tc', 'sigma0', 'n0'), optional parameters ('sigma1', 'n1', 'sigma2', 'n2').
- "Somayajulu": Somayajulu, required parameters ('Tc', 'A', 'B', 'C').
- "Jasper": Jasper, required parameters ('a', 'b').
- "PPDS14": PPDS14, required parameters ('Tc', 'a0', 'a1', 'a2').

- "Watson_sigma": Watson_sigma, required parameters ('Tc', 'a1', 'a2', 'a3', 'a4', 'a5').
- "ISTExpansion": ISTExpansion, required parameters ('Tc', 'a1', 'a2', 'a3', 'a4', 'a5').
- "Chemsep_16": Chemsep_16, required parameters ('A', 'B', 'C', 'D', 'E').
- "PPDS8": PPDS8, required parameters ('Tc', 'a0', 'a1', 'a2', 'a3').
- "PPDS3": PPDS3, required parameters ('Tc', 'a1', 'a2', 'a3').
- "TDE_VDNS_rho": TDE_VDNS_rho, required parameters ('Tc', 'rhoc', 'a1', 'a2', 'a3', 'a4', 'MW').
- "PPDS17": PPDS17, required parameters ('Tc', 'a0', 'a1', 'a2', 'MW').
- "volume_VDI_PPDS": volume_VDI_PPDS, required parameters ('Tc', 'rhoc', 'a', 'b', 'c', 'd', 'MW').
- "Rackett_fit": Rackett_fit, required parameters ('Tc', 'rhoc', 'b', 'n', 'MW').
- "DIPPR100": EQ100, required parameters (), optional parameters ('A', 'B', 'C', 'D', 'E', 'F', 'G').
- "constant": EQ100, required parameters (), optional parameters ('A',).
- "linear": EQ100, required parameters (), optional parameters ('A', 'B').
- "quadratic": EQ100, required parameters (), optional parameters ('A', 'B', 'C').
- "cubic": EQ100, required parameters (), optional parameters ('A', 'B', 'C', 'D').
- "quintic": EQ100, required parameters (), optional parameters ('A', 'B', 'C', 'D', 'E').
- "polynomial": horner_backwards, required parameters ('coeffs',).
- "exp_polynomial": exp_horner_backwards, required parameters ('coeffs',).
- "polynomial_ln_tau": horner_backwards_ln_tau, required parameters ('Tc', 'coeffs').
- "exp_polynomial_ln_tau": exp_horner_backwards_ln_tau, required parameters ('Tc', 'coeffs').
- "DIPPR101": EQ101, required parameters ('A', 'B'), optional parameters ('C', 'D', 'E').
- "DIPPR102": EQ102, required parameters ('A', 'B', 'C', 'D').
- "DIPPR104": EQ104, required parameters ('A', 'B'), optional parameters ('C', 'D', 'E').
- "DIPPR105": EQ105, required parameters ('A', 'B', 'C', 'D').
- "DIPPR106": EQ106, required parameters ('Tc', 'A', 'B'), optional parameters ('C', 'D', 'E').
- "YawsSigma": EQ106, required parameters ('Tc', 'A', 'B'), optional parameters ('C', 'D', 'E').
- "DIPPR107": EQ107, required parameters (), optional parameters ('A', 'B', 'C', 'D', 'E').
- "DIPPR114": EQ114, required parameters ('Tc', 'A', 'B', 'C', 'D').
- "DIPPR115": EQ115, required parameters ('A', 'B'), optional parameters ('C', 'D', 'E').
- "DIPPR116": EQ116, required parameters ('Tc', 'A', 'B', 'C', 'D', 'E').
- "DIPPR127": EQ127, required parameters ('A', 'B', 'C', 'D', 'E', 'F', 'G').
- "Twu91_alpha_pure": *Twu91_alpha_pure*, required parameters ('Tc', 'c0', 'c1', 'c2').
- "Heyen_alpha_pure": *Heyen_alpha_pure*, required parameters ('Tc', 'c1', 'c2').
- "Harmens_Knapp_alpha_pure": *Harmens_Knapp_alpha_pure*, required parameters ('Tc', 'c1', 'c2').
- "Mathias_Copeman_untruncated_alpha_pure": *Mathias_Copeman_untruncated_alpha_pure*, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Mathias_1983_alpha_pure": *Mathias_1983_alpha_pure*, required parameters ('Tc', 'c1', 'c2').

- "Soave_1972_alpha_pure": `Soave_1972_alpha_pure`, required parameters ('Tc', 'c0').
- "Soave_1979_alpha_pure": `Soave_1979_alpha_pure`, required parameters ('Tc', 'M', 'N').
- "Gibbons_Laughton_alpha_pure": `Gibbons_Laughton_alpha_pure`, required parameters ('Tc', 'c1', 'c2').
- "Soave_1984_alpha_pure": `Soave_1984_alpha_pure`, required parameters ('Tc', 'c1', 'c2').
- "Yu_Lu_alpha_pure": `Yu_Lu_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3', 'c4').
- "Trebble_Bishnoi_alpha_pure": `Trebble_Bishnoi_alpha_pure`, required parameters ('Tc', 'c1').
- "Melhem_alpha_pure": `Melhem_alpha_pure`, required parameters ('Tc', 'c1', 'c2').
- "Androulakis_alpha_pure": `Androulakis_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Schwartzentruber_alpha_pure": `Schwartzentruber_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3', 'c4').
- "Almeida_alpha_pure": `Almeida_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Soave_1993_alpha_pure": `Soave_1993_alpha_pure`, required parameters ('Tc', 'c1', 'c2').
- "Gasem_alpha_pure": `Gasem_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Coquelet_alpha_pure": `Coquelet_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Haghtalab_alpha_pure": `Haghtalab_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Saffari_alpha_pure": `Saffari_alpha_pure`, required parameters ('Tc', 'c1', 'c2', 'c3').
- "Chen_Yang_alpha_pure": `Chen_Yang_alpha_pure`, required parameters ('Tc', 'omega', 'c1', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7').
- "Wagner2,5": `Wagner`, required parameters ('Tc', 'Pc', 'a', 'b', 'c', 'd').
- "Wagner3,6": `Wagner_original`, required parameters ('Tc', 'Pc', 'a', 'b', 'c', 'd').
- "Andrade": `Viswanath_Natarajan_2`, required parameters ('A', 'B').
- "YawsHvap": `EQ106`, required parameters ('Tc', 'A', 'B'), optional parameters ('C', 'D', 'E').

add_method(*f*, *Tmin=None*, *Tmax=None*, *f_der=None*, *f_der2=None*, *f_der3=None*, *f_int=None*, *f_int_over_T=None*, *name=None*)

Define a new method and select it for future property calculations.

Parameters

f [callable] Object which calculates the property given the temperature in K, [-]

Tmin [float, optional] Minimum temperature to use the method at, [K]

Tmax [float, optional] Maximum temperature to use the method at, [K]

f_der [callable, optional] If specified, should take as an argument the temperature and return the first derivative of the property, [-]

f_der2 [callable, optional] If specified, should take as an argument the temperature and return the second derivative of the property, [-]

f_der3 [callable, optional] If specified, should take as an argument the temperature and return the third derivative of the property, [-]

f_int [callable, optional] If specified, should take *T1* and *T2* and return the integral of the property from *T1* to *T2*, [-]

f_int_over_T [callable, optional] If specified, should take *T1* and *T2* and return the integral of the property over T from *T1* to *T2*, [-]

name [str, optional] Name of method.

Notes

Once a custom method has been added to an object, that object can no longer be serialized to json and the *TDependentProperty.__repr__* method can no longer be used to reconstruct the object completely.

add_tabular_data(*Ts*, *properties*, *name=None*, *check_properties=True*)

Method to set tabular data to be used for interpolation. Ts must be in increasing order. If no name is given, data will be assigned the name 'Tabular data series #x', where x is the number of previously added tabular data series.

After adding the data, this method becomes the selected method.

Parameters

Ts [array-like] Increasing array of temperatures at which properties are specified, [K]

properties [array-like] List of properties at Ts, [*units*]

name [str, optional] Name assigned to the data

check_properties [bool] If True, the properties will be checked for validity with *test_property_validity* and raise an exception if any are not valid

as_json(*references=1*)

Method to create a JSON serialization of the property model which can be stored, and reloaded later.

Parameters

references [int] How to handle references to other objects; internal parameter, [-]

Returns

json_repr [dict] JSON-friendly representation, [-]

calculate(*T*, *method*)

Method to calculate a property with a specified method, with no validity checking or error handling. Demo function for testing only; must be implemented according to the methods available for each individual method. Include the interpolation call here.

Parameters

T [float] Temperature at which to calculate the property, [K]

method [str] Method name to use

Returns

prop [float] Calculated property, [*units*]

calculate_derivative(*T*, *method*, *order=1*)

Method to calculate a derivative of a property with respect to temperature, of a given order using a specified method. Uses SciPy's derivative function, with a delta of 1E-6 K and a number of points equal to 2*order + 1.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

T [float] Temperature at which to calculate the derivative, [K]

method [str] Method for which to find the derivative

order [int] Order of the derivative, >= 1

Returns

derivative [float] Calculated derivative property, [*units/K^order*]

`calculate_integral`(*T1*, *T2*, *method*)

Method to calculate the integral of a property with respect to temperature, using a specified method. Uses SciPy's *quad* function to perform the integral, with no options.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

T1 [float] Lower limit of integration, [K]

T2 [float] Upper limit of integration, [K]

method [str] Method for which to find the integral

Returns

integral [float] Calculated integral of the property over the given range, [*units*K*]

`calculate_integral_over_T`(*T1*, *T2*, *method*)

Method to calculate the integral of a property over temperature with respect to temperature, using a specified method. Uses SciPy's *quad* function to perform the integral, with no options.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

T1 [float] Lower limit of integration, [K]

T2 [float] Upper limit of integration, [K]

method [str] Method for which to find the integral

Returns

integral [float] Calculated integral of the property over the given range, [*units*]

`critical_zero = False`

Whether or not the property is declining and reaching zero at the critical point. This is used by numerical solvers.

`extrapolate`(*T*, *method*, *in_range='error'*)

Method to perform extrapolation on a given method according to the `extrapolation` setting.

Parameters

T [float] Temperature at which to extrapolate the property, [K]

method [str] The method to use, [-]

in_range [str] How to handle inputs which are not outside the temperature limits; set to 'low' to use the low T extrapolation, 'high' to use the high T extrapolation, 'nearest' to use the nearest value, and 'error' or anything else to raise an error in those cases, [-]

Returns

prop [float] Calculated property, [*units*]

property extrapolation

The string setting of the current extrapolation settings. This can be set to a new value to change which extrapolation setting is used.

fit_add_model(*name*, *model*, *Ts*, *data*, ***kwargs*)

Method to add a new emperical fit equation to the object by fitting its coefficients to specified data. Once added, the new method is set as the default.

A number of hardcoded *model* names are implemented; other models are not supported.

This is a wrapper around *TDependentProperty.fit_data_to_model* and *TDependentProperty.add_correlation*.

The data is also stored in the object as a tabular method with the name *name`+'_data', through :obj:`TDependentProperty.add_tabular_data*.

Parameters

name [str] The name of the coefficient set; user specified, [-]

model [str] A string representing the supported models, [-]

Ts [list[float]] Temperatures of the data points, [K]

data [list[float]] Data points, [*units*]

kwargs [dict] Various keyword arguments accepted by *fit_data_to_model*, [-]

classmethod fit_data_to_model(*Ts*, *data*, *model*, *model_kwargs=None*, *fit_method='lm'*, *sigma=None*, *use_numba=False*, *do_statistics=False*, *guesses=None*, *solver_kwargs=None*, *objective='MeanSquareErr'*, *multiple_tries=False*, *multiple_tries_max_err=1e-05*, *multiple_tries_max_objective='MeanRelErr'*)

Method to fit T-dependent property data to one of the available model correlations.

Parameters

Ts [list[float]] Temperatures of the data points, [K]

data [list[float]] Data points, [*units*]

model [str] A string representing the supported models, [-]

model_kwargs [dict, optional] Various keyword arguments accepted by the model; not necessary for most models. Parameters which are normally fit, can be specified here as well with a constant value and then that fixed value will be used instead of fitting the parameter. [-]

fit_method [str, optional] The fit method to use; one of {*lm*, *trf*, *dogbox*, *differential_evolution*}, [-]

sigma [None or list[float]] Uncertainty parameters used by *curve_fit*, [-]

use_numba [bool, optional] Whether or not to try to use numba to speed up the computation, [-]

do_statistics [bool, optional] Whether or not to compute statistical measures on the outputs, [-]

guesses [dict[str: float], optional] Parameter guesses, by name; any number of parameters can be specified, [-]

solver_kwargs [dict] Extra parameters to be passed to the solver chosen, [-]

objective [str] The minimimization criteria; supported by *differential_evolution*. One of:

- 'MeanAbsErr': Mean absolute error
- 'MeanRelErr': Mean relative error
- 'MeanSquareErr': Mean squared absolute error
- 'MeanSquareRelErr': Mean squared relative error
- 'MaxAbsErr': Maximum absolute error
- 'MaxRelErr': Maximum relative error
- 'MaxSquareErr': Maximum squared absolute error
- 'MaxSquareRelErr': Maximum squared relative error

multiple_tries [bool or int] For most solvers, multiple initial guesses are available and the best guess is normally tried. When this is set to True, all guesses are tried until one is found with an error lower than *multiple_tries_max_err*. If an int is supplied, the best *multiple_tries* guesses are tried only. [-]

multiple_tries_max_err [float] Only used when *multiple_tries* is true; if a solution is found with lower error than this, no further guesses are tried, [-]

multiple_tries_max_objective [str] The error criteria to use for minimization, [-]

Returns

coefficients [dict[str: float]] Calculated coefficients, [*various*]

statistics [dict[str: float]] Statistics, calculated and returned only if *do_statistics* is True, [-]

classmethod from_json(*json_repr*)

Method to create a property model from a JSON serialization of another property model.

Parameters

json_repr [dict] JSON-friendly representation, [-]

Returns

model [`TDependentProperty` or `TPDependentProperty`] Newly created object from the json serialization, [-]

Notes

It is important that the input string be in the same format as that created by `TDependentProperty.as_json`.

interpolate(*T*, *name*)

Method to perform interpolation on a given tabular data set previously added via `add_tabular_data`. This method will create the interpolators the first time it is used on a property set, and store them for quick future use.

Interpolation is cubic-spline based if 5 or more points are available, and linearly interpolated if not. Extrapolation is always performed linearly. This function uses the transforms `interpolation_T`, `interpolation_property`, and `interpolation_property_inv` if set. If any of these are changed after the interpolators were first created, new interpolators are created with the new transforms. All interpolation is performed via the *interp1d* function.

Parameters

T [float] Temperature at which to interpolate the property, [K]

name [str] The name assigned to the tabular data set

Returns

prop [float] Calculated property, [*units*]

`interpolation_T = None`

`interpolation_T_inv = None`

`interpolation_property = None`

`interpolation_property_inv = None`

`property method`

Method used to set a specific property method or to obtain the name of the method in use.

When setting a method, an exception is raised if the method specified isn't available for the chemical with the provided information.

If *method* is None, no calculations can be performed.

Parameters

method [str] Method to use, [-]

`name = 'Property name'`

`plot_T_dependent_property`(*Tmin=None*, *Tmax=None*, *methods=[]*, *pts=250*, *only_valid=True*, *order=0*, *show=True*, *tabular_points=True*, *axes='semilogy'*)

Method to create a plot of the property vs temperature according to either a specified list of methods, or user methods (if set), or all methods. User-selectable number of points, and temperature range. If only_valid is set,:obj:*test_method_validity* will be used to check if each temperature in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the method fails.

Parameters

Tmin [float] Minimum temperature, to begin calculating the property, [K]

Tmax [float] Maximum temperature, to stop calculating the property, [K]

methods [list, optional] List of methods to consider

pts [int, optional] A list of points to calculate the property at; if Tmin to Tmax covers a wide range of method validities, only a few points may end up calculated for a given method so this may need to be large

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

show [bool] If True, displays the plot; otherwise, returns it

tabular_points [bool, optional] If True, tabular data will only be shows as the original points; otherwise interpolated values are shown, [-]

`polynomial_from_method`(*method*, *n=None*, *start_n=3*, *max_n=30*, *eval_pts=100*, *fit_form='POLY_FIT'*, *fit_method=None*)

Method to fit a T-dependent property to a polynomial. The degree of the polynomial can be specified with the *n* parameter, or it will be automatically selected for maximum accuracy.

Parameters

method [str] Method name to fit, [-]

n [int, optional] The degree of the polynomial, if specified

start_n [int] If *n* is not specified, all polynomials of degree *start_n* to *max_n* will be tried and the highest-accuracy will be selected; [-]

max_n [int] If *n* is not specified, all polynomials of degree *start_n* to *max_n* will be tried and the highest-accuracy will be selected; [-]

eval_pts [int] The number of points to evaluate the fitted functions at to check for accuracy; more is better but slower, [-]

fit_form [str] The shape of the polynomial; options are 'POLY_FIT', 'EXP_POLY_FIT', 'EXP_POLY_FIT_LN_TAU', and 'POLY_FIT_LN_TAU' [-]

Returns

coeffs [list[float]] Fit coefficients, [-]

Tmin [float] The minimum temperature used for the fitting, [K]

Tmax [float] The maximum temperature used for the fitting, [K]

err_avg [float] Mean error in the evaluated points, [-]

err_std [float] Standard deviation of errors in the evaluated points, [-]

min_ratio [float] Lowest ratio of calc/actual in any found points, [-]

max_ratio [float] Highest ratio of calc/actual in any found points, [-]

`property_max = 10000.0`

`property_min = 0`

`ranked_methods = []`

`solve_property`(*goal*)

Method to solve for the temperature at which a property is at a specified value. `T_dependent_property` is used to calculate the value of the property as a function of temperature.

Checks the given property value with `test_property_validity` first and raises an exception if it is not valid.

Parameters

goal [float] Propoerty value desired, [*units*]

Returns

T [float] Temperature at which the property is the specified value [K]

`test_method_validity`(*T*, *method*)

Method to test the validity of a specified method for a given temperature. Demo function for testing only; must be implemented according to the methods available for each individual method. Include the interpolation check here.

Parameters

T [float] Temperature at which to determine the validity of the method, [K]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

classmethod test_property_validity(*prop*)

Method to test the validity of a calculated property. Normally, this method is used by a given property class, and has maximum and minimum limits controlled by the variables `property_min` and `property_max`.

Parameters

prop [float] property to be tested, [*units*]

Returns

validity [bool] Whether or not a specifid method is valid

units = 'Property units'

valid_methods(*T=None*)

Method to obtain a sorted list of methods that have data available to be used. The methods are ranked in the following order:

- The currently selected method is first (if one is selected)
- Other available methods are ranked by the attribute `ranked_methods`

If *T* is provided, the methods will be checked against the temperature limits of the correlations as well.

Parameters

T [float or None] Temperature at which to test methods, [K]

Returns

sorted_valid_methods [list] Sorted lists of methods valid at T according to `test_method_validity`, [-]

7.31.2 Temperature and Pressure Dependent

class thermo.utils.**TPDependentProperty**(*extrapolation*, ***kwargs*)

Bases: `thermo.utils.t_dependent_property.TDependentProperty`

Class for calculating temperature and pressure dependent chemical properties.

On creation, a `TPDependentProperty` examines all the possible methods implemented for calculating the property, loads whichever coefficients it needs (unless *load_data* is set to False), examines its input parameters, and selects the method it prefers. This method will continue to be used for all calculations until the method is changed by setting a new method to the to `method` attribute.

Because many pressure dependent property methods are implemented as a low-pressure correlation and a high-pressure correlation, this class works essentially the same as `TDependentProperty` but with extra methods that accept pressure as a parameter.

The object also selects the pressure-dependent method it prefers. This method will continue to be used for all pressure-dependent calculations until the pressure-dependent method is changed by setting a new method_P to the to `method_P` attribute.

The default list of preferred pressure-dependent method orderings is at `ranked_methods_P` for all properties; the order can be modified there in-place, and this will take effect on all new `TPDependentProperty` instances created but NOT on existing instances.

Tabular data can be provided as either temperature-dependent or pressure-dependent data. The same *extrapolation* settings as in `TDependentProperty` are implemented here for the low-pressure correlations.

In addition to the methods and attributes shown here, all those from `TPDependentProperty` are also available.

Attributes

method_P [str] Method used to set or get a specific property method.

method [str] Method used to set a specific property method or to obtain the name of the method in use.

all_methods [set] All low-pressure methods available, [-]

all_methods_P [set] All pressure-dependent methods available, [-]

Methods

TP_dependent_property(T, P)	Method to calculate the property given a temperature and pressure according to the selected method_P and method.
TP_dependent_property_derivative_P(T, P[, order])	Method to calculate a derivative of a temperature and pressure dependent property with respect to pressure at constant temperature, of a given order, according to the selected method_P.
TP_dependent_property_derivative_T(T, P[, order])	Method to calculate a derivative of a temperature and pressure dependent property with respect to temperature at constant pressure, of a given order, according to the selected method_P.
TP_or_T_dependent_property(T, P)	Method to calculate the property given a temperature and pressure according to the selected method_P and method.
__call__(T, P)	Convenience method to calculate the property; calls TP_dependent_property.
add_method(f[, Tmin, Tmax, f_der, f_der2, ...])	Define a new method and select it for future property calculations.
add_tabular_data(Ts, properties[, name, ...])	Method to set tabular data to be used for interpolation.
add_tabular_data_P(Ts, Ps, properties[, ...])	Method to set tabular data to be used for interpolation.
calculate(T, method)	Method to calculate a property with a specified method, with no validity checking or error handling.
calculate_derivative_P(P, T, method[, order])	Method to calculate a derivative of a temperature and pressure dependent property with respect to pressure at constant temperature, of a given order using a specified method.
calculate_derivative_T(T, P, method[, order])	Method to calculate a derivative of a temperature and pressure dependent property with respect to temperature at constant pressure, of a given order using a specified method.
extrapolate(T, method[, in_range])	Method to perform extrapolation on a given method according to the extrapolation setting.
plot_TP_dependent_property([Tmin, Tmax, ...])	Method to create a plot of the property vs temperature and pressure according to either a specified list of methods, or user methods (if set), or all methods.
plot_isobar(P[, Tmin, Tmax, methods_P, pts, ...])	Method to create a plot of the property vs temperature at a specific pressure according to either a specified list of methods, or user methods (if set), or all methods.

continues on next page

Table 98 – continued from previous page

plot_isotherm(T[, Pmin, Pmax, methods_P, ...])	Method to create a plot of the property vs pressure at a specified temperature according to either a specified list of methods, or the user methods (if set), or all methods.
solve_property(goal)	Method to solve for the temperature at which a property is at a specified value.
test_method_validity(T, method)	Method to test the validity of a specified method for a given temperature.
test_property_validity(prop)	Method to test the validity of a calculated property.
valid_methods([T])	Method to obtain a sorted list of methods that have data available to be used.
valid_methods_P([T, P])	Method to obtain a sorted list of high-pressure methods that have data available to be used.

TP_dependent_property(*T*, *P*)

Method to calculate the property given a temperature and pressure according to the selected `method_P` and `method`. The pressure-dependent method is always used and required to succeed. The result is checked with `test_property_validity`.

If the method does not succeed, returns None.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

Returns

prop [float] Calculated property, [*units*]

TP_dependent_property_derivative_P(*T*, *P*, *order=1*)

Method to calculate a derivative of a temperature and pressure dependent property with respect to pressure at constant temperature, of a given order, according to the selected `method_P`.

Calls `calculate_derivative_P` internally to perform the actual calculation.

$$\text{derivative} = \frac{d(\text{property})}{dP}|_T$$

Parameters

T [float] Temperature at which to calculate the derivative, [K]

P [float] Pressure at which to calculate the derivative, [Pa]

order [int] Order of the derivative, >= 1

Returns

dprop_dP_T [float] Calculated derivative property, [*units/Pa^order*]

TP_dependent_property_derivative_T(*T*, *P*, *order=1*)

Method to calculate a derivative of a temperature and pressure dependent property with respect to temperature at constant pressure, of a given order, according to the selected `method_P`.

Calls `calculate_derivative_T` internally to perform the actual calculation.

$$\text{derivative} = \frac{d(\text{property})}{dT}|_P$$

Parameters

T [float] Temperature at which to calculate the derivative, [K]

P [float] Pressure at which to calculate the derivative, [Pa]

order [int] Order of the derivative, >= 1

Returns

dprop_dT_P [float] Calculated derivative property, [*units/K^order*]

TP_or_T_dependent_property(*T*, *P*)

Method to calculate the property given a temperature and pressure according to the selected `method_P` and `method`. The pressure-dependent method is always tried. The result is checked with `test_property_validity`.

If the pressure-dependent method does not succeed, the low-pressure method is tried and its result is returned.

> **Warning:** It can seem like a good idea to switch between a low-pressure and a high-pressure method if the high pressure method is not working, however it can cause discontinuities and prevent numerical methods from converging

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

Returns

prop [float] Calculated property, [*units*]

T_limits = {}

Dictionary containing method: (Tmin, Tmax) pairs for all methods applicable to the chemical

__call__(*T*, *P*)

Convenience method to calculate the property; calls `TP_dependent_property`. Caches previously calculated value, which is an overhead when calculating many different values of a property. See `TP_dependent_property` for more details as to the calculation procedure.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

Returns

prop [float] Calculated property, [*units*]

add_method(*f*, *Tmin=None*, *Tmax=None*, *f_der=None*, *f_der2=None*, *f_der3=None*, *f_int=None*, *f_int_over_T=None*, *name=None*)

Define a new method and select it for future property calculations.

Parameters

f [callable] Object which calculates the property given the temperature in K, [-]

Tmin [float, optional] Minimum temperature to use the method at, [K]

Tmax [float, optional] Maximum temperature to use the method at, [K]

f_der [callable, optional] If specified, should take as an argument the temperature and return the first derivative of the property, [-]

f_der2 [callable, optional] If specified, should take as an argument the temperature and return the second derivative of the property, [-]

f_der3 [callable, optional] If specified, should take as an argument the temperature and return the third derivative of the property, [-]

f_int [callable, optional] If specified, should take *T1* and *T2* and return the integral of the property from *T1* to *T2*, [-]

f_int_over_T [callable, optional] If specified, should take *T1* and *T2* and return the integral of the property over T from *T1* to *T2*, [-]

name [str, optional] Name of method.

Notes

Once a custom method has been added to an object, that object can no longer be serialized to json and the `TDependentProperty.__repr__` method can no longer be used to reconstruct the object completely.

`add_tabular_data`(*Ts*, *properties*, *name=None*, *check_properties=True*)

Method to set tabular data to be used for interpolation. Ts must be in increasing order. If no name is given, data will be assigned the name 'Tabular data series #x', where x is the number of previously added tabular data series.

After adding the data, this method becomes the selected method.

Parameters

Ts [array-like] Increasing array of temperatures at which properties are specified, [K]

properties [array-like] List of properties at Ts, [*units*]

name [str, optional] Name assigned to the data

check_properties [bool] If True, the properties will be checked for validity with `test_property_validity` and raise an exception if any are not valid

`add_tabular_data_P`(*Ts*, *Ps*, *properties*, *name=None*, *check_properties=True*)

Method to set tabular data to be used for interpolation. Ts and Psmust be in increasing order. If no name is given, data will be assigned the name 'Tabular data series #x', where x is the number of previously added tabular data series.

After adding the data, this method becomes the selected high-pressure method.

Parameters

Ts [array-like] Increasing array of temperatures at which properties are specified, [K]

Ps [array-like] Increasing array of pressures at which properties are specified, [Pa]

properties [array-like] List of properties at *Ts* and *Ps*; the data should be indexed [P][T], [*units*]

name [str, optional] Name assigned to the data

check_properties [bool] If True, the properties will be checked for validity with `test_property_validity` and raise an exception if any are not valid

calculate(*T*, *method*)

Method to calculate a property with a specified method, with no validity checking or error handling. Demo function for testing only; must be implemented according to the methods available for each individual method. Include the interpolation call here.

Parameters

T [float] Temperature at which to calculate the property, [K]

method [str] Method name to use

Returns

prop [float] Calculated property, [*units*]

calculate_derivative_P(*P*, *T*, *method*, *order=1*)

Method to calculate a derivative of a temperature and pressure dependent property with respect to pressure at constant temperature, of a given order using a specified method. Uses SciPy's derivative function, with a delta of 0.01 Pa and a number of points equal to 2*order + 1.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

P [float] Pressure at which to calculate the derivative, [Pa]

T [float] Temperature at which to calculate the derivative, [K]

method [str] Method for which to find the derivative

order [int] Order of the derivative, >= 1

Returns

dprop_dP_T [float] Calculated derivative property at constant temperature, [*units/Pa^order*]

calculate_derivative_T(*T*, *P*, *method*, *order=1*)

Method to calculate a derivative of a temperature and pressure dependent property with respect to temperature at constant pressure, of a given order using a specified method. Uses SciPy's derivative function, with a delta of 1E-6 K and a number of points equal to 2*order + 1.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

T [float] Temperature at which to calculate the derivative, [K]

P [float] Pressure at which to calculate the derivative, [Pa]

method [str] Method for which to find the derivative

order [int] Order of the derivative, >= 1

Returns

dprop_dT_P [float] Calculated derivative property at constant pressure, [*units/K^order*]

extrapolate(*T*, *method*, *in_range='error'*)

Method to perform extrapolation on a given method according to the `extrapolation` setting.

Parameters

T [float] Temperature at which to extrapolate the property, [K]

method [str] The method to use, [-]

in_range [str] How to handle inputs which are not outside the temperature limits; set to 'low' to use the low T extrapolation, 'high' to use the high T extrapolation, 'nearest' to use the nearest value, and 'error' or anything else to raise an error in those cases, [-]

Returns

prop [float] Calculated property, [*units*]

property extrapolation

The string setting of the current extrapolation settings. This can be set to a new value to change which extrapolation setting is used.

interpolation_T = None

interpolation_T_inv = None

interpolation_property = None

interpolation_property_inv = None

property method

Method used to set a specific property method or to obtain the name of the method in use.

When setting a method, an exception is raised if the method specified isn't available for the chemical with the provided information.

If *method* is None, no calculations can be performed.

Parameters

method [str] Method to use, [-]

property method_P

Method used to set or get a specific property method.

An exception is raised if the method specified isn't available for the chemical with the provided information.

Parameters

method [str or list] Methods by name to be considered or preferred

name = 'Property name'

plot_TP_dependent_property(*Tmin=None*, *Tmax=None*, *Pmin=None*, *Pmax=None*, *methods_P=[]*, *pts=15*, *only_valid=True*)

Method to create a plot of the property vs temperature and pressure according to either a specified list of methods, or user methods (if set), or all methods. User-selectable number of points for each variable. If only_valid is set,:obj:*test_method_validity_P* will be used to check if each condition in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the any method fails for any point.

Parameters

Tmin [float] Minimum temperature, to begin calculating the property, [K]

Tmax [float] Maximum temperature, to stop calculating the property, [K]

Pmin [float] Minimum pressure, to begin calculating the property, [Pa]

Pmax [float] Maximum pressure, to stop calculating the property, [Pa]

methods_P [list, optional] List of methods to plot

pts [int, optional] A list of points to calculate the property at for both temperature and pressure; pts^2 points will be calculated.

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

`plot_isobar`(*P*, *Tmin=None*, *Tmax=None*, *methods_P=[]*, *pts=50*, *only_valid=True*, *show=True*)

Method to create a plot of the property vs temperature at a specific pressure according to either a specified list of methods, or user methods (if set), or all methods. User-selectable number of points, and temperature range. If only_valid is set,:obj:*test_method_validity_P* will be used to check if each condition in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the method fails.

Parameters

P [float] Pressure for the isobar, [Pa]

Tmin [float] Minimum temperature, to begin calculating the property, [K]

Tmax [float] Maximum temperature, to stop calculating the property, [K]

methods_P [list, optional] List of methods to consider

pts [int, optional] A list of points to calculate the property at; if Tmin to Tmax covers a wide range of method validities, only a few points may end up calculated for a given method so this may need to be large

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

`plot_isotherm`(*T*, *Pmin=None*, *Pmax=None*, *methods_P=[]*, *pts=50*, *only_valid=True*, *show=True*)

Method to create a plot of the property vs pressure at a specified temperature according to either a specified list of methods, or the user methods (if set), or all methods. User-selectable number of points, and pressure range. If only_valid is set, `test_method_validity_P` will be used to check if each condition in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the method fails.

Parameters

T [float] Temperature at which to create the plot, [K]

Pmin [float] Minimum pressure, to begin calculating the property, [Pa]

Pmax [float] Maximum pressure, to stop calculating the property, [Pa]

methods_P [list, optional] List of methods to consider

pts [int, optional] A list of points to calculate the property at; if Pmin to Pmax covers a wide range of method validities, only a few points may end up calculated for a given method so this may need to be large

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

show [bool] If True, displays the plot; otherwise, returns it

`property_max = 10000.0`

property_min = 0

ranked_methods = []

solve_property(*goal*)

Method to solve for the temperature at which a property is at a specified value. *T_dependent_property* is used to calculate the value of the property as a function of temperature.

Checks the given property value with *test_property_validity* first and raises an exception if it is not valid.

Parameters

goal [float] Propoerty value desired, [*units*]

Returns

T [float] Temperature at which the property is the specified value [K]

test_method_validity(*T*, *method*)

Method to test the validity of a specified method for a given temperature. Demo function for testing only; must be implemented according to the methods available for each individual method. Include the interpolation check here.

Parameters

T [float] Temperature at which to determine the validity of the method, [K]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

classmethod test_property_validity(*prop*)

Method to test the validity of a calculated property. Normally, this method is used by a given property class, and has maximum and minimum limits controlled by the variables *property_min* and *property_max*.

Parameters

prop [float] property to be tested, [*units*]

Returns

validity [bool] Whether or not a specifid method is valid

units = 'Property units'

valid_methods(*T=None*)

Method to obtain a sorted list of methods that have data available to be used. The methods are ranked in the following order:

- The currently selected method is first (if one is selected)
- Other available methods are ranked by the attribute *ranked_methods*

If *T* is provided, the methods will be checked against the temperature limits of the correlations as well.

Parameters

T [float or None] Temperature at which to test methods, [K]

Returns

sorted_valid_methods [list] Sorted lists of methods valid at T according to *test_method_validity*, [-]

valid_methods_P(*T=None*, *P=None*)

Method to obtain a sorted list of high-pressure methods that have data available to be used. The methods are ranked in the following order:

- The currently selected *method_P* is first (if one is selected)
- Other available pressure-depenent methods are ranked by the attribute `ranked_methods_P`

If *T* and *P* are provided, the methods will be checked against the temperature and pressure limits of the correlations as well.

Parameters

T [float] Temperature at which to test methods, [K]

P [float] Pressure at which to test methods, [Pa]

Returns

sorted_valid_methods_P [list] Sorted lists of methods valid at T and P according to `test_method_validity_P`

7.31.3 Temperature, Pressure, and Composition Dependent

class thermo.utils.**MixtureProperty**(***kwargs*)

Bases: `object`

Attributes

correct_pressure_pure Method to set the pressure-dependence of the model; if set to False, only temperature dependence is used, and if True, temperature and pressure dependence are used.

method Method to set the T, P, and composition dependent property method desired.

prop_cached

Methods

`__call__`(T, P[, zs, ws])	Convenience method to calculate the property; calls *mixture_property*.
as_json([references])	Method to create a JSON serialization of the mixture property which can be stored, and reloaded later.
calculate_derivative_P(P, T, zs, ws, method)	Method to calculate a derivative of a mixture property with respect to pressure at constant temperature and composition of a given order using a specified method.
calculate_derivative_T(T, P, zs, ws, method)	Method to calculate a derivative of a mixture property with respect to temperature at constant pressure and composition of a given order using a specified method.
excess_property(T, P[, zs, ws])	Method to calculate the excess property with sanity checking and without specifying a specific method.
from_json(string)	Method to create a MixtureProperty from a JSON serialization of another MixtureProperty.

continues on next page

Table 99 – continued from previous page

mixture_property(T, P[, zs, ws])	Method to calculate the property with sanity checking and without specifying a specific method.
partial_property(T, P, i[, zs, ws])	Method to calculate the partial molar property with sanity checking and without specifying a specific method for the specified compound index and composition.
plot_isobar(P[, zs, ws, Tmin, Tmax, ...])	Method to create a plot of the property vs temperature at a specific pressure and composition according to either a specified list of methods, or the selected method.
plot_isotherm(T[, zs, ws, Pmin, Pmax, ...])	Method to create a plot of the property vs pressure at a specified temperature and composition according to either a specified list of methods, or the set method.
plot_property([zs, ws, Tmin, Tmax, Pmin, ...])	Method to create a plot of the property vs temperature and pressure according to either a specified list of methods, or the selected method.
property_derivative_P(T, P[, zs, ws, order])	Method to calculate a derivative of a mixture property with respect to pressure at constant temperature and composition, of a given order.
property_derivative_T(T, P[, zs, ws, order])	Method to calculate a derivative of a mixture property with respect to temperature at constant pressure and composition, of a given order.
test_property_validity(prop)	Method to test the validity of a calculated property.

pure_objs	
set_poly_fit_coeffs	

RAISE_PROPERTY_CALCULATION_ERROR = False

TP_zs_ws_cached = (None, None, None, None)

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

all_methods
Set of all methods available for a given set of information; filled by `load_all_methods`.

all_poly_fit = False

as_json(*references=1*)
Method to create a JSON serialization of the mixture property which can be stored, and reloaded later.

Parameters

references [int] How to handle references to other objects; internal parameter, [-]

Returns

json_repr [dict] JSON-friendly representation, [-]

calculate_derivative_P(*P*, *T*, *zs*, *ws*, *method*, *order=1*)
Method to calculate a derivative of a mixture property with respect to pressure at constant temperature and composition of a given order using a specified method. Uses SciPy's derivative function, with a delta of 0.01 Pa and a number of points equal to 2*order + 1.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

P [float] Pressure at which to calculate the derivative, [Pa]

T [float] Temperature at which to calculate the derivative, [K]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method for which to find the derivative

order [int] Order of the derivative, >= 1

Returns

d_prop_d_P_at_T [float] Calculated derivative property at constant temperature, [*units/Pa^order*]

`calculate_derivative_T`(*T*, *P*, *zs*, *ws*, *method*, *order=1*)

Method to calculate a derivative of a mixture property with respect to temperature at constant pressure and composition of a given order using a specified method. Uses SciPy's derivative function, with a delta of 1E-6 K and a number of points equal to 2*order + 1.

This method can be overwritten by subclasses who may perfer to add analytical methods for some or all methods as this is much faster.

If the calculation does not succeed, returns the actual error encountered.

Parameters

T [float] Temperature at which to calculate the derivative, [K]

P [float] Pressure at which to calculate the derivative, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method for which to find the derivative

order [int] Order of the derivative, >= 1

Returns

d_prop_d_T_at_P [float] Calculated derivative property at constant pressure, [*units/K^order*]

`property correct_pressure_pure`

Method to set the pressure-dependence of the model; if set to False, only temperature dependence is used, and if True, temperature and pressure dependence are used.

`excess_property`(*T*, *P*, *zs=None*, *ws=None*)

Method to calculate the excess property with sanity checking and without specifying a specific method. This requires the calculation of the property as a function of composition at the limiting concentration of each component. One or both of *zs* and *ws* are required.

$$m^E = m_{mixing} = m - \sum_i m_{i,pure} \cdot z_i$$

Parameters

T [float] Temperature at which to calculate the excess property, [K]

P [float] Pressure at which to calculate the excess property, [Pa]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

Returns

excess_prop [float] Calculated excess property, [*units*]

classmethod from_json(*string*)

Method to create a MixtureProperty from a JSON serialization of another MixtureProperty.

Parameters

json_repr [dict] JSON-friendly representation, [-]

Returns

constants [`MixtureProperty`] Newly created object from the json serialization, [-]

Notes

It is important that the input string be in the same format as that created by `MixtureProperty.as_json`.

property method

Method to set the T, P, and composition dependent property method desired. See the `all_methods` attribute for a list of methods valid for the specified chemicals and inputs.

mixture_property(*T*, *P*, *zs=None*, *ws=None*)

Method to calculate the property with sanity checking and without specifying a specific method. `valid_methods` is used to obtain a sorted list of methods to try. Methods are then tried in order until one succeeds. The methods are allowed to fail, and their results are checked with `test_property_validity`. On success, the used method is stored in the variable `method`.

If `method` is set, this method is first checked for validity with `test_method_validity` for the specified temperature, and if it is valid, it is then used to calculate the property. The result is checked for validity, and returned if it is valid. If either of the checks fail, the function retrieves a full list of valid methods with `valid_methods` and attempts them as described above.

If no methods are found which succeed, returns None. One or both of *zs* and *ws* are required.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

Returns

prop [float] Calculated property, [*units*]

name = 'Test'

partial_property(*T*, *P*, *i*, *zs=None*, *ws=None*)
Method to calculate the partial molar property with sanity checking and without specifying a specific method for the specified compound index and composition.

$$\bar{m}_i = \left(\frac{\partial (n_T m)}{\partial n_i}\right)_{T,P,n_{j\ne i}}$$

Parameters

T [float] Temperature at which to calculate the partial property, [K]

P [float] Pressure at which to calculate the partial property, [Pa]

i [int] Compound index, [-]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

Returns

partial_prop [float] Calculated partial property, [*units*]

plot_isobar(*P*, *zs=None*, *ws=None*, *Tmin=None*, *Tmax=None*, *methods=[]*, *pts=50*, *only_valid=True*)
Method to create a plot of the property vs temperature at a specific pressure and composition according to either a specified list of methods, or the selected method. User-selectable number of points, and temperature range. If only_valid is set,:obj:*test_method_validity* will be used to check if each condition in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the method fails. One or both of *zs* and *ws* are required.

Parameters

P [float] Pressure for the isobar, [Pa]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

Tmin [float] Minimum temperature, to begin calculating the property, [K]

Tmax [float] Maximum temperature, to stop calculating the property, [K]

methods [list, optional] List of methods to consider

pts [int, optional] A list of points to calculate the property at; if Tmin to Tmax covers a wide range of method validities, only a few points may end up calculated for a given method so this may need to be large

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

plot_isotherm(*T*, *zs=None*, *ws=None*, *Pmin=None*, *Pmax=None*, *methods=[]*, *pts=50*, *only_valid=True*)
Method to create a plot of the property vs pressure at a specified temperature and composition according to either a specified list of methods, or the set method. User-selectable number of points, and pressure range. If only_valid is set, `test_method_validity` will be used to check if each condition in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the method fails. One or both of *zs* and *ws* are required.

Parameters

T [float] Temperature at which to create the plot, [K]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

Pmin [float] Minimum pressure, to begin calculating the property, [Pa]

Pmax [float] Maximum pressure, to stop calculating the property, [Pa]

methods [list, optional] List of methods to consider

pts [int, optional] A list of points to calculate the property at; if Pmin to Pmax covers a wide range of method validities, only a few points may end up calculated for a given method so this may need to be large

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

`plot_property`(*zs=None*, *ws=None*, *Tmin=None*, *Tmax=None*, *Pmin=100000.0*, *Pmax=1000000.0*, *methods=[]*, *pts=15*, *only_valid=True*)

Method to create a plot of the property vs temperature and pressure according to either a specified list of methods, or the selected method. User-selectable number of points for each variable. If only_valid is set,:obj:*test_method_validity* will be used to check if each condition in the specified range is valid, and `test_property_validity` will be used to test the answer, and the method is allowed to fail; only the valid points will be plotted. Otherwise, the result will be calculated and displayed as-is. This will not suceed if the any method fails for any point. One or both of *zs* and *ws* are required.

Parameters

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

Tmin [float] Minimum temperature, to begin calculating the property, [K]

Tmax [float] Maximum temperature, to stop calculating the property, [K]

Pmin [float] Minimum pressure, to begin calculating the property, [Pa]

Pmax [float] Maximum pressure, to stop calculating the property, [Pa]

methods [list, optional] List of methods to consider

pts [int, optional] A list of points to calculate the property at for both temperature and pressure; pts^2 points will be calculated.

only_valid [bool] If True, only plot successful methods and calculated properties, and handle errors; if False, attempt calculation without any checking and use methods outside their bounds

`prop_cached = None`

`property_derivative_P`(*T*, *P*, *zs=None*, *ws=None*, *order=1*)

Method to calculate a derivative of a mixture property with respect to pressure at constant temperature and composition, of a given order. Methods found valid by `valid_methods` are attempted until a method succeeds. If no methods are valid and succeed, None is returned.

Calls `calculate_derivative_P` internally to perform the actual calculation.

$$\text{derivative} = \frac{d(\text{property})}{dP}|_{T,z}$$

Parameters

T [float] Temperature at which to calculate the derivative, [K]

P [float] Pressure at which to calculate the derivative, [Pa]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

order [int] Order of the derivative, >= 1

Returns

d_prop_d_P_at_T [float] Calculated derivative property, [*units/Pa^order*]

property_derivative_T(*T*, *P*, *zs=None*, *ws=None*, *order=1*)

Method to calculate a derivative of a mixture property with respect to temperature at constant pressure and composition, of a given order. Methods found valid by `valid_methods` are attempted until a method succeeds. If no methods are valid and succeed, None is returned.

Calls `calculate_derivative_T` internally to perform the actual calculation.

$$\text{derivative} = \frac{d(\text{property})}{dT}|_{P,z}$$

One or both of *zs* and *ws* are required.

Parameters

T [float] Temperature at which to calculate the derivative, [K]

P [float] Pressure at which to calculate the derivative, [Pa]

zs [list[float], optional] Mole fractions of all species in the mixture, [-]

ws [list[float], optional] Weight fractions of all species in the mixture, [-]

order [int] Order of the derivative, >= 1

Returns

d_prop_d_T_at_P [float] Calculated derivative property, [*units/K^order*]

property_max = 10.0

property_min = 0.0

pure_objs()

pure_reference_types = ()

pure_references = ()

ranked_methods = []

set_poly_fit_coeffs()

skip_method_validity_check = False

Flag to disable checking the validity of the method at the specified conditions. Saves a little time.

skip_prop_validity_check = False

Flag to disable checking the output of the value. Saves a little time.

classmethod test_property_validity(*prop*)

Method to test the validity of a calculated property. Normally, this method is used by a given property class, and has maximum and minimum limits controlled by the variables `property_min` and `property_max`.

Parameters

prop [float] property to be tested, [*units*]

Returns

validity [bool] Whether or not a specifid method is valid

units = 'test units'

7.32 Vapor Pressure and Sublimation Pressure (thermo.vapor_pressure)

This module contains implementations of *thermo.utils.TDependentProperty* representing vapor pressure and sublimation pressure. A variety of estimation and data methods are available as included in the *chemicals* library.

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

7.32.1 Vapor Pressure

class thermo.vapor_pressure.**VaporPressure**(*Tb=None*, *Tc=None*, *Pc=None*, *omega=None*, *CASRN=''*, *eos=None*, *extrapolation='AntoineAB|DIPPR101_ABC'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with vapor pressure as a function of temperature. Consists of five coefficient-based methods and four data sources, one source of tabular information, four corresponding-states estimators, any provided equation of state, the external library CoolProp, and one substance-specific formulation.

Parameters

Tb [float, optional] Boiling point, [K]

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

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

omega [float, optional] Acentric factor, [-]

CASRN [str, optional] The CAS number of the chemical

eos [object, optional] Equation of State object after *thermo.eos.GCEOS*

load_data [bool, optional] If False, do not load property coefficients from data sources in files; this can be used to reduce the memory consumption of an object as well, [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

chemicals.vapor_pressure.Wagner_original

chemicals.vapor_pressure.Wagner

chemicals.vapor_pressure.TRC_Antoine_extended

chemicals.vapor_pressure.Antoine

chemicals.vapor_pressure.boiling_critical_relation

chemicals.vapor_pressure.Lee_Kesler

chemicals.vapor_pressure.Ambrose_Walton

chemicals.vapor_pressure.Sanjari

chemicals.vapor_pressure.Edalat

chemicals.iapws.iapws95_Psat

Notes

To iterate over all methods, use the list stored in *vapor_pressure_methods*.

WAGNER_MCGARRY: The Wagner 3,6 original model equation documented in chemicals.vapor_pressure.Wagner_original, with data for 245 chemicals, from [1],

WAGNER_POLING: The Wagner 2.5, 5 model equation documented in chemicals.vapor_pressure.Wagner in [2], with data for 104 chemicals.

ANTOINE_EXTENDED_POLING: The TRC extended Antoine model equation documented in chemicals.vapor_pressure.TRC_Antoine_extended with data for 97 chemicals in [2].

ANTOINE_POLING: Standard Antoine equation, as documented in the function chemicals.vapor_pressure.Antoine and with data for 325 fluids from [2]. Coefficients were altered to be in units of Pa and Kelvin.

ANTOINE_WEBBOOK: Standard Antoine equation, as documented in the function chemicals.vapor_pressure.Antoine and with data for ~1400 fluids from [6]. Coefficients were altered to be in units of Pa and Kelvin.

DIPPR_PERRY_8E: A collection of 341 coefficient sets from the DIPPR database published openly in [5]. Provides temperature limits for all its fluids. chemicals.dippr.EQ101 is used for its fluids.

VDI_PPDS: Coefficients for a equation form developed by the PPDS, published openly in [4].

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [3]. Very slow.

BOILING_CRITICAL: Fundamental relationship in thermodynamics making several approximations; see chemicals.vapor_pressure.boiling_critical_relation for details. Least accurate method in most circumstances.

LEE_KESLER_PSAT: CSP method documented in chemicals.vapor_pressure.Lee_Kesler. Widely used.

AMBROSE_WALTON: CSP method documented in chemicals.vapor_pressure.Ambrose_Walton.

SANJARI: CSP method documented in chemicals.vapor_pressure.Sanjari.

EDALAT: CSP method documented in chemicals.vapor_pressure.Edalat.

VDI_TABULAR: Tabular data in [4] along the saturation curve; interpolation is as set by the user or the default.

EOS: Equation of state provided by user; must implement *thermo.eos.GCEOS.Psat*

IAPWS: IAPWS-95 formulation documented in chemicals.iapws.iapws95_Psat.

References

[1], [2], [3], [4], [5], [6]

Methods

calculate(T, method)	Method to calculate vapor pressure of a fluid at temperature *T* with a given method.
interpolation_T(T)	Function to make the data-based interpolation as linear as possible.
interpolation_property(P)	log(P) interpolation transformation by default.
interpolation_property_inv(P)	exp(P) interpolation transformation by default; reverses interpolation_property_inv.
test_method_validity(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)

Method to calculate vapor pressure of a fluid at temperature *T* with a given method.

This method has no exception handling; see thermo.utils.TDependentProperty.T_dependent_property for that.

Parameters

T [float] Temperature at calculate vapor pressure, [K]

method [str] Name of the method to use

Returns

Psat [float] Vapor pressure at T, [Pa]

static interpolation_T(*T*)

Function to make the data-based interpolation as linear as possible. This transforms the input *T* into the *1/T* domain.

static interpolation_property(*P*)

log(P) interpolation transformation by default.

static interpolation_property_inv(*P*)

exp(P) interpolation transformation by default; reverses interpolation_property_inv.

name = 'Vapor pressure'

property_max = 10000000000.0

Maximum valid value of vapor pressure. Set slightly above the critical point estimated for Iridium; Mercury's 160 MPa critical point is the highest known.

property_min = 0

Mimimum valid value of vapor pressure.

ranked_methods = ['IAPWS', 'WAGNER_MCGARRY', 'WAGNER_POLING', 'ANTOINE_EXTENDED_POLING', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'COOLPROP', 'ANTOINE_POLING', 'VDI_TABULAR', 'ANTOINE_WEBBOOK', 'AMBROSE_WALTON', 'LEE_KESLER_PSAT', 'EDALAT', 'BOILING_CRITICAL', 'EOS', 'SANJARI']

Default rankings of the available methods.

test_method_validity(*T*, *method*)

Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For CSP methods, the models are considered valid from 0 K to the critical point. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'Pa'

thermo.vapor_pressure.**vapor_pressure_methods = ['IAPWS', 'WAGNER_MCGARRY', 'WAGNER_POLING', 'ANTOINE_EXTENDED_POLING', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'COOLPROP', 'ANTOINE_POLING', 'VDI_TABULAR', 'ANTOINE_WEBBOOK', 'AMBROSE_WALTON', 'LEE_KESLER_PSAT', 'EDALAT', 'EOS', 'BOILING_CRITICAL', 'SANJARI']**

Holds all methods available for the VaporPressure class, for use in iterating over them.

7.32.2 Sublimation Pressure

class thermo.vapor_pressure.**SublimationPressure**(*CASRN=None*, *Tt=None*, *Pt=None*, *Hsub_t=None*, *extrapolation='linear'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with sublimation pressure as a function of temperature. Consists of one estimation method.

Parameters

CASRN [str, optional] The CAS number of the chemical

Tt [float, optional] Triple temperature, [K]

Pt [float, optional] Triple pressure, [Pa]

Hsub_t [float, optional] Sublimation enthalpy at the triple point, [J/mol]

load_data [bool, optional] If False, do not load property coefficients from data sources in files; this can be used to reduce the memory consumption of an object as well, [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.vapor_pressure.Psub_Clapeyron`

Notes

To iterate over all methods, use the list stored in *sublimation_pressure_methods*.

PSUB_CLAPEYRON: Clapeyron thermodynamic identity, Psub_Clapeyron

References

[1]

Methods

calculate(T, method)	Method to calculate sublimation pressure of a fluid at temperature T with a given method.
interpolation_T(T)	Function to make the data-based interpolation as linear as possible.
interpolation_property(P)	log(P) interpolation transformation by default.
interpolation_property_inv(P)	exp(P) interpolation transformation by default; reverses *interpolation_property_inv*.
test_method_validity(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)
Method to calculate sublimation pressure of a fluid at temperature T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature at calculate sublimation pressure, [K]

method [str] Name of the method to use

Returns

Psub [float] Sublimation pressure at T, [Pa]

static interpolation_T(*T*)
Function to make the data-based interpolation as linear as possible. This transforms the input T into the *1/T* domain.

static interpolation_property(*P*)
log(P) interpolation transformation by default.

static interpolation_property_inv(*P*)
exp(P) interpolation transformation by default; reverses *interpolation_property_inv*.

name = 'Sublimation pressure'

property_max = 100000.0
Maximum valid value of sublimation pressure. Set to 1 bar tentatively.

property_min = 1e-300
Mimimum valid value of sublimation pressure.

ranked_methods = ['PSUB_CLAPEYRON']
Default rankings of the available methods.

test_method_validity(*T*, *method*)

Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For CSP methods, the models are considered valid from 0 K to the critical point. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'Pa'

thermo.vapor_pressure.**sublimation_pressure_methods = ['PSUB_CLAPEYRON']**

Holds all methods available for the SublimationPressure class, for use in iterating over them.

7.33 Viscosity (thermo.viscosity)

This module contains implementations of `TPDependentProperty` representing liquid and vapor viscosity. A variety of estimation and data methods are available as included in the *chemicals* library. Additionally liquid and vapor mixture viscosity predictor objects are implemented subclassing `MixtureProperty`.

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

7.33.1 Pure Liquid Viscosity

class thermo.viscosity.**ViscosityLiquid**(*CASRN=''*, *MW=None*, *Tm=None*, *Tc=None*, *Pc=None*, *Vc=None*, *omega=None*, *Psat=None*, *Vml=None*, *extrapolation='linear'*, *extrapolation_min=1e-05*, ***kwargs*)

Bases: `thermo.utils.tp_dependent_property.TPDependentProperty`

Class for dealing with liquid viscosity as a function of temperature and pressure.

For low-pressure (at 1 atm while under the vapor pressure; along the saturation line otherwise) liquids, there are six coefficient-based methods from three data sources, one source of tabular information, two corresponding-states estimators, one group contribution method, and the external library CoolProp.

For high-pressure liquids (also, <1 atm liquids), there is one corresponding-states estimator, and the external library CoolProp.

Parameters

CASRN [str, optional] The CAS number of the chemical

MW [float, optional] Molecular weight, [g/mol]

Tm [float, optional] Melting point, [K]

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

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

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

omega [float, optional] Acentric factor, [-]

Psat [float or callable, optional] Vapor pressure at a given temperature or callable for the same, [Pa]

Vml [float or callable, optional] Liquid molar volume at a given temperature and pressure or callable for the same, [m^3/mol]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see `TDependentProperty` for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.viscosity.Viswanath_Natarajan_3`

`chemicals.viscosity.Viswanath_Natarajan_2`

`chemicals.viscosity.Viswanath_Natarajan_2_exponential`

`chemicals.viscosity.Letsou_Stiel`

`chemicals.viscosity.Przedziecki_Sridhar`

`chemicals.viscosity.Lucas`

`thermo.joback.Joback`

Notes

To iterate over all methods, use the lists stored in `viscosity_liquid_methods` and `viscosity_liquid_methods_P` for low and high pressure methods respectively.

Low pressure methods:

DUTT_PRASAD: A simple function as expressed in [1], with data available for 100 fluids. Temperature limits are available for all fluids. See `chemicals.viscosity.Viswanath_Natarajan_3` for details.

VISWANATH_NATARAJAN_3: A simple function as expressed in [1], with data available for 432 fluids. Temperature limits are available for all fluids. See `chemicals.viscosity.Viswanath_Natarajan_3` for details.

VISWANATH_NATARAJAN_2: A simple function as expressed in [1], with data available for 135 fluids. Temperature limits are available for all fluids. See `chemicals.viscosity.Viswanath_Natarajan_2` for details.

VISWANATH_NATARAJAN_2E: A simple function as expressed in [1], with data available for 14 fluids. Temperature limits are available for all fluids. See `chemicals.viscosity.Viswanath_Natarajan_2_exponential` for details.

DIPPR_PERRY_8E: A collection of 337 coefficient sets from the DIPPR database published openly in [4]. Provides temperature limits for all its fluids. `EQ101` is used for its fluids.

LETSOU_STIEL: CSP method, described in `chemicals.viscosity.Letsou_Stiel`.

PRZEDZIECKI_SRIDHAR: CSP method, described in `chemicals.viscosity.Przedziecki_Sridhar`.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [2]. Very slow.

VDI_TABULAR: Tabular data in [3] along the saturation curve; interpolation is as set by the user or the default.

VDI_PPDS: Coefficients for a equation form developed by the PPDS, published openly in [3]. 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.

JOBACK: An estimation method for organic substances in [5]; this also requires molecular weight as an input.

High pressure methods:

LUCAS: CSP method, described in `chemicals.viscosity.Lucas`. Calculates a low-pressure liquid viscosity as its input.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [2]. Very slow, but unparalled in accuracy for pressure dependence.

A minimum viscosity value of 1e-5 Pa*s is set according to [4]. This is also just above the lowest experimental values of viscosity of helium, 9.4e-6 Pa*s. This excludes the behavior of superfluids, and also systems where the mean free path between moleules approaches the geometry of the system and then the viscosity is geometry-dependent.

References

[1], [2], [3], [4], [5], [6]

Attributes

`Tmax` Maximum temperature (K) at which the current method can calculate the property.

`Tmin` Minimum temperature (K) at which the current method can calculate the property.

Methods

`calculate`(T, method)	Method to calculate low-pressure liquid viscosity at tempearture T with a given method.
`calculate_P`(T, P, method)	Method to calculate pressure-dependent liquid viscosity at temperature T and pressure P with a given method.
`test_method_validity`(T, method)	Method to check the validity of a method.
`test_method_validity_P`(T, P, method)	Method to check the validity of a high-pressure method.

property Tmax

Maximum temperature (K) at which the current method can calculate the property.

property Tmin
Minimum temperature (K) at which the current method can calculate the property.

calculate(*T*, *method*)
Method to calculate low-pressure liquid viscosity at tempearture *T* with a given method.

This method has no exception handling; see `T_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate viscosity, [K]

method [str] Name of the method to use

Returns

mu [float] Viscosity of the liquid at T and a low pressure, [Pa*s]

calculate_P(*T*, *P*, *method*)
Method to calculate pressure-dependent liquid viscosity at temperature *T* and pressure *P* with a given method.

This method has no exception handling; see `TP_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate viscosity, [K]

P [float] Pressure at which to calculate viscosity, [K]

method [str] Name of the method to use

Returns

mu [float] Viscosity of the liquid at T and P, [Pa*s]

name = 'liquid viscosity'

property_max = 200000000.0
Maximum valid value of liquid viscosity. Generous limit, as the value is that of bitumen in a Pitch drop experiment.

property_min = 0.0
Mimimum valid value of liquid viscosity.

ranked_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'DUTT_PRASAD', 'VISWANATH_NATARAJAN_3', 'VISWANATH_NATARAJAN_2', 'VISWANATH_NATARAJAN_2E', 'VDI_TABULAR', 'LETSOU_STIEL', 'JOBACK', 'PRZEDZIECKI_SRIDHAR']
Default rankings of the low-pressure methods.

ranked_methods_P = ['COOLPROP', 'LUCAS']
Default rankings of the high-pressure methods.

test_method_validity(*T*, *method*)
Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For CSP methods, the models are considered valid from 0 K to the critical point. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`test_method_validity_P`(*T*, *P*, *method*)

Method to check the validity of a high-pressure method. For **COOLPROP**, the fluid must be both a liquid and under the maximum pressure of the fluid's EOS. **LUCAS** doesn't work on some occasions, due to something related to Tr and negative powers - but is otherwise considered correct for all circumstances.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures and pressures.

Parameters

T [float] Temperature at which to test the method, [K]

P [float] Pressure at which to test the method, [Pa]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`units = 'Pa*s'`

`thermo.viscosity.viscosity_liquid_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'DUTT_PRASAD', 'VISWANATH_NATARAJAN_3', 'VISWANATH_NATARAJAN_2', 'VISWANATH_NATARAJAN_2E', 'VDI_TABULAR', 'LETSOU_STIEL', 'JOBACK', 'PRZEDZIECKI_SRIDHAR']`

Holds all low-pressure methods available for the ViscosityLiquid class, for use in iterating over them.

`thermo.viscosity.viscosity_liquid_methods_P = ['COOLPROP', 'LUCAS']`

Holds all high-pressure methods available for the ViscosityLiquid class, for use in iterating over them.

7.33.2 Pure Gas Viscosity

class `thermo.viscosity.ViscosityGas`(*CASRN=''*, *MW=None*, *Tc=None*, *Pc=None*, *Zc=None*, *dipole=None*, *Vmg=None*, *extrapolation='linear'*, *extrapolation_min=1e-05*, ***kwargs*)

Bases: *thermo.utils.tp_dependent_property.TPDependentProperty*

Class for dealing with gas viscosity as a function of temperature and pressure.

For gases at atmospheric pressure, there are 4 corresponding-states estimators, two sources of coefficient-based models, one source of tabular information, and the external library CoolProp.

For gases under the fluid's boiling point (at sub-atmospheric pressures), and high-pressure gases above the boiling point, there are zero corresponding-states estimators, and the external library CoolProp.

Parameters

CASRN [str, optional] The CAS number of the chemical

MW [float, optional] Molecular weight, [g/mol]

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

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

Zc [float, optional] Critical compressibility, [-]

dipole [float, optional] Dipole moment of the fluid, [debye]

Vmg [float, optional] Molar volume of the fluid at a pressure and temperature, [m^3/mol]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.viscosity.Gharagheizi_gas_viscosity`

`chemicals.viscosity.Yoon_Thodos`

`chemicals.viscosity.Stiel_Thodos`

`chemicals.viscosity.Lucas_gas`

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the lists stored in *viscosity_gas_methods* and *viscosity_gas_methods_P* for low and high pressure methods respectively.

Low pressure methods:

GHARAGHEIZI: CSP method, described in `chemicals.viscosity.Gharagheizi_gas_viscosity`.

YOON_THODOS: CSP method, described in `chemicals.viscosity.Yoon_Thodos`.

STIEL_THODOS: CSP method, described in `chemicals.viscosity.Stiel_Thodos`.

LUCAS_GAS: CSP method, described in `chemicals.viscosity.Lucas_gas`.

DIPPR_PERRY_8E: A collection of 345 coefficient sets from the DIPPR database published openly in [3]. Provides temperature limits for all its fluids. `chemicals.dippr.EQ102` is used for its fluids.

VDI_PPDS: Coefficients for a equation form developed by the PPDS, published openly in [2]. Provides no temperature limits, but provides reasonable values at fairly high and very low temperatures.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [1]. Very slow.

VDI_TABULAR: Tabular data in [2] along the saturation curve; interpolation is as set by the user or the default.

High pressure methods:

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [1]. Very slow, but unparalled in accuracy for pressure dependence.

A minimum viscosity value of 1e-5 Pa*s is set according to [4]. This is also just above the lowest experimental values of viscosity of helium, 9.4e-6 Pa*s.

References

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

Attributes

Tmax Maximum temperature (K) at which the current method can calculate the property.

Tmin Minimum temperature (K) at which the current method can calculate the property.

Methods

`calculate`(T, method)	Method to calculate low-pressure gas viscosity at tempearture T with a given method.
`calculate_P`(T, P, method)	Method to calculate pressure-dependent gas viscosity at temperature T and pressure P with a given method.
`test_method_validity`(T, method)	Method to check the validity of a temperature-dependent low-pressure method.
`test_method_validity_P`(T, P, method)	Method to check the validity of a high-pressure method.

property Tmax
Maximum temperature (K) at which the current method can calculate the property.

property Tmin
Minimum temperature (K) at which the current method can calculate the property.

calculate(*T*, *method*)
Method to calculate low-pressure gas viscosity at tempearture T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature of the gas, [K]

method [str] Name of the method to use

Returns

mu [float] Viscosity of the gas at T and a low pressure, [Pa*s]

calculate_P(*T*, *P*, *method*)
Method to calculate pressure-dependent gas viscosity at temperature T and pressure P with a given method.

This method has no exception handling; see *TP_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate gas viscosity, [K]

P [float] Pressure at which to calculate gas viscosity, [K]

method [str] Name of the method to use

Returns

mu [float] Viscosity of the gas at T and P, [Pa*]

name = 'Gas viscosity'

property_max = 0.001
Maximum valid value of gas viscosity. Might be too high, or too low.

property_min = 0.0
Mimimum valid value of gas viscosity; limiting condition at low pressure is 0.

ranked_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'VDI_TABULAR', 'GHARAGHEIZI', 'YOON_THODOS', 'STIEL_THODOS', 'LUCAS_GAS']
Default rankings of the low-pressure methods.

ranked_methods_P = ['COOLPROP']
Default rankings of the high-pressure methods.

test_method_validity(*T*, *method*)
Method to check the validity of a temperature-dependent low-pressure method. For CSP most methods, the all methods are considered valid from 0 K up to 5000 K. For method **GHARAGHEIZI**, the method is considered valud from 20 K to 2000 K.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

test_method_validity_P(*T*, *P*, *method*)
Method to check the validity of a high-pressure method. For **COOLPROP**, the fluid must be both a gas and under the maximum pressure of the fluid's EOS. No other methods are implemented.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures and pressures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

P [float] Pressure at which to test the method, [Pa]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'Pa*s'

thermo.viscosity.**viscosity_gas_methods = ['COOLPROP', 'DIPPR_PERRY_8E', 'VDI_PPDS', 'VDI_TABULAR', 'GHARAGHEIZI', 'YOON_THODOS', 'STIEL_THODOS', 'LUCAS_GAS']**
Holds all low-pressure methods available for the ViscosityGas class, for use in iterating over them.

thermo.viscosity.**viscosity_gas_methods_P = ['COOLPROP']**
Holds all high-pressure methods available for the ViscosityGas class, for use in iterating over them.

7.33.3 Mixture Liquid Viscosity

class thermo.viscosity.**ViscosityLiquidMixture**(*CASs=[]*, *ViscosityLiquids=[]*, *MWs=[]*, ***kwargs*)

Bases: *thermo.utils.mixture_property.MixtureProperty*

Class for dealing with the viscosity of a liquid mixture as a function of temperature, pressure, and composition. Consists of one electrolyte-specific method, and logarithmic rules based on either mole fractions of mass fractions.

Prefered method is `mixing_logarithmic` with mole fractions, or **Laliberte** if the mixture is aqueous and has electrolytes.

Parameters

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

ViscosityLiquids [list[ViscosityLiquid], optional] ViscosityLiquid objects created for all species in the mixture, [-]

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

correct_pressure_pure [bool, optional] Whether to try to use the better pressure-corrected pure component models or to use only the T-only dependent pure species models, [-]

See also:

thermo.electrochem.Laliberte_viscosity

Notes

To iterate over all methods, use the list stored in *viscosity_liquid_mixture_methods*.

LALIBERTE_MU: Electrolyte model equation with coefficients; see *thermo.electrochem.Laliberte_viscosity* for more details.

MIXING_LOG_MOLAR: Logarithmic mole fraction mixing rule described in `chemicals.utils.mixing_logarithmic`.

MIXING_LOG_MASS: Logarithmic mole fraction mixing rule described in `chemicals.utils.mixing_logarithmic`.

LINEAR: Linear mole fraction mixing rule described in `mixing_simple`.

References

[1]

Methods

calculate(T, P, zs, ws, method)	Method to calculate viscosity of a liquid mixture at temperature T, pressure P, mole fractions zs and weight fractions ws with a given method.
test_method_validity(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax

Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate viscosity of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see *mixture_property* for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

mu [float] Viscosity of the liquid mixture, [Pa*s]

name = 'liquid viscosity'

property_max = 200000000.0
Maximum valid value of liquid viscosity. Generous limit, as the value is that of bitumen in a Pitch drop experiment.

property_min = 0
Mimimum valid value of liquid viscosity.

ranked_methods = ['Laliberte', 'Logarithmic mixing, molar', 'Logarithmic mixing, mass', 'LINEAR']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. If **Laliberte** is applicable, all other methods are returned as inapplicable. Otherwise, there are no checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'Pa*s'

thermo.viscosity.**viscosity_liquid_mixture_methods = ['Laliberte', 'Logarithmic mixing, molar', 'Logarithmic mixing, mass', 'LINEAR']**
Holds all mixing rules available for the *ViscosityLiquidMixture* class, for use in iterating over them.

7.33.4 Mixture Gas Viscosity

class thermo.viscosity.**ViscosityGasMixture**(*MWs=[]*, *molecular_diameters=[]*, *Stockmayers=[]*, *CASs=[]*, *ViscosityGases=[]*, ***kwargs*)

Bases: *thermo.utils.mixture_property.MixtureProperty*

Class for dealing with the viscosity of a gas mixture as a function of temperature, pressure, and composition. Consists of three gas viscosity specific mixing rules and a mole-weighted simple mixing rule.

Prefered method is `Brokaw`.

Parameters

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

molecular_diameters [list[float], optional] Lennard-Jones molecular diameters, [angstrom]

Stockmayers [list[float], optional] Lennard-Jones depth of potential-energy minimum over k or epsilon_k, [K]

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

ViscosityGases [list[ViscosityGas], optional] ViscosityGas objects created for all species in the mixture, [-]

correct_pressure_pure [bool, optional] Whether to try to use the better pressure-corrected pure component models or to use only the T-only dependent pure species models, [-]

See also:

chemicals.viscosity.Brokaw

chemicals.viscosity.Herning_Zipperer

chemicals.viscosity.Wilke

Notes

To iterate over all methods, use the list stored in *viscosity_liquid_mixture_methods*.

BROKAW: Mixing rule described in `Brokaw`.

HERNING_ZIPPERER: Mixing rule described in `Herning_Zipperer`.

WILKE: Mixing rule described in `Wilke`.

LINEAR: Mixing rule described in `mixing_simple`.

References

[1]

Methods

`calculate`(T, P, zs, ws, method)	Method to calculate viscosity of a gas mixture at temperature T, pressure P, mole fractions zs and weight fractions ws with a given method.
`test_method_validity`(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate viscosity of a gas mixture at temperature T, pressure P, mole fractions zs and weight fractions ws with a given method.

This method has no exception handling; see `mixture_property` for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

mu [float] Viscosity of gas mixture, [Pa*s]

name = 'gas viscosity'

property_max = 0.001
Maximum valid value of gas viscosity. Might be too high, or too low.

property_min = 0
Mimimum valid value of gas viscosity; limiting condition at low pressure is 0.

ranked_methods = ['BROKAW', 'HERNING_ZIPPERER', 'LINEAR', 'WILKE']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'Pa*s'

thermo.viscosity.**viscosity_gas_mixture_methods = ['BROKAW', 'HERNING_ZIPPERER', 'WILKE', 'LINEAR']**

Holds all mixing rules available for the `ViscosityGasMixture` class, for use in iterating over them.

7.34 Density/Volume (thermo.volume)

This module contains implementations of `TDependentProperty` representing liquid, vapor, and solid volume. A variety of estimation and data methods are available as included in the *chemicals* library. Additionally liquid, vapor, and solid mixture volume predictor objects are implemented subclassing `MixtureProperty`.

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

7.34.1 Pure Liquid Volume

class thermo.volume.**VolumeLiquid**(*MW=None, Tb=None, Tc=None, Pc=None, Vc=None, Zc=None, omega=None, dipole=None, Psat=None, CASRN='', eos=None, has_hydroxyl=None, extrapolation='constant', **kwargs*)

Bases: `thermo.utils.tp_dependent_property.TPDependentProperty`

Class for dealing with liquid molar volume as a function of temperature and pressure.

For low-pressure (at 1 atm while under the vapor pressure; along the saturation line otherwise) liquids, there are six coefficient-based methods from five data sources, one source of tabular information, one source of constant values, eight corresponding-states estimators, the external library CoolProp and the equation of state.

For high-pressure liquids (also, <1 atm liquids), there is one corresponding-states estimator, and the external library CoolProp.

Parameters

CASRN [str, optional] The CAS number of the chemical

MW [float, optional] Molecular weight, [g/mol]

Tb [float, optional] Boiling point, [K]

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, [-]

dipole [float, optional] Dipole, [debye]

Psat [float or callable, optional] Vapor pressure at a given temperature, or callable for the same [Pa]

eos [object, optional] Equation of State object after *thermo.eos.GCEOS*

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

chemicals.volume.Yen_Woods_saturation

chemicals.volume.Rackett

chemicals.volume.Yamada_Gunn

chemicals.volume.Townsend_Hales

chemicals.volume.Bhirud_normal

chemicals.volume.COSTALD

chemicals.volume.Campbell_Thodos

chemicals.volume.SNM0

chemicals.volume.CRC_inorganic

chemicals.volume.COSTALD_compressed

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the lists stored in *volume_liquid_methods* and *volume_liquid_methods_P* for low and high pressure methods respectively.

Low pressure methods:

DIPPR_PERRY_8E: A simple polynomial as expressed in [1], with data available for 344 fluids. Temperature limits are available for all fluids. Believed very accurate.

VDI_PPDS: Coefficients for a equation form developed by the PPDS (EQ116 in terms of mass density), published openly in [3]. Valid up to the critical temperature, and extrapolates to very low temperatures well.

MMSNM0FIT: Uses a fit coefficient for better accuracy in the SNM0 method, Coefficients available for 73 fluids from [2]. Valid to the critical point.

HTCOSTALDFIT: A method with two fit coefficients to the COSTALD method. Coefficients available for 192 fluids, from [3]. Valid to the critical point.

RACKETTFIT: The Rackett method, with a fit coefficient Z_RA. Data is available for 186 fluids, from [3]. Valid to the critical point.

CRC_INORG_L: Single-temperature coefficient linear model in terms of mass density for the density of inorganic liquids; converted to molar units internally. Data is available for 177 fluids normally valid over a narrow range above the melting point, from [4]; described in CRC_inorganic.

MMSNM0: CSP method, described in `SNM0`.

HTCOSTALD: CSP method, described in `COSTALD`.

YEN_WOODS_SAT: CSP method, described in `Yen_Woods_saturation`.

RACKETT: CSP method, described in `Rackett`.

YAMADA_GUNN: CSP method, described in `Yamada_Gunn`.

BHIRUD_NORMAL: CSP method, described in `Bhirud_normal`.

TOWNSEND_HALES: CSP method, described in `Townsend_Hales`.

CAMPBELL_THODOS: CSP method, described in `Campbell_Thodos`.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [5]. Very slow.

CRC_INORG_L_CONST: Constant inorganic liquid densities, in [4].

VDI_TABULAR: Tabular data in [6] along the saturation curve; interpolation is as set by the user or the default.

EOS: Equation of state provided by user.

High pressure methods:

COSTALD_COMPRESSED: CSP method, described in `COSTALD_compressed`. Calculates a low-pressure molar volume first, using *`T_dependent_property`*.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [5]. Very slow, but unparalled in accuracy for pressure dependence.

EOS: Equation of state provided by user.

References

[1], [2], [3], [4], [5], [6]

Attributes

`Tmax` Maximum temperature (K) at which the current method can calculate the property.

`Tmin` Minimum temperature (K) at which the current method can calculate the property.

Methods

`calculate`(T, method)	Method to calculate low-pressure liquid molar volume at tempearture T with a given method.
`calculate_P`(T, P, method)	Method to calculate pressure-dependent liquid molar volume at temperature T and pressure P with a given method.
`test_method_validity`(T, method)	Method to check the validity of a method.
`test_method_validity_P`(T, P, method)	Method to check the validity of a high-pressure method.

property Tmax
Maximum temperature (K) at which the current method can calculate the property.

property Tmin
Minimum temperature (K) at which the current method can calculate the property.

`calculate`(*T*, *method*)
Method to calculate low-pressure liquid molar volume at tempearture T with a given method.

This method has no exception handling; see `T_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate molar volume, [K]

method [str] Name of the method to use

Returns

Vm [float] Molar volume of the liquid at T and a low pressure, [m^3/mol]

`calculate_P`(*T*, *P*, *method*)
Method to calculate pressure-dependent liquid molar volume at temperature T and pressure P with a given method.

This method has no exception handling; see `TP_dependent_property` for that.

Parameters

T [float] Temperature at which to calculate molar volume, [K]

P [float] Pressure at which to calculate molar volume, [K]

method [str] Name of the method to use

Returns

Vm [float] Molar volume of the liquid at T and P, [m^3/mol]

`name = 'Liquid molar volume'`

`property_max = 0.002`
Maximum valid value of liquid molar volume. Generous limit.

`property_min = 0`
Mimimum valid value of liquid molar volume. It should normally occur at the triple point, and be well above this.

`ranked_methods = ['DIPPR_PERRY_8E', 'VDI_PPDS', 'COOLPROP', 'MMSNM0FIT', 'VDI_TABULAR', 'HTCOSTALDFIT', 'RACKETTFIT', 'CRC_INORG_L', 'CRC_INORG_L_CONST', 'COMMON_CHEMISTRY', 'MMSNM0', 'HTCOSTALD', 'YEN_WOODS_SAT', 'RACKETT', 'YAMADA_GUNN', 'BHIRUD_NORMAL', 'TOWNSEND_HALES', 'CAMPBELL_THODOS', 'EOS']`
Default rankings of the low-pressure methods.

`ranked_methods_P = ['COOLPROP', 'COSTALD_COMPRESSED', 'EOS']`
Default rankings of the high-pressure methods.

`test_method_validity`(*T*, *method*)
Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For CSP methods, the models are considered valid from 0 K to the critical point. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

BHIRUD_NORMAL behaves poorly at low temperatures and is not used under 0.35Tc. The constant value available for inorganic chemicals, from method **CRC_INORG_L_CONST**, is considered valid for all temperatures.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`test_method_validity_P(T, P, method)`

Method to check the validity of a high-pressure method. For **COOLPROP**, the fluid must be both a liquid and under the maximum pressure of the fluid's EOS. **COSTALD_COMPRESSED** is considered valid for all values of temperature and pressure. However, it very often will not actually work, due to the form of the polynomial in terms of Tr, the result of which is raised to a negative power. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures and pressures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

P [float] Pressure at which to test the method, [Pa]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`units = 'm^3/mol'`

`thermo.volume.volume_liquid_methods = ['DIPPR_PERRY_8E', 'VDI_PPDS', 'COOLPROP', 'MMSNM0FIT', 'VDI_TABULAR', 'HTCOSTALDFIT', 'RACKETTFIT', 'CRC_INORG_L', 'CRC_INORG_L_CONST', 'COMMON_CHEMISTRY', 'MMSNM0', 'HTCOSTALD', 'YEN_WOODS_SAT', 'RACKETT', 'YAMADA_GUNN', 'BHIRUD_NORMAL', 'TOWNSEND_HALES', 'CAMPBELL_THODOS', 'EOS']`

Holds all low-pressure methods available for the *VolumeLiquid* class, for use in iterating over them.

`thermo.volume.volume_liquid_methods_P = ['COOLPROP', 'COSTALD_COMPRESSED', 'EOS']`

Holds all high-pressure methods available for the *VolumeLiquid* class, for use in iterating over them.

7.34.2 Pure Gas Volume

`class thermo.volume.VolumeGas(CASRN='', MW=None, Tc=None, Pc=None, omega=None, dipole=None, eos=None, extrapolation=None, **kwargs)`

Bases: *thermo.utils.tp_dependent_property.TPDependentProperty*

Class for dealing with gas molar volume as a function of temperature and pressure.

All considered methods are both temperature and pressure dependent. Included are four CSP methods for calculating second virial coefficients, one source of polynomials for calculating second virial coefficients, one equation of state (Peng-Robinson), and the ideal gas law.

Parameters

CASRN [str, optional] The CAS number of the chemical

MW [float, optional] Molecular weight, [g/mol]

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

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

omega [float, optional] Acentric factor, [-]

dipole [float, optional] Dipole, [debye]

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

`chemicals.virial.BVirial_Pitzer_Curl`

`chemicals.virial.BVirial_Abbott`

`chemicals.virial.BVirial_Tsonopoulos`

`chemicals.virial.BVirial_Tsonopoulos_extended`

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the list stored in *volume_gas_methods*.

PR: Peng-Robinson Equation of State. See the appropriate module for more information.

CRC_VIRIAL: Short polynomials, for 105 fluids from [1]. The full expression is:

$$B = \sum_1^4 a_i \left[T_0/298.15 - 1\right]^{i-1}$$

TSONOPOULOS_EXTENDED: CSP method for second virial coefficients, described in `chemicals.virial.BVirial_Tsonopoulos_extended`

TSONOPOULOS: CSP method for second virial coefficients, described in `chemicals.virial.BVirial_Tsonopoulos`

ABBOTT: CSP method for second virial coefficients, described in `chemicals.virial.BVirial_Abbott`. This method is the simplest CSP method implemented.

PITZER_CURL: CSP method for second virial coefficients, described in `chemicals.virial.BVirial_Pitzer_Curl`.

COOLPROP: CoolProp external library; with select fluids from its library. Range is limited to that of the equations of state it uses, as described in [2]. Very slow, but unparalled in accuracy for pressure dependence.

References

[1], [2]

Attributes

Tmax Maximum temperature (K) at which the current method can calculate the property.

Tmin Minimum temperature (K) at which the current method can calculate the property.

Methods

calculate(T, method)	Method to calculate a property with a specified method, with no validity checking or error handling.
calculate_P(T, P, method)	Method to calculate pressure-dependent gas molar volume at temperature T and pressure P with a given method.
test_method_validity(T, method)	Method to test the validity of a specified method for a given temperature.
test_method_validity_P(T, P, method)	Method to check the validity of a pressure and temperature dependent gas molar volume method.

property Tmax
Maximum temperature (K) at which the current method can calculate the property.

property Tmin
Minimum temperature (K) at which the current method can calculate the property.

calculate(*T*, *method*)
Method to calculate a property with a specified method, with no validity checking or error handling. Demo function for testing only; must be implemented according to the methods available for each individual method. Include the interpolation call here.

Parameters

T [float] Temperature at which to calculate the property, [K]

method [str] Method name to use

Returns

prop [float] Calculated property, [*units*]

calculate_P(*T*, *P*, *method*)
Method to calculate pressure-dependent gas molar volume at temperature T and pressure P with a given method.

This method has no exception handling; see *TP_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate molar volume, [K]

P [float] Pressure at which to calculate molar volume, [K]

method [str] Name of the method to use

Returns

Vm [float] Molar volume of the gas at T and P, [m^3/mol]

name = 'Gas molar volume'

property_max = 10000000000.0
Maximum valid value of gas molar volume. Set roughly at an ideal gas at 1 Pa and 2 billion K.

property_min = 0
Mimimum valid value of gas molar volume. It should normally be well above this.

ranked_methods = []
Default rankings of the low-pressure methods.

ranked_methods_P = ['COOLPROP', 'EOS', 'TSONOPOULOS_EXTENDED', 'TSONOPOULOS', 'ABBOTT', 'PITZER_CURL', 'CRC_VIRIAL', 'IDEAL']
Default rankings of the pressure-dependent methods.

test_method_validity(*T*, *method*)
Method to test the validity of a specified method for a given temperature. Demo function for testing only; must be implemented according to the methods available for each individual method. Include the interpolation check here.

Parameters

T [float] Temperature at which to determine the validity of the method, [K]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

test_method_validity_P(*T*, *P*, *method*)
Method to check the validity of a pressure and temperature dependent gas molar volume method. For the four CSP methods that calculate second virial coefficient, the method is considered valid for all temperatures and pressures, with validity checking based on the result only. For **CRC_VIRIAL**, there is no limit but there should be one; at some conditions, a negative volume will result! For **COOLPROP**, the fluid must be both a gas at the given conditions and under the maximum pressure of the fluid's EOS.

For the equation of state **PR**, the determined phase must be a gas. For **IDEAL**, there are no limits.

For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures and pressures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

P [float] Pressure at which to test the method, [Pa]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

units = 'm^3/mol'

thermo.volume.**volume_gas_methods = ['COOLPROP', 'EOS', 'CRC_VIRIAL', 'TSONOPOULOS_EXTENDED', 'TSONOPOULOS', 'ABBOTT', 'PITZER_CURL', 'IDEAL']**
Holds all methods available for the *VolumeGas* class, for use in iterating over them.

7.34.3 Pure Solid Volume

class thermo.volume.**VolumeSolid**(*CASRN=''*, *MW=None*, *Tt=None*, *Vml_Tt=None*, *extrapolation='linear'*, ***kwargs*)

Bases: *thermo.utils.t_dependent_property.TDependentProperty*

Class for dealing with solid molar volume as a function of temperature. Consists of one constant value source, and one simple estimator based on liquid molar volume.

Parameters

CASRN [str, optional] CAS number

MW [float, optional] Molecular weight, [g/mol]

Tt [float, optional] Triple temperature

Vml_Tt [float, optional] Liquid molar volume at the triple point

load_data [bool, optional] If False, do not load property coefficients from data sources in files [-]

extrapolation [str or None] None to not extrapolate; see *TDependentProperty* for a full list of all options, [-]

method [str or None, optional] If specified, use this method by default and do not use the ranked sorting; an exception is raised if this is not a valid method for the provided inputs, [-]

See also:

chemicals.volume.Goodman

Notes

A string holding each method's name is assigned to the following variables in this module, intended as the most convenient way to refer to a method. To iterate over all methods, use the list stored in *volume_solid_methods*.

CRC_INORG_S: Constant values in [1], for 1872 chemicals.

GOODMAN: Simple method using the liquid molar volume. Good up to 0.3*Tt. See Goodman for details.

References

[1]

Methods

calculate(T, method)	Method to calculate the molar volume of a solid at tempearture T with a given method.
test_method_validity(T, method)	Method to check the validity of a method.

calculate(*T*, *method*)

Method to calculate the molar volume of a solid at tempearture T with a given method.

This method has no exception handling; see *T_dependent_property* for that.

Parameters

T [float] Temperature at which to calculate molar volume, [K]

method [str] Name of the method to use

Returns

Vms [float] Molar volume of the solid at T, [m^3/mol]

`name = 'Solid molar volume'`

`property_max = 0.002`
Maximum value of Heat capacity; arbitrarily set to 0.002, as the largest in the data is 0.00136.

`property_min = 0.0`
Molar volume cannot be under 0.

`ranked_methods = ['CRC_INORG_S', 'GOODMAN']`
Default rankings of the available methods.

`test_method_validity`(*T*, *method*)
Method to check the validity of a method. Follows the given ranges for all coefficient-based methods. For tabular data, extrapolation outside of the range is used if `tabular_extrapolation_permitted` is set; if it is, the extrapolation is considered valid for all temperatures.

It is not guaranteed that a method will work or give an accurate prediction simply because this method considers the method valid.

Parameters

T [float] Temperature at which to test the method, [K]

method [str] Name of the method to test

Returns

validity [bool] Whether or not a method is valid

`units = 'm^3/mol'`

`thermo.volume.volume_solid_methods = ['GOODMAN', 'CRC_INORG_S']`
Holds all methods available for the `VolumeSolid` class, for use in iterating over them.

7.34.4 Mixture Liquid Volume

class `thermo.volume.VolumeLiquidMixture`(*MWs=[]*, *Tcs=[]*, *Pcs=[]*, *Vcs=[]*, *Zcs=[]*, *omegas=[]*, *CASs=[]*, *VolumeLiquids=[]*, ***kwargs*)
Bases: `thermo.utils.mixture_property.MixtureProperty`

Class for dealing with the molar volume of a liquid mixture as a function of temperature, pressure, and composition. Consists of one electrolyte-specific method, four corresponding states methods which do not use pure-component volumes, and one mole-weighted averaging method.

Prefered method is **LINEAR**, or **LALIBERTE** if the mixture is aqueous and has electrolytes.

Parameters

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

Tcs [list[float], optional] Critical temperatures of all species in the mixture, [K]

Pcs [list[float], optional] Critical pressures of all species in the mixture, [Pa]

Vcs [list[float], optional] Critical molar volumes of all species in the mixture, [m^3/mol]

Zcs [list[float], optional] Critical compressibility factors of all species in the mixture, [Pa]

omegas [list[float], optional] Accentric factors of all species in the mixture, [-]

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

VolumeLiquids [list[VolumeLiquid], optional] VolumeLiquid objects created for all species in the mixture, [-]

correct_pressure_pure [bool, optional] Whether to try to use the better pressure-corrected pure component models or to use only the T-only dependent pure species models, [-]

Notes

To iterate over all methods, use the list stored in `volume_liquid_mixture_methods`.

LALIBERTE: Aqueous electrolyte model equation with coefficients; see `thermo.electrochem.Laliberte_density` for more details.

COSTALD_MIXTURE: CSP method described in `COSTALD_mixture`.

COSTALD_MIXTURE_FIT: CSP method described in `COSTALD_mixture`, with two mixture composition independent fit coefficients, *Vc* and *omega*.

RACKETT: CSP method described in `Rackett_mixture`.

RACKETT_PARAMETERS: CSP method described in `Rackett_mixture`, but with a mixture independent fit coefficient for compressibility factor for each species.

LINEAR: Linear mole fraction mixing rule described in `mixing_simple`; also known as Amgat's law.

References

[1]

Methods

`calculate`(T, P, zs, ws, method)	Method to calculate molar volume of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
`test_method_validity`(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate molar volume of a liquid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see `mixture_property` for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

Vm [float] Molar volume of the liquid mixture at the given conditions, [m^3/mol]

`name = 'Liquid volume'`

`property_max = 0.002`
Maximum valid value of liquid molar volume. Generous limit.

`property_min = 0`
Mimimum valid value of liquid molar volume. It should normally occur at the triple point, and be well above this.

`ranked_methods = ['LALIBERTE', 'LINEAR', 'COSTALD_MIXTURE_FIT', 'RACKETT_PARAMETERS', 'COSTALD_MIXTURE', 'RACKETT']`

`test_method_validity`(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

`units = 'm^3/mol'`

`thermo.volume.volume_liquid_mixture_methods = ['LALIBERTE', 'LINEAR', 'COSTALD_MIXTURE_FIT', 'RACKETT_PARAMETERS', <function COSTALD>, 'RACKETT']`
Holds all low-pressure methods available for the `VolumeLiquidMixture` class, for use in iterating over them.

7.34.5 Mixture Gas Volume

`class thermo.volume.VolumeGasMixture`(*eos=None*, *CASs=[]*, *VolumeGases=[]*, *MWs=[]*, ***kwargs*)
Bases: `thermo.utils.mixture_property.MixtureProperty`

Class for dealing with the molar volume of a gas mixture as a function of temperature, pressure, and composition. Consists of an equation of state, the ideal gas law, and one mole-weighted averaging method.

Prefered method is **EOS**, or **IDEAL** if critical properties of components are unavailable.

Parameters

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

VolumeGases [list[VolumeGas], optional] VolumeGas objects created for all species in the mixture, [-]

eos [container[EOS Object], optional] Equation of state mixture object, [-]

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

See also:

`chemicals.volume.ideal_gas`

`thermo.eos_mix`

Notes

To iterate over all methods, use the list stored in `volume_gas_mixture_methods`.

EOS: Equation of state mixture object; see `thermo.eos_mix` for more details.

LINEAR: Linear mole fraction mixing rule described in `mixing_simple`; more correct than the ideal gas law.

IDEAL: The ideal gas law.

References

[1]

Methods

`calculate`(T, P, zs, ws, method)	Method to calculate molar volume of a gas mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
`test_method_validity`(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate molar volume of a gas mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see `mixture_property` for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

Vm [float] Molar volume of the gas mixture at the given conditions, [m^3/mol]

name = 'Gas volume'

property_max = 10000000000.0
Maximum valid value of gas molar volume. Set roughly at an ideal gas at 1 Pa and 2 billion K.

property_min = 0.0
Mimimum valid value of gas molar volume. It should normally be well above this.

ranked_methods = ['EOS', 'LINEAR', 'IDEAL', 'LINEAR_MISSING_IDEAL']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

units = 'm^3/mol'

thermo.volume.**volume_gas_mixture_methods = ['EOS', 'LINEAR', 'IDEAL']**
Holds all methods available for the `VolumeGasMixture` class, for use in iterating over them.

7.34.6 Mixture Solid Volume

class thermo.volume.**VolumeSolidMixture**(*CASs=[]*, *VolumeSolids=[]*, *MWs=[]*, ***kwargs*)
Bases: `thermo.utils.mixture_property.MixtureProperty`

Class for dealing with the molar volume of a solid mixture as a function of temperature, pressure, and composition. Consists of only mole-weighted averaging.

Parameters

CASs [list[str], optional] The CAS numbers of all species in the mixture, [-]

VolumeSolids [list[VolumeSolid], optional] VolumeSolid objects created for all species in the mixture, [-]

MWs [list[float], optional] Molecular weights of all species in the mixture, [g/mol]

Notes

To iterate over all methods, use the list stored in *volume_solid_mixture_methods*.

LINEAR: Linear mole fraction mixing rule described in mixing_simple.

Methods

calculate(T, P, zs, ws, method)	Method to calculate molar volume of a solid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.
test_method_validity(T, P, zs, ws, method)	Method to test the validity of a specified method for the given conditions.

Tmax
Maximum temperature at which no method can calculate the property above.

Tmin
Minimum temperature at which no method can calculate the property under.

calculate(*T*, *P*, *zs*, *ws*, *method*)
Method to calculate molar volume of a solid mixture at temperature *T*, pressure *P*, mole fractions *zs* and weight fractions *ws* with a given method.

This method has no exception handling; see *mixture_property* for that.

Parameters

T [float] Temperature at which to calculate the property, [K]

P [float] Pressure at which to calculate the property, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Name of the method to use

Returns

Vm [float] Molar volume of the solid mixture at the given conditions, [m^3/mol]

name = 'Solid molar volume'

property_max = 0.002
Maximum value of Heat capacity; arbitrarily set to 0.002, as the largest in the data is 0.00136.

property_min = 0
Molar volume cannot be under 0.

ranked_methods = ['LINEAR']

test_method_validity(*T*, *P*, *zs*, *ws*, *method*)
Method to test the validity of a specified method for the given conditions. No methods have implemented checks or strict ranges of validity.

Parameters

T [float] Temperature at which to check method validity, [K]

P [float] Pressure at which to check method validity, [Pa]

zs [list[float]] Mole fractions of all species in the mixture, [-]

ws [list[float]] Weight fractions of all species in the mixture, [-]

method [str] Method name to use

Returns

validity [bool] Whether or not a specifid method is valid

`units = 'm^3/mol'`

`thermo.volume.volume_solid_mixture_methods = ['LINEAR']`

Holds all methods available for the *VolumeSolidMixture* class, for use in iterating over them.

7.35 Wilson Gibbs Excess Model (thermo.wilson)

This module contains a class *Wilson* for performing activity coefficient calculations with the Wilson model. An older, functional calculation for activity coefficients only is also present, *Wilson_gammas*.

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

7.35.1 Wilson Class

`class thermo.wilson.Wilson(T, xs, lambda_coeffs=None, ABCDEF=None, lambda_as=None, lambda_bs=None, lambda_cs=None, lambda_ds=None, lambda_es=None, lambda_fs=None)`

Bases: *thermo.activity.GibbsExcess*

Class for representing an a liquid with excess gibbs energy represented by the Wilson equation. This model is capable of representing most nonideal liquids for vapor-liquid equilibria, but is not recommended for liquid-liquid equilibria.

The two basic equations are as follows; all other properties are derived from these.

$$g^E = -RT\sum_i x_i \ln\left(\sum_j x_j \lambda_{i,j} \right)$$

$$\Lambda_{ij} = \exp\left[a_{ij}+\frac{b_{ij}}{T}+c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2} + f_{ij}{T^2}\right]$$

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

lambda_coeffs [list[list[list[float]]], optional] Wilson parameters, indexed by [i][j] and then each value is a 6 element list with parameters (a, b, c, d, e, f); either *lambda_coeffs* or the lambda parameters are required, [various]

ABCDEF [tuple(list[list[float]], 6), optional] The lamba parameters can be provided as a tuple, [various]

lambda_as [list[list[float]], optional] *a* parameters used in calculating `Wilson.lambdas`, [-]

lambda_bs [list[list[float]], optional] *b* parameters used in calculating `Wilson.lambdas`, [K]

lambda_cs [list[list[float]], optional] *c* parameters used in calculating `Wilson.lambdas`, [-]

lambda_ds [list[list[float]], optional] *d* paraemeters used in calculating `Wilson.lambdas`, [1/K]

lambda_es [list[list[float]], optional] *e* parameters used in calculating `Wilson.lambdas`, [K^2]

lambda_fs [list[list[float]], optional] *f* parameters used in calculating `Wilson.lambdas`, [1/K^2]

Notes

In addition to the methods presented here, the methods of its base class `thermo.activity.GibbsExcess` are available as well.

Warning: If parameters are ommited for all interactions, this model reverts to `thermo.activity.IdealSolution`. In large systems it is common to only regress parameters for the most important components; set *lambda* parameters for other components to 0 to "ignore" them and treat them as ideal components.

This class works with python lists, numpy arrays, and can be accelerated with Numba or PyPy quite effectively.

References

[1], [2], [3]

Examples

Example 1

This object-oriented class provides access to many more thermodynamic properties than `Wilson_gammas`, but it can also be used like that function. In the following example, *gammas* are calculated with both functions. The *lambdas* cannot be specified in this class; but fixed values can be converted with the *log* function so that fixed values will be obtained.

```
>>> Wilson_gammas([0.252, 0.748], [[1, 0.154], [0.888, 1]])
[1.881492608717, 1.165577493112]
>>> GE = Wilson(T=300.0, xs=[0.252, 0.748], lambda_as=[[0, log(0.154)], [log(0.888),
↪ 0]])
>>> GE.gammas()
[1.881492608717, 1.165577493112]
```

We can check that the same lambda values were computed as well, and that there is no temperature dependency:

```
>>> GE.lambdas()
[[1.0, 0.154], [0.888, 1.0]]
>>> GE.dlambdas_dT()
[[0.0, 0.0], [0.0, 0.0]]
```

In this case, there is no temperature dependency in the Wilson model as the *lambda* values are fixed, so the excess enthalpy is always zero. Other properties are not always zero.

```
>>> GE.HE(), GE.CpE()
(0.0, 0.0)
>>> GE.GE(), GE.SE(), GE.dGE_dT()
(683.165839398, -2.277219464, 2.2772194646)
```

Example 2

ChemSep is a (partially) free program for modeling distillation. Besides being a wonderful program, it also ships with a permissive license several sets of binary interaction parameters. The Wilson parameters in it can be accessed from Thermo as follows. In the following case, we compute activity coefficients of the ethanol-water system at mole fractions of [.252, 0.748].

```
>>> from thermo.interaction_parameters import IPDB
>>> CAS1, CAS2 = '64-17-5', '7732-18-5'
>>> lambda_as = IPDB.get_ip_asymmetric_matrix(name='ChemSep Wilson', CASs=[CAS1,␣
↪CAS2], ip='aij')
>>> lambda_bs = IPDB.get_ip_asymmetric_matrix(name='ChemSep Wilson', CASs=[CAS1,␣
↪CAS2], ip='bij')
>>> GE = Wilson(T=273.15+70, xs=[.252, .748], lambda_as=lambda_as, lambda_bs=lambda_
↪bs)
>>> GE.gammas()
[1.95733110, 1.1600677]
```

In ChemSep, the form of the Wilson *lambda* equation is

$$\Lambda_{ij} = \frac{V_j}{V_i}\exp\left(\frac{-A_{ij}}{RT}\right)$$

The parameters were converted to the form used by Thermo as follows:

$$a_{ij} = \log\left(\frac{V_j}{V_i}\right)$$

$$b_{ij} = \frac{-A_{ij}}{R} = \frac{-A_{ij}}{1.9872042586408316}$$

This system was chosen because there is also a sample problem for the same components from the DDBST which can be found here: http://chemthermo.ddbst.com/Problems_Solutions/Mathcad_Files/P05.01a%20VLE%20Behavior%20of%20Ethanol%20-%20Water%20Using%20Wilson.xps

In that example, with different data sets and parameters, they obtain at the same conditions activity coefficients of [1.881, 1.165]. Different sources of parameters for the same system will generally have similar behavior if regressed in the same temperature range. As higher order *lambda* parameters are added, models become more likely to behave differently. It is recommended in [3] to regress the minimum number of parameters required.

Example 3

The DDBST has published some sample problems which are fun to work with. Because the DDBST uses a different equation form for the coefficients than this model implements, we must initialize the `Wilson` object with a different method.

```
>>> T = 331.42
>>> N = 3
>>> Vs_ddbst = [74.04, 80.67, 40.73]
>>> as_ddbst = [[0, 375.2835, 31.1208], [-1722.58, 0, -1140.79], [747.217, 3596.17,␣
↪0.0]]
```

(continues on next page)

(continued from previous page)

```
>>> bs_ddbst = [[0, -3.78434, -0.67704], [6.405502, 0, 2.59359], [-0.256645, -6.
→2234, 0]]
>>> cs_ddbst = [[0.0, 7.91073e-3, 8.68371e-4], [-7.47788e-3, 0.0, 3.1e-5], [-1.
→24796e-3, 3e-5, 0.0]]
>>> dis = eis = fis = [[0.0]*N for _ in range(N)]
>>> params = Wilson.from_DDBST_as_matrix(Vs=Vs_ddbst, ais=as_ddbst, bis=bs_ddbst,␣
→cis=cs_ddbst, dis=dis, eis=eis, fis=fis, unit_conversion=False)
>>> xs = [0.229, 0.175, 0.596]
>>> GE = Wilson(T=T, xs=xs, lambda_as=params[0], lambda_bs=params[1], lambda_
→cs=params[2], lambda_ds=params[3], lambda_es=params[4], lambda_fs=params[5])
>>> GE
Wilson(T=331.42, xs=[0.229, 0.175, 0.596], lambda_as=[[0.0, 3.870101271243586, 0.
→07939943395502425], [-6.491263271243587, 0.0, -3.276991837288562], [0.
→8542855660449756, 6.906801837288562, 0.0]], lambda_bs=[[0.0, -375.2835, -31.1208],
→ [1722.58, 0.0, 1140.79], [-747.217, -3596.17, -0.0]], lambda_ds=[[-0.0, -0.
→00791073, -0.000868371], [0.00747788, -0.0, -3.1e-05], [0.00124796, -3e-05, -0.
→0]])
>>> GE.GE(), GE.dGE_dT(), GE.d2GE_dT2()
(480.2639266306882, 4.355962766232997, -0.029130384525017247)
>>> GE.HE(), GE.SE(), GE.dHE_dT(), GE.dSE_dT()
(-963.3892533542517, -4.355962766232997, 9.654392039281216, 0.029130384525017247)
>>> GE.gammas()
[1.2233934334, 1.100945902470, 1.205289928117]
```

The solution given by the DDBST has the same values [1.223, 1.101, 1.205], and can be found here: http://chemthermo.ddbst.com/Problems_Solutions/Mathcad_Files/05.09%20Compare%20Experimental%20VLE%20to%20Wilson%20Equation%20Results.xps

Example 4

A simple example is given in [1]; other textbooks sample problems are normally in the same form as this - with only volumes and the *a* term specified. The system is 2-propanol/water at 353.15 K, and the mole fraction of 2-propanol is 0.25.

```
>>> T = 353.15
>>> N = 2
>>> Vs = [76.92, 18.07] # cm^3/mol
>>> ais = [[0.0, 437.98],[1238.0, 0.0]] # cal/mol
>>> bis = cis = dis = eis = fis = [[0.0]*N for _ in range(N)]
>>> params = Wilson.from_DDBST_as_matrix(Vs=Vs, ais=ais, bis=bis, cis=cis, dis=dis,␣
→eis=eis, fis=fis, unit_conversion=True)
>>> xs = [0.25, 0.75]
>>> GE = Wilson(T=T, xs=xs, lambda_as=params[0], lambda_bs=params[1], lambda_
→cs=params[2], lambda_ds=params[3], lambda_es=params[4], lambda_fs=params[5])
>>> GE.gammas()
[2.124064516, 1.1903745834]
```

The activity coefficients given in [1] are [2.1244, 1.1904]; matching (with a slight deviation from their use of 1.987 as a gas constant).

Attributes

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

model_id [int] Unique identifier for the Wilson activity model, [-]

Methods

GE()	Calculate and return the excess Gibbs energy of a liquid phase represented with the Wilson model.
d2GE_dT2()	Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phase using the Wilson activity coefficient model.
d2GE_dTdxs()	Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy of a liquid represented by the Wilson model.
d2GE_dxixjs()	Calculate and return the second mole fraction derivatives of excess Gibbs energy for the Wilson model.
d2lambdas_dT2()	Calculate and return the second temperature derivative of the *lambda* termsfor the Wilson model at the system temperature.
d3GE_dT3()	Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase using the Wilson activity coefficient model.
d3GE_dxixjxks()	Calculate and return the third mole fraction derivatives of excess Gibbs energy using the Wilson model.
d3lambdas_dT3()	Calculate and return the third temperature derivative of the *lambda* terms for the Wilson model at the system temperature.
dGE_dT()	Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase represented by the Wilson model.
dGE_dxs()	Calculate and return the mole fraction derivatives of excess Gibbs energy for the Wilson model.
dlambdas_dT()	Calculate and return the temperature derivative of the *lambda* terms for the Wilson model at the system temperature.
from_DDBST(Vi, Vj, a, b, c[, d, e, f, ...])	Converts parameters for the wilson equation in the DDBST to the basis used in this implementation.
from_DDBST_as_matrix(Vs[, ais, bis, cis, ...])	Converts parameters for the wilson equation in the DDBST to the basis used in this implementation.
lambdas()	Calculate and return the *lambda* terms for the Wilson model for at system temperature.
to_T_xs(T, xs)	Method to construct a new `Wilson` instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

GE()

Calculate and return the excess Gibbs energy of a liquid phase represented with the Wilson model.

$$g^E = -RT\sum_i x_i \ln\left(\sum_j x_j \lambda_{i,j} \right)$$

Returns

GE [float] Excess Gibbs energy of an ideal liquid, [J/mol]

d2GE_dT2()

Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phase using the Wilson activity coefficient model.

$$\frac{\partial^2 G^E}{\partial T^2} = -R\left[T\sum_i \left(\frac{x_i \sum_j (x_j \frac{\partial^2 \Lambda_{ij}}{\partial T^2})}{\sum_j x_j \Lambda_{ij}} - \frac{x_i (\sum_j x_j \frac{\partial \Lambda_{ij}}{\partial T})^2}{(\sum_j x_j \Lambda_{ij})^2}\right) + 2\sum_i \left(\frac{x_i \sum_j x_j \frac{\partial \Lambda_{ij}}{\partial T}}{\sum_j x_j \Lambda_{ij}}\right)\right]$$

Returns

d2GE_dT2 [float] Second temperature derivative of excess Gibbs energy, [J/(mol*K^2)]

d2GE_dTdxs()

Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy of a liquid represented by the Wilson model.

$$\frac{\partial^2 G^E}{\partial x_k \partial T} = -R\left[T\left(\sum_i \left(\frac{x_i \frac{\partial n_{ik}}{\partial T}}{\sum_j x_j \Lambda_{ij}} - \frac{x_i \Lambda_{ik} (\sum_j x_j \frac{\partial \Lambda_{ij}}{\partial T})}{(\partial_j x_j \Lambda_{ij})^2}\right) + \frac{\sum_i x_i \frac{\partial \Lambda_{ki}}{\partial T}}{\sum_j x_j \Lambda_{kj}}\right) + \ln\left(\sum_i x_i \Lambda_{ki}\right) + \sum_i \frac{x_i \Lambda_{ik}}{\sum_j x_j \Lambda_{ij}}\right]$$

Returns

d2GE_dTdxs [list[float]] Temperature derivative of mole fraction derivatives of excess Gibbs energy, [J/mol/K]

d2GE_dxixjs()

Calculate and return the second mole fraction derivatives of excess Gibbs energy for the Wilson model.

$$\frac{\partial^2 G^E}{\partial x_k \partial x_m} = RT\left(\sum_i \frac{x_i \Lambda_{ik} \Lambda_{im}}{(\sum_j x_j \Lambda_{ij})^2} - \frac{\Lambda_{km}}{\sum_j x_j \Lambda_{kj}} - \frac{\Lambda_{mk}}{\sum_j x_j \Lambda_{mj}}\right)$$

Returns

d2GE_dxixjs [list[list[float]]] Second mole fraction derivatives of excess Gibbs energy, [J/mol]

d2lambdas_dT2()

Calculate and return the second temperature derivative of the *lambda* termsfor the Wilson model at the system temperature.

$$\frac{\partial^2 \Lambda_{ij}}{\partial^2 T} = \left(2f_{ij} + \left(2Tf_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right)^2 - \frac{c_{ij}}{T^2} + \frac{2b_{ij}}{T^3} + \frac{6e_{ij}}{T^4}\right) e^{T^2 f_{ij} + T d_{ij} + a_{ij} + c_{ij}\ln(T) + \frac{b_{ij}}{T} + \frac{e_{ij}}{T^2}}$$

Returns

d2lambdas_dT2 [list[list[float]]] Second temperature deriavtives of Lambda terms, asymmetric matrix, [1/K^2]

Notes

These *Lambda ij* values (and the coefficients) are NOT symmetric.

d3GE_dT3()

Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase using the Wilson activity coefficient model.

$$\frac{\partial^3 G^E}{\partial T^3} = -R\left[3\left(\frac{x_i \sum_j (x_j \frac{\partial^2 \Lambda_{ij}}{\partial T^2})}{\sum_j x_j \Lambda_{ij}} - \frac{x_i (\sum_j x_j \frac{\partial \Lambda_{ij}}{\partial T})^2}{(\sum_j x_j \Lambda_{ij})^2}\right) + T\left(\sum_i \frac{x_i (\sum_j x_j \frac{\partial^3 \Lambda_{ij}}{\partial T^3})}{\sum_j x_j \Lambda_{ij}} - \frac{3x_i (\sum_j x_j \frac{\partial \Lambda_{ij}^2}{\partial T^2})(\sum_j x_j \frac{\partial \Lambda_{ij}}{\partial T})}{(\sum_j x_j \Lambda_{ij})^2} + \right.\right.$$

Returns

d3GE_dT3 [float] Third temperature derivative of excess Gibbs energy, [J/(mol*K^3)]

`d3GE_dxixjxks()`

Calculate and return the third mole fraction derivatives of excess Gibbs energy using the Wilson model.

$$\frac{\partial^3 G^E}{\partial x_k \partial x_m \partial x_n} = -RT\left[\sum_i \left(\frac{2x_i\Lambda_{ik}\Lambda_{im}\Lambda_{in}}{(\sum x_j\Lambda_{ij})^3}\right) - \frac{\Lambda_{km}\Lambda_{kn}}{(\sum_j x_j\Lambda_{kj})^2} - \frac{\Lambda_{mk}\Lambda_{mn}}{(\sum_j x_j\Lambda_{mj})^2} - \frac{\Lambda_{nk}\Lambda_{nm}}{(\sum_j x_j\Lambda_{nj})^2}\right]$$

Returns

d3GE_dxixjxks [list[list[list[float]]]] Third mole fraction derivatives of excess Gibbs energy, [J/mol]

`d3lambdas_dT3()`

Calculate and return the third temperature derivative of the *lambda* terms for the Wilson model at the system temperature.

$$\frac{\partial^3 \Lambda_{ij}}{\partial^3 T} = \left(3\left(2f_{ij} - \frac{c_{ij}}{T^2} + \frac{2b_{ij}}{T^3} + \frac{6e_{ij}}{T^4}\right)\left(2Tf_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right) + \left(2Tf_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right)^3 -$$

Returns

d3lambdas_dT3 [list[list[float]]] Third temperature deriavtives of Lambda terms, asymmetric matrix, [1/K^3]

Notes

These *Lambda ij* values (and the coefficients) are NOT symmetric.

`dGE_dT()`

Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase represented by the Wilson model.

$$\frac{\partial G^E}{\partial T} = -R\sum_i x_i \ln\left(\sum_j x_i \Lambda_{ij}\right) - RT\sum_i \frac{x_i \sum_j x_j \frac{\Lambda_{ij}}{\partial T}}{\sum_j x_j \Lambda_{ij}}$$

Returns

dGE_dT [float] First temperature derivative of excess Gibbs energy of a liquid phase represented by the Wilson model, [J/(mol*K)]

`dGE_dxs()`

Calculate and return the mole fraction derivatives of excess Gibbs energy for the Wilson model.

$$\frac{\partial G^E}{\partial x_k} = -RT\left[\sum_i \frac{x_i\Lambda_{ik}}{\sum_j \Lambda_{ij}x_j} + \ln\left(\sum_j x_j\Lambda_{kj}\right)\right]$$

Returns

dGE_dxs [list[float]] Mole fraction derivatives of excess Gibbs energy, [J/mol]

`dlambdas_dT()`

Calculate and return the temperature derivative of the *lambda* terms for the Wilson model at the system temperature.

$$\frac{\partial \Lambda_{ij}}{\partial T} = \left(2Th_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right)e^{T^2h_{ij} + Td_{ij} + a_{ij} + c_{ij}\ln(T) + \frac{b_{ij}}{T} + \frac{e_{ij}}{T^2}}$$

Returns

dlambdas_dT [list[list[float]]] Temperature deriavtives of Lambda terms, asymmetric matrix [1/K]

Notes

These *Lambda ij* values (and the coefficients) are NOT symmetric.

static from_DDBST(*Vi*, *Vj*, *a*, *b*, *c*, *d=0.0*, *e=0.0*, *f=0.0*, *unit_conversion=True*)

Converts parameters for the wilson equation in the DDBST to the basis used in this implementation.

$$\Lambda_{ij} = \frac{V_j}{V_i}\exp\left(\frac{-\Delta\lambda_{ij}}{RT}\right)$$

$$\Delta\lambda_{ij} = a_{ij} + b_{ij}T + cT^2 + d_{ij}T\ln T + e_{ij}T^3 + f_{ij}/T$$

Parameters

Vi [float] Molar volume of component i; needs only to be in the same units as *Vj*, [cm^3/mol]

Vj [float] Molar volume of component j; needs only to be in the same units as *Vi*, [cm^3/mol]

a [float] *a* parameter in DDBST form, [K]

b [float] *b* parameter in DDBST form, [-]

c [float] *c* parameter in DDBST form, [1/K]

d [float, optional] *d* parameter in DDBST form, [-]

e [float, optional] *e* parameter in DDBST form, [1/K^2]

f [float, optional] *f* parameter in DDBST form, [K^2]

unit_conversion [bool] If True, the input coefficients are in units of cal/K/mol, and a *R* gas constant of 1.9872042... is used for the conversion; the DDBST uses this generally, [-]

Returns

a [float] *a* parameter in `Wilson` form, [-]

b [float] *b* parameter in `Wilson` form, [K]

c [float] *c* parameter in `Wilson` form, [-]

d [float] *d* parameter in `Wilson` form, [1/K]

e [float] *e* parameter in `Wilson` form, [K^2]

f [float] *f* parameter in `Wilson` form, [1/K^2]

Notes

The units show how the different variables are related to each other.

Examples

```
>>> Wilson.from_DDBST(Vi=74.04, Vj=80.67, a=375.2835, b=-3.78434, c=0.00791073,␣
↪d=0.0, e=0.0, f=0.0, unit_conversion=False)
(3.8701012712, -375.2835, -0.0, -0.00791073, -0.0, -0.0)
```

static from_DDBST_as_matrix(*Vs*, *ais=None*, *bis=None*, *cis=None*, *dis=None*, *eis=None*, *fis=None*, *unit_conversion=True*)

Converts parameters for the wilson equation in the DDBST to the basis used in this implementation. Matrix wrapper around *Wilson.from_DDBST*.

Parameters

Vs [list[float]] Molar volume of component; needs only to be in consistent units, [cm^3/mol]

ais [list[list[float]]] a parameters in DDBST form, [K]

bis [list[list[float]]] b parameters in DDBST form, [-]

cis [list[list[float]]] c parameters in DDBST form, [1/K]

dis [list[list[float]], optional] d parameters in DDBST form, [-]

eis [list[list[float]], optional] e parameters in DDBST form, [1/K^2]

fis [list[list[float]], optional] f parameters in DDBST form, [K^2]

unit_conversion [bool] If True, the input coefficients are in units of cal/K/mol, and a R gas constant of 1.9872042… is used for the conversion; the DDBST uses this generally, [-]

Returns

a [list[list[float]]] a parameters in *Wilson* form, [-]

b [list[list[float]]] b parameters in *Wilson* form, [K]

c [list[list[float]]] c parameters in *Wilson* form, [-]

d [list[list[float]]] d paraemeters in *Wilson* form, [1/K]

e [list[list[float]]] e parameters in *Wilson* form, [K^2]

f [list[list[float]]] f parameters in *Wilson* form, [1/K^2]

lambdas()

Calculate and return the *lambda* terms for the Wilson model for at system temperature.

$$\Lambda_{ij} = \exp\left[a_{ij} + \frac{b_{ij}}{T} + c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2} + f_{ij}T^2\right]$$

Returns

lambdas [list[list[float]]] Lambda terms, asymmetric matrix [-]

Notes

These *Lambda ij* values (and the coefficients) are NOT symmetric.

to_T_xs(*T*, *xs*)

Method to construct a new `Wilson` instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions of each component, [-]

Returns

obj [Wilson] New `Wilson` object at the specified conditions [-]

Notes

If the new temperature is the same temperature as the existing temperature, if the *lambda* terms or their derivatives have been calculated, they will be set to the new object as well.

7.35.2 Wilson Functional Calculations

thermo.wilson.**Wilson_gammas**(*xs*, *params*)

Calculates the activity coefficients of each species in a mixture using the Wilson method, given their mole fractions, and dimensionless interaction parameters. Those are normally correlated with temperature, and need to be calculated separately.

$$\ln \gamma_i = 1 - \ln \left(\sum_j^N \Lambda_{ij} x_j\right) - \sum_j^N \frac{\Lambda_{ji} x_j}{\displaystyle\sum_k^N \Lambda_{jk} x_k}$$

Parameters

xs [list[float]] Liquid mole fractions of each species, [-]

params [list[list[float]]] Dimensionless interaction parameters of each compound with each other, [-]

Returns

gammas [list[float]] Activity coefficient for each species in the liquid mixture, [-]

Notes

This model needs N^2 parameters.

The original model correlated the interaction parameters using the standard pure-component molar volumes of each species at 25°C, in the following form:

$$\Lambda_{ij} = \frac{V_j}{V_i} \exp\left(\frac{-\lambda_{i,j}}{RT}\right)$$

If a compound is not liquid at that temperature, the liquid volume is taken at the saturated pressure; and if the component is supercritical, its liquid molar volume should be extrapolated to 25°C.

However, that form has less flexibility and offered no advantage over using only regressed parameters.

Most correlations for the interaction parameters include some of the terms shown in the following form:

$$\ln \Lambda_{ij} = a_{ij} + \frac{b_{ij}}{T} + c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2} + h_{ij}T^2$$

The Wilson model is not applicable to liquid-liquid systems.

For this model to produce ideal acitivty coefficients (gammas = 1), all interaction parameters should be 1.

The specific process simulator implementations are as follows:

References

[1], [2]

Examples

Ethanol-water example, at 343.15 K and 1 MPa, from [2] also posted online http://chemthermo.ddbst.com/Problems_Solutions/Mathcad_Files/P05.01a%20VLE%20Behavior%20of%20Ethanol%20-%20Water%20Using%20Wilson.xps :

```
>>> Wilson_gammas([0.252, 0.748], [[1, 0.154], [0.888, 1]])
[1.881492608717, 1.165577493112]
```

7.35.3 Wilson Regression Calculations

thermo.wilson.**wilson_gammas_binaries**(*xs*, *lambda12*, *lambda21*, *calc=None*)

Calculates activity coefficients at fixed *lambda* values for a binary system at a series of mole fractions. This is used for regression of *lambda* parameters. This function is highly optimized, and operates on multiple points at a time.

$$\ln \gamma_1 = -\ln(x_1 + \Lambda_{12}x_2) + x_2\left(\frac{\Lambda_{12}}{x_1 + \Lambda_{12}x_2} - \frac{\Lambda_{21}}{x_2 + \Lambda_{21}x_1}\right)$$

$$\ln \gamma_2 = -\ln(x_2 + \Lambda_{21}x_1) - x_1\left(\frac{\Lambda_{12}}{x_1 + \Lambda_{12}x_2} - \frac{\Lambda_{21}}{x_2 + \Lambda_{21}x_1}\right)$$

Parameters

xs [list[float]] Liquid mole fractions of each species in the format x0_0, x1_0, (component 1 point1, component 2 point 1), x0_1, x1_1, (component 1 point2, component 2 point 2), ... [-]

lambda12 [float] *lambda* parameter for 12, [-]

lambda21 [float] *lambda* parameter for 21, [-]

gammas [list[float], optional] Array to store the activity coefficient for each species in the liquid mixture, indexed the same as *xs*; can be omitted or provided for slightly better performance [-]

Returns

gammas [list[float]] Activity coefficient for each species in the liquid mixture, indexed the same as *xs*, [-]

Notes

The lambda values are hard-coded to replace values under zero which are mathematically impossible, with a very small number. This is helpful for regression which might try to make those values negative.

Examples

```
>>> wilson_gammas_binaries([.1, .9, 0.3, 0.7, .85, .15], 0.1759, 0.7991)
[3.42989, 1.03432, 1.74338, 1.21234, 1.01766, 2.30656]
```

7.36 UNIQUAC Gibbs Excess Model (thermo.uniquac)

This module contains a class `UNIQUAC` for performing activity coefficient calculations with the UNIQUAC model. An older, functional calculation for activity coefficients only is also present, `UNIQUAC_gammas`.

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

7.36.1 UNIQUAC Class

class thermo.uniquac.**UNIQUAC**(*T*, *xs*, *rs*, *qs*, *tau_coeffs=None*, *ABCDEF=None*, *tau_as=None*, *tau_bs=None*, *tau_cs=None*, *tau_ds=None*, *tau_es=None*, *tau_fs=None*)

Bases: `thermo.activity.GibbsExcess`

Class for representing an a liquid with excess gibbs energy represented by the UNIQUAC equation. This model is capable of representing VL and LL behavior.

$$\frac{G^E}{RT} = \sum_i x_i \ln\frac{\phi_i}{x_i} + \frac{z}{2}\sum_i q_i x_i \ln\frac{\theta_i}{\phi_i} - \sum_i q_i x_i \ln\left(\sum_j \theta_j \tau_{ji}\right)$$

$$\phi_i = \frac{r_i x_i}{\sum_j r_j x_j}$$

$$\theta_i = \frac{q_i x_i}{\sum_j q_j x_j}$$

$$\tau_{ij} = \exp\left[a_{ij} + \frac{b_{ij}}{T} + c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2} + f_{ij}T^2\right]$$

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

rs [list[float]] r parameters $r_i = \sum_{k=1}^{n} \nu_k R_k$ if from UNIFAC, otherwise regressed, [-]

qs [list[float]] q parameters $q_i = \sum_{k=1}^{n} \nu_k Q_k$ if from UNIFAC, otherwise regressed, [-]

tau_coeffs [list[list[list[float]]], optional] UNIQUAC parameters, indexed by [i][j] and then each value is a 6 element list with parameters [a, b, c, d, e, f]; either *tau_coeffs* or *ABCDEF* are required, [-]

ABCDEF [tuple[list[list[float]], 6], optional] Contains the following. One of *tau_coeffs* or *ABCDEF* or some of the *tau_as*, etc parameters are required, [-]

tau_as [list[list[float]] or None, optional] a parameters used in calculating `UNIQUAC.taus`, [-]

tau_bs [list[list[float]] or None, optional] b parameters used in calculating `UNIQUAC.taus`, [K]

tau_cs [list[list[float]] or None, optional] c parameters used in calculating `UNIQUAC.taus`, [-]

tau_ds [list[list[float]] or None, optional] d paraemeters used in calculating `UNIQUAC.taus`, [1/K]

tau_es [list[list[float]] or None, optional] e parameters used in calculating `UNIQUAC.taus`, [K^2]

tau_fs [list[list[float]] or None, optional] f parameters used in calculating `UNIQUAC.taus`, [1/K^2]

Notes

In addition to the methods presented here, the methods of its base class `thermo.activity.GibbsExcess` are available as well.

Warning: There is no such thing as a missing parameter in the UNIQUAC model. It is possible to find τ_{ij} and τ_{ji} which make $\gamma_i = 1$ and $\gamma_j = 1$, but those tau values depend on *rs*, *qs*, and *xs* - the composition, which obviously will change. It is therefore impossible to make an interaction parameter "missing"; whatever value it has will always impact the phase equilibria problem. At best, the tau values can produce close to ideal behavior.

References

[1], [2]

Examples

Example 1

Example 5.19 in [2] includes the calculation of liquid-liquid activity coefficients for the water-ethanol-benzene system. Two calculations are reproduced accurately here. Note that the DDBST-style coefficients assume a negative sign; for compatibility, their coefficients need to have their sign flipped.

```
>>> N = 3
>>> T = 25.0 + 273.15
>>> xs = [0.7273, 0.0909, 0.1818]
>>> rs = [.92, 2.1055, 3.1878]
>>> qs = [1.4, 1.972, 2.4]
>>> tausA = tausC = tausD = tausE = tausF = [[0.0]*N for i in range(N)]
>>> tausB = [[0, 526.02, 309.64], [-318.06, 0, -91.532], [1325.1, 302.57, 0]]
>>> tausB = [[-v for v in r] for r in tausB] # Flip the sign to come into UNIQUAC
↪convention
```

(continues on next page)

(continued from previous page)

```
>>> ABCDEF = (tausA, tausB, tausC, tausD, tausE, tausF)
>>> GE = UNIQUAC(T=T, xs=xs, rs=rs, qs=qs, ABCDEF=ABCDEF)
>>> GE.gammas()
[1.570393328, 0.2948241614, 18.114329048]
```

The given values in [2] are [1.570, 0.2948, 18.11], matching exactly. The second phase has a different composition; the expected values are [8.856, 0.860, 1.425]. Once the *UNIQUAC* object has been constructed, it is very easy to obtain properties at different conditions:

```
>>> GE.to_T_xs(T=T, xs=[1/6., 1/6., 2/3.]).gammas()
[8.8559908058, 0.8595242462, 1.42546014081]
```

The string representation of the object presents enough information to reconstruct it as well.

```
>>> GE
UNIQUAC(T=298.15, xs=[0.7273, 0.0909, 0.1818], rs=[0.92, 2.1055, 3.1878], qs=[1.4,␣
↪1.972, 2.4], ABCDEF=([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]], [[0, -
↪526.02, -309.64], [318.06, 0, 91.532], [-1325.1, -302.57, 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, 0.0]], [[0.0, 0.0, 0.0], [0.
↪0, 0.0, 0.0], [0.0, 0.0, 0.0]]))
```

The phase exposes many properties and derivatives as well.

```
>>> GE.GE(), GE.dGE_dT(), GE.d2GE_dT2()
(1843.96486834, 6.69851118521, -0.015896025970)
>>> GE.HE(), GE.SE(), GE.dHE_dT(), GE.dSE_dT()
(-153.19624152, -6.69851118521, 4.7394001431, 0.0158960259705)
```

Example 2

Another problem is 8.32 in [1] - acetonitrile, benzene, n-heptane at 45 °C. The sign flip is needed here as well to convert their single temperature-dependent values into the correct form, but it has already been done to the coefficients:

```
>>> N = 3
>>> T = 45 + 273.15
>>> xs = [.1311, .0330, .8359]
>>> rs = [1.87, 3.19, 5.17]
>>> qs = [1.72, 2.4, 4.4]
>>> tausA = tausC = tausD = tausE = tausF = [[0.0]*N for i in range(N)]
>>> tausB = [[0.0, -60.28, -23.71], [-89.57, 0.0, 135.9], [-545.8, -245.4, 0.0]]
>>> ABCDEF = (tausA, tausB, tausC, tausD, tausE, tausF)
>>> GE = UNIQUAC(T=T, xs=xs, rs=rs, qs=qs, ABCDEF=ABCDEF)
>>> GE.gammas()
[7.1533533992, 1.25052436922, 1.060392792605]
```

The given values in [1] are [7.15, 1.25, 1.06].

Example 3

ChemSep is a program for modeling distillation. Chemsep ships with a permissive license several sets of binary interaction parameters. The UNIQUAC parameters in it can be accessed from Thermo as follows. In the following case, we compute activity coefficients of the ethanol-water system at mole fractions of [.252, 0.748].

```
>>> from thermo.interaction_parameters import IPDB
>>> CAS1, CAS2 = '64-17-5', '7732-18-5'
>>> xs = [0.252, 0.748]
>>> rs = [2.11, 0.92]
>>> qs = [1.97, 1.400]
>>> N = 2
>>> T = 343.15
>>> tau_bs = IPDB.get_ip_asymmetric_matrix(name='ChemSep UNIQUAC', CASs=['64-17-5',
↪'7732-18-5'], ip='bij')
>>> GE = UNIQUAC(T=T, xs=xs, rs=rs, qs=qs, tau_bs=tau_bs)
>>> GE.gammas()
[1.977454, 1.1397696]
```

In ChemSep, the form of the UNIQUAC *tau* equation is

$$\tau_{ij} = \exp\left(\frac{-A_{ij}}{RT}\right)$$

The parameters were converted to the form used by Thermo as follows:

$$b_{ij} = \frac{-A_{ij}}{R} = \frac{-A_{ij}}{1.9872042586408316}$$

This system was chosen because there is also a sample problem for the same components from the DDBST which can be found here: http://chemthermo.ddbst.com/Problems_Solutions/Mathcad_Files/P05.01c%20VLE%20Behavior%20of%20Ethanol%20-%20Water%20Using%20UNIQUAC.xps

In that example, with different data sets and parameters, they obtain at the same conditions activity coefficients of [2.359, 1.244].

Attributes

T [float] Temperature, [K]

xs [list[float]] Mole fractions, [-]

Methods

GE()	Calculate and return the excess Gibbs energy of a liquid phase using the UNIQUAC model.
d2GE_dT2()	Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phase using the UNIQUAC model.
d2GE_dTdxs()	Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy using the UNIQUAC model.
d2GE_dxixjs()	Calculate and return the second mole fraction derivatives of excess Gibbs energy using the UNIQUAC model.
d2taus_dT2()	Calculate and return the second temperature derivative of the *tau*
d3GE_dT3()	Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase using the UNIQUAC model.

continues on next page

Table 113 – continued from previous page

d3taus_dT3()	Calculate and return the third temperature derivative of the *tau* terms for the UNIQUAC model for a specified temperature.
dGE_dT()	Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase using the UNIQUAC model.
dGE_dxs()	Calculate and return the mole fraction derivatives of excess Gibbs energy using the UNIQUAC model.
dtaus_dT()	Calculate and return the temperature derivative of the *tau* terms for the UNIQUAC model for a specified temperature.
phis()	Calculate and return the *phi* parameters at the system composition and temperature.
regress_binary_parameters(gammas, xs, rs, qs)	Perform a basic regression to determine the values of the *tau* terms in the UNIQUAC model, given a series of known or predicted activity coefficients and mole fractions.
taus()	Calculate and return the *tau* terms for the UNIQUAC model for the system temperature.
thetas()	Calculate and return the *theta* parameters at the system composition and temperature.
to_T_xs(T, xs)	Method to construct a new *UNIQUAC* instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

GE()

Calculate and return the excess Gibbs energy of a liquid phase using the UNIQUAC model.

$$\frac{G^E}{RT} = \sum_i x_i \ln\frac{\phi_i}{x_i} + \frac{z}{2}\sum_i q_i x_i \ln\frac{\theta_i}{\phi_i} - \sum_i q_i x_i \ln\left(\sum_j \theta_j \tau_{ji}\right)$$

Returns

GE [float] Excess Gibbs energy, [J/mol]

d2GE_dT2()

Calculate and return the second temperature derivative of excess Gibbs energy of a liquid phase using the UNIQUAC model.

$$\frac{\partial G^E}{\partial T^2} = -R\left[T\sum_i\left(\frac{q_i x_i(\sum_j \theta_j \frac{\partial^2 \tau_{ji}}{\partial T^2})}{\sum_j \theta_j \tau_{ji}} - \frac{q_i x_i(\sum_j \theta_j \frac{\partial \tau_{ji}}{\partial T})^2}{(\sum_j \theta_j \tau_{ji})^2}\right) + 2\left(\sum_i \frac{q_i x_i(\sum_j \theta_j \frac{\partial \tau_{ji}}{\partial T})}{\sum_j \theta_j \tau_{ji}}\right)\right]$$

Returns

d2GE_dT2 [float] Second temperature derivative of excess Gibbs energy, [J/(mol*K^2)]

d2GE_dTdxs()

Calculate and return the temperature derivative of mole fraction derivatives of excess Gibbs energy using the UNIQUAC model.

$$\frac{\partial G^E}{\partial x_i \partial T} = R\left[-T\left\{\frac{q_i(\sum_j \theta_j \frac{\partial \tau_{ji}}{\partial T})}{\sum_j \tau_{ki}\theta_k} + \sum_j \frac{q_j x_j(\sum_k \frac{\partial \tau_{kj}}{\partial T}\frac{\partial \theta_k}{\partial x_i})}{\sum_k \tau_{kj}\theta_k} - \sum_j \frac{q_j x_j(\sum_k \tau_{kj}\frac{\partial \theta_k}{\partial x_i})(\sum_k \theta_k \frac{\partial \tau_{kj}}{\partial T})}{(\sum_k \tau_{kj}\theta_k)^2}\right\} + \sum_j \frac{q_j x_j z \left(\frac{\partial \theta_j}{\partial x_i}\right.}{2\theta}\right.$$

Returns

d2GE_dTdxs [list[float]] Temperature derivative of mole fraction derivatives of excess Gibbs energy, [J/(mol*K)]

`d2GE_dxixjs()`

Calculate and return the second mole fraction derivatives of excess Gibbs energy using the UNIQUAC model.

$$\frac{\partial^2 g^E}{\partial x_i \partial x_j}$$

Returns

d2GE_dxixjs [list[list[float]]] Second mole fraction derivatives of excess Gibbs energy, [J/mol]

Notes

The formula is extremely long and painful; see the source code for details.

`d2taus_dT2()`

Calculate and return the second temperature derivative of the *tau* terms for the UNIQUAC model for a specified temperature.

$$\frac{\partial^2 \tau_{ij}}{\partial^2 T} = \left(2f_{ij} + \left(2Tf_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right)^2 - \frac{c_{ij}}{T^2} + \frac{2b_{ij}}{T^3} + \frac{6e_{ij}}{T^4}\right) e^{T^2 f_{ij} + Td_{ij} + a_{ij} + c_{ij}\ln(T) + \frac{b_{ij}}{T} + \frac{e_{ij}}{T^2}}$$

Returns

d2taus_dT2 [list[list[float]]] Second temperature derivatives of tau terms, asymmetric matrix [1/K^2]

Notes

These values (and the coefficients) are NOT symmetric.

`d3GE_dT3()`

Calculate and return the third temperature derivative of excess Gibbs energy of a liquid phase using the UNIQUAC model.

$$\frac{\partial^3 G^E}{\partial T^3} = -R\left[T\sum_i \left(\frac{q_i x_i(\sum_j \theta_j \frac{\partial^3 \tau_{ji}}{\partial T^3})}{(\sum_j \theta_j \tau_{ji})} - \frac{3q_i x_i(\sum_j \theta_j \frac{\partial^2 \tau_{ji}}{\partial T^2})(\sum_j \theta_j \frac{\partial \tau_{ji}}{\partial T})}{(\sum_j \theta_j \tau_{ji})^2} + \frac{2q_i x_i(\sum_j \theta_j \frac{\partial \tau_{ji}}{\partial T})^3}{(\sum_j \theta_j \tau_{ji})^3}\right) + \sum_i \left(\frac{3q_i x_i(\sum_j \right.\right.$$

Returns

d3GE_dT3 [float] Third temperature derivative of excess Gibbs energy, [J/(mol*K^3)]

`d3taus_dT3()`

Calculate and return the third temperature derivative of the *tau* terms for the UNIQUAC model for a specified temperature.

$$\frac{\partial^3 \tau_{ij}}{\partial^3 T} = \left(3\left(2f_{ij} - \frac{c_{ij}}{T^2} + \frac{2b_{ij}}{T^3} + \frac{6e_{ij}}{T^4}\right)\left(2Tf_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right) + \left(2Tf_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right)^3 - \right.$$

Returns

d3taus_dT3 [list[list[float]]] Third temperature derivatives of tau terms, asymmetric matrix [1/K^3]

Notes

These values (and the coefficients) are NOT symmetric.

dGE_dT()

Calculate and return the temperature derivative of excess Gibbs energy of a liquid phase using the UNIQUAC model.

$$\frac{\partial G^E}{\partial T} = \frac{G^E}{T} - RT\left(\sum_i \frac{q_i x_i(\sum_j \theta_j \frac{\partial \tau_{ji}}{\partial T})}{\sum_j \theta_j \tau_{ji}}\right)$$

Returns

dGE_dT [float] First temperature derivative of excess Gibbs energy, [J/(mol*K)]

dGE_dxs()

Calculate and return the mole fraction derivatives of excess Gibbs energy using the UNIQUAC model.

$$\frac{\partial G^E}{\partial x_i} = RT\left[\sum_j \frac{q_j x_j \phi_j z}{2\theta_j}\left(\frac{1}{\phi_j}\cdot\frac{\partial \theta_j}{\partial x_i} - \frac{\theta_j}{\phi_j^2}\cdot\frac{\partial \phi_j}{\partial x_i}\right) - \sum_j \left(\frac{q_j x_j(\sum_k \tau_{kj}\frac{\partial \theta_k}{\partial x_i})}{\sum_k \tau_{kj}\theta_k}\right) + 0.5 z q_i \ln\left(\frac{\theta_i}{\phi_i}\right) - q_i \ln\left(\sum_j \tau_{ji}\theta_j\right)\right.$$

Returns

dGE_dxs [list[float]] Mole fraction derivatives of excess Gibbs energy, [J/mol]

dtaus_dT()

Calculate and return the temperature derivative of the *tau* terms for the UNIQUAC model for a specified temperature.

$$\frac{\partial \tau_{ij}}{\partial T} = \left(2Th_{ij} + d_{ij} + \frac{c_{ij}}{T} - \frac{b_{ij}}{T^2} - \frac{2e_{ij}}{T^3}\right) e^{T^2 h_{ij} + T d_{ij} + a_{ij} + c_{ij}\ln(T) + \frac{b_{ij}}{T} + \frac{e_{ij}}{T^2}}$$

Returns

dtaus_dT [list[list[float]]] First temperature derivatives of tau terms, asymmetric matrix [1/K]

Notes

These values (and the coefficients) are NOT symmetric.

phis()

Calculate and return the *phi* parameters at the system composition and temperature.

$$\phi_i = \frac{r_i x_i}{\sum_j r_j x_j}$$

Returns

phis [list[float]] phi parameters, [-]

classmethod regress_binary_parameters(*gammas*, *xs*, *rs*, *qs*, *use_numba=False*, *do_statistics=True*, ***kwargs*)

Perform a basic regression to determine the values of the *tau* terms in the UNIQUAC model, given a series of known or predicted activity coefficients and mole fractions.

Parameters

gammas [list[list[float, 2]]] List of activity coefficient pairs, [-]

xs [list[list[float, 2]]] List of binary mole fraction pairs, [-]

rs [list[float]] Van der Waals volume parameters for each species, [-]

qs [list[float]] Surface area parameters for each species, [-]

use_numba [bool, optional] Whether or not to try to use numba to speed up the computation, [-]

do_statistics [bool, optional] Whether or not to compute statistical measures on the outputs, [-]

kwargs [dict] Extra parameters to be passed to the fitting function (not yet documented), [-]

Returns

parameters [dict[str, float]] Dimentionless interaction parameters of each compound with each other; these are the actual *tau* values. [-]

statistics [dict[str: float]] Statistics, calculated and returned only if *do_statistics* is True, [-]

Notes

Notes on getting fitting coefficients that yield gammas of 1:

- This is possible some of the time to a pretty high accuracy
- This is not possible whatsoever in some cases
- The values of *rs*, and *qs* determine how close the fitting can be
- If *rs* and *qs* are close to each other, it may well fit nicely
- If they are distant (1.2-1.5x) they usually will not fit

Examples

In the following example, the *tau* values required to zero-out the coefficients for the n-pentane and n-hexane system are calculated. The parameters are converted back into *aij* parameters as used by this activity coefficient object, and then the calculated values are verified to be fairly nearly one.

```
>>> from thermo import UNIQUAC
>>> import numpy as np
>>> pts = 30
>>> rs = [3.8254, 4.4998]
>>> qs = [3.316, 3.856]
>>> xs = [[xi, 1.0 - xi] for xi in np.linspace(1e-7, 1-1e-7, pts)]
>>> gammas = [[1, 1] for i in range(pts)]
>>> coeffs, stats = UNIQUAC.regress_binary_parameters(gammas, xs, rs, qs)
>>> coeffs
{'tau12': 1.04220685, 'tau21': 0.95538082}
>>> assert stats['MAE'] < 1e-6
>>> tausB = tausC = tausD = tausE = tausF = [[0.0]*2 for i in range(2)]
>>> tausA = [[0, np.log(coeffs['tau12'])], [np.log(coeffs['tau21']), 0]]
>>> ABCDEF = (tausA, tausB, tausC, tausD, tausE, tausF)
>>> GE = UNIQUAC(T=300, xs=[.5, .5], rs=rs, qs=qs, ABCDEF=ABCDEF)
>>> GE.gammas()
[1.000000466, 1.000000180]
```

Note how the *tau* coefficients need to be converted into the *a* parameters of the *tau* equation. They could also have been converted into any of the other parameters, but then the activity coefficients predicted would no longer be close to 1 at other temperatures.

$$\tau_{ij} = \exp\left[a_{ij} + \frac{b_{ij}}{T} + c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2} + f_{ij}T^2\right]$$

The UNIQUAC model's *r* and *q* parameters create their own biases in the model, based on the structure of each of the pure species. Water and n-pentane are not miscible liquids; they will form two liquid phases except when one component is present in trace amounts. No matter the values of *tau*, it is not possible to make the UNIQUAC equation predict activity coefficients very close to one for this system, as shown in the following sample.

```
>>> rs = [3.8254, 0.92]
>>> qs = [3.316, 1.4]
>>> pts = 6
>>> xs = [[xi, 1.0 - xi] for xi in np.linspace(1e-7, 1-1e-7, pts)]
>>> gammas = [[1, 1] for i in range(pts)]
>>> coeffs, stats = UNIQUAC.regress_binary_parameters(gammas, xs, rs, qs)
>>> stats['MAE']
0.0254
```

taus()

Calculate and return the *tau* terms for the UNIQUAC model for the system temperature.

$$\tau_{ij} = \exp\left[a_{ij} + \frac{b_{ij}}{T} + c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2} + f_{ij}T^2\right]$$

Returns

taus [list[list[float]]] tau terms, asymmetric matrix [-]

Notes

These *tau ij* values (and the coefficients) are NOT symmetric.

thetas()

Calculate and return the *theta* parameters at the system composition and temperature.

$$\theta_i = \frac{q_i x_i}{\sum_j q_j x_j}$$

Returns

thetas [list[float]] theta parameters, [-]

to_T_xs(*T*, *xs*)

Method to construct a new *UNIQUAC* instance at temperature *T*, and mole fractions *xs* with the same parameters as the existing object.

Parameters

T [float] Temperature, [K]

xs [list[float]] Mole fractions of each component, [-]

Returns

obj [UNIQUAC] New *UNIQUAC* object at the specified conditions [-]

Notes

If the new temperature is the same temperature as the existing temperature, if the *tau* terms or their derivatives have been calculated, they will be set to the new object as well.

7.36.2 UNIQUAC Functional Calculations

`thermo.uniquac.UNIQUAC_gammas`(*xs*, *rs*, *qs*, *taus*)

Calculates the activity coefficients of each species in a mixture using the Universal quasi-chemical (UNIQUAC) equation, given their mole fractions, *rs*, *qs*, and dimensionless interaction parameters. The interaction parameters are normally correlated with temperature, and need to be calculated separately.

$$\ln \gamma_i = \ln \frac{\Phi_i}{x_i} + \frac{z}{2} q_i \ln\frac{\theta_i}{\Phi_i} + l_i - \frac{\Phi_i}{x_i}\sum_j^N x_j l_j - q_i \ln\left(\sum_j^N \theta_j \tau_{ji}\right) + q_i - q_i\sum_j^N \frac{\theta_j \tau_{ij}}{\sum_k^N \theta_k \tau_{kj}}$$

$$\theta_i = \frac{x_i q_i}{\displaystyle\sum_{j=1}^{n} x_j q_j}$$

$$\Phi_i = \frac{x_i r_i}{\displaystyle\sum_{j=1}^{n} x_j r_j}$$

$$l_i = \frac{z}{2}(r_i - q_i) - (r_i - 1)$$

Parameters

xs [list[float]] Liquid mole fractions of each species, [-]

rs [list[float]] Van der Waals volume parameters for each species, [-]

qs [list[float]] Surface area parameters for each species, [-]

taus [list[list[float]]] Dimensionless interaction parameters of each compound with each other, [-]

Returns

gammas [list[float]] Activity coefficient for each species in the liquid mixture, [-]

Notes

This model needs N^2 parameters.

The original expression for the interaction parameters is as follows:

$$\tau_{ji} = \exp\left(\frac{-\Delta u_{ij}}{RT}\right)$$

However, it is seldom used. Most correlations for the interaction parameters include some of the terms shown in the following form:

$$\ln \tau_{ij} = a_{ij} + \frac{b_{ij}}{T} + c_{ij}\ln T + d_{ij}T + \frac{e_{ij}}{T^2}$$

This model is recast in a slightly more computationally efficient way in [2], as shown below:

$$\ln \gamma_i = \ln \gamma_i^{res} + \ln \gamma_i^{comb}$$

$$\ln \gamma_i^{res} = q_i \left(1 - \ln \frac{\sum_j^N q_j x_j \tau_{ji}}{\sum_j^N q_j x_j} - \sum_j \frac{q_k x_j \tau_{ij}}{\sum_k q_k x_k \tau_{kj}}\right)$$

$$\ln \gamma_i^{comb} = (1 - V_i + \ln V_i) - \frac{z}{2} q_i \left(1 - \frac{V_i}{F_i} + \ln \frac{V_i}{F_i}\right)$$

$$V_i = \frac{r_i}{\sum_j^N r_j x_j}$$

$$F_i = \frac{q_i}{\sum_j q_j x_j}$$

There is no global set of parameters which will make this model yield ideal acitivty coefficients (gammas = 1) for this model.

References

[1], [2], [3]

Examples

Ethanol-water example, at 343.15 K and 1 MPa:

```
>>> UNIQUAC_gammas(xs=[0.252, 0.748], rs=[2.1055, 0.9200], qs=[1.972, 1.400],
... taus=[[1.0, 1.0919744384510301], [0.37452902779205477, 1.0]])
[2.35875137797083, 1.2442093415968987]
```

7.37 Joback Group Contribution Method (thermo.group_contribution.joback)

This module contains an implementation of the Joback group-contribution method. This functionality requires the RDKit library to work.

For submitting pull requests, please use the GitHub issue tracker.

Warning: The Joback class method does not contain all the groups for every chemical. There are often multiple ways of fragmenting a chemical. Other times, the fragmentation algorithm will fail. These limitations are present in both the implementation and the method itself. You are welcome to seek to improve this code but no to little help can be offered.

class thermo.group_contribution.joback.Joback(*mol*, *atom_count=None*, *MW=None*, *Tb=None*)

Bases: object

Class for performing chemical property estimations with the Joback group contribution method as described in [1] and [2]. This is a very common method with low accuracy but wide applicability. This routine can be used with either its own automatic fragmentation routine, or user specified groups. It is applicable to organic compounds only, and has only 41 groups with no interactions between them. Each method's documentation describes its accuracy. The automatic fragmentation routine is possible only because of the development of SMARTS expressions to match the Joback groups by Dr. Jason Biggs. The list of SMARTS expressions was posted publically on the RDKit mailing list.

Parameters

mol [rdkitmol or smiles str] Input molecule for the analysis, [-]

atom_count [int, optional] The total number of atoms including hydrogen in the molecule; this will be counted by rdkit if it not provided, [-]

MW [float, optional] Molecular weight of the molecule; this will be calculated by rdkit if not provided, [g/mol]

Tb [float, optional] An experimentally known boiling temperature for the chemical; this increases the accuracy of the calculated critical point if provided. [K]

Notes

Be sure to check the status of the automatic fragmentation; not all chemicals with the Joback method are applicable.

Approximately 68% of chemcials in the thermo database seem to be able to be estimated with the Joback method.

If a group which was identified is missign a regressed contribution, the estimated property will be None. However, if not all atoms of a molecule are identified as particular groups, property estimation will go ahead with heavily reduced accuracy. Check the *status* attribute to be sure a molecule was properly fragmented.

References

[1], [2]

Examples

Analysis of Acetone:

```
>>> J = Joback('CC(=O)C')
>>> J.Hfus(J.counts)
5125.0
>>> J.Cpig(350)
84.69109750000001
>>> J.status
'OK'
```

All properties can be obtained in one go with the *estimate* method:

```
>>> J.estimate(callables=False)
{'Tb': 322.11, 'Tm': 173.5, 'Tc': 500.5590049525365, 'Pc': 4802499.604994407, 'Vc':␣
↪0.0002095, 'Hf': -217829.99999999997, 'Gf': -154540.00000000003, 'Hfus': 5125.0,
↪'Hvap': 29018.0, 'mul_coeffs': [839.1099999999998, -14.99], 'Cpig_coeffs': [7.
↪520000000000003, 0.26084, -0.0001207, 1.545999999999998e-08]}
```

The results for propionic anhydride (if the status is not OK) should not be used.

```
>>> J = Joback('CCC(=O)OC(=O)CC')
>>> J.status
'Matched some atoms repeatedly: [4]'
>>> J.Cpig(300)
175.85999999999999
```

None of the routines need to use the automatic routine; they can be used manually too:

```
>>> Joback.Tb({1: 2, 24: 1})
322.11
```

Attributes

calculated_Cpig_coeffs

calculated_mul_coeffs

Methods

`Cpig`(T)	Computes ideal-gas heat capacity at a specified temperature of an organic compound using the Joback method as a function of chemical structure only.
`Cpig_coeffs`(counts)	Computes the ideal-gas polynomial heat capacity coefficients of an organic compound using the Joback method as a function of chemical structure only.
`Gf`(counts)	Estimates the ideal-gas Gibbs energy of formation at 298.15 K of an organic compound using the Joback method as a function of chemical structure only.
`Hf`(counts)	Estimates the ideal-gas enthalpy of formation at 298.15 K of an organic compound using the Joback method as a function of chemical structure only.
`Hfus`(counts)	Estimates the enthalpy of fusion of an organic compound at its melting point using the Joback method as a function of chemical structure only.
`Hvap`(counts)	Estimates the enthalpy of vaporization of an organic compound at its normal boiling point using the Joback method as a function of chemical structure only.
`Pc`(counts, atom_count)	Estimates the critcal pressure of an organic compound using the Joback method as a function of chemical structure only.
`Tb`(counts)	Estimates the normal boiling temperature of an organic compound using the Joback method as a function of chemical structure only.
`Tc`(counts[, Tb])	Estimates the critcal temperature of an organic compound using the Joback method as a function of chemical structure only, or optionally improved by using an experimental boiling point.
`Tm`(counts)	Estimates the melting temperature of an organic compound using the Joback method as a function of chemical structure only.
`Vc`(counts)	Estimates the critcal volume of an organic compound using the Joback method as a function of chemical structure only.
`estimate`([callables])	Method to compute all available properties with the Joback method; returns their results as a dict.

continues on next page

Table 114 – continued from previous page

mul(T)	Computes liquid viscosity at a specified temperature of an organic compound using the Joback method as a function of chemical structure only.
mul_coeffs(counts)	Computes the liquid phase viscosity Joback coefficients of an organic compound using the Joback method as a function of chemical structure only.

Cpig(*T*)

Computes ideal-gas heat capacity at a specified temperature of an organic compound using the Joback method as a function of chemical structure only.

$$C_p^{ig} = \sum_i a_i - 37.93 + \left[\sum_i b_i + 0.210\right] T + \left[\sum_i c_i - 3.91 \cdot 10^{-4}\right] T^2 + \left[\sum_i d_i + 2.06 \cdot 10^{-7}\right] T^3$$

Parameters

T [float] Temperature, [K]

Returns

Cpig [float] Ideal-gas heat capacity, [J/mol/K]

Examples

```
>>> J = Joback('CC(=O)C')
>>> J.Cpig(300)
75.32642000000001
```

static Cpig_coeffs(*counts*)

Computes the ideal-gas polynomial heat capacity coefficients of an organic compound using the Joback method as a function of chemical structure only.

$$C_p^{ig} = \sum_i a_i - 37.93 + \left[\sum_i b_i + 0.210\right] T + \left[\sum_i c_i - 3.91 \cdot 10^{-4}\right] T^2 + \left[\sum_i d_i + 2.06 \cdot 10^{-7}\right] T^3$$

288 compounds were used by Joback in this determination. No overall error was reported.

The ideal gas heat capacity values used in developing the heat capacity polynomials used 9 data points between 298 K and 1000 K.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

coefficients [list[float]] Coefficients which will result in a calculated heat capacity in in units of J/mol/K, [-]

Examples

```
>>> c = Joback.Cpig_coeffs({1: 2, 24: 1})
>>> c
[7.520000000000003, 0.26084, -0.0001207, 1.545999999999998e-08]
>>> Cp = lambda T : c[0] + c[1]*T + c[2]*T**2 + c[3]*T**3
>>> Cp(300)
75.32642000000001
```

static Gf(*counts*)

Estimates the ideal-gas Gibbs energy of formation at 298.15 K of an organic compound using the Joback method as a function of chemical structure only.

$$G_{formation} = 53.88 + \sum G_{f,i}$$

In the above equation, Gibbs energy of formation is calculated in kJ/mol; it is converted to J/mol here.

328 compounds were used by Joback in this determination, with an absolute average error of 2.0 kcal/mol, standard devaition 4.37 kcal/mol, and AARE of 15.7%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Gf [float] Estimated ideal-gas Gibbs energy of formation at 298.15 K, [J/mol]

Examples

```
>>> Joback.Gf({1: 2, 24: 1})
-154540.00000000003
```

static Hf(*counts*)

Estimates the ideal-gas enthalpy of formation at 298.15 K of an organic compound using the Joback method as a function of chemical structure only.

$$H_{formation} = 68.29 + \sum_i H_{f,i}$$

In the above equation, enthalpy of formation is calculated in kJ/mol; it is converted to J/mol here.

370 compounds were used by Joback in this determination, with an absolute average error of 2.2 kcal/mol, standard devaition 2.0 kcal/mol, and AARE of 15.2%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Hf [float] Estimated ideal-gas enthalpy of formation at 298.15 K, [J/mol]

Examples

```
>>> Joback.Hf({1: 2, 24: 1})
-217829.99999999997
```

static Hfus(*counts*)

Estimates the enthalpy of fusion of an organic compound at its melting point using the Joback method as a function of chemical structure only.

$$\Delta H_{fus} = -0.88 + \sum_i H_{fus,i}$$

In the above equation, enthalpy of fusion is calculated in kJ/mol; it is converted to J/mol here.

For 155 compounds tested by Joback, the absolute average error was 485.2 cal/mol and standard deviation was 661.4 cal/mol; the average relative error was 38.7%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Hfus [float] Estimated enthalpy of fusion of the compound at its melting point, [J/mol]

Examples

```
>>> Joback.Hfus({1: 2, 24: 1})
5125.0
```

static Hvap(*counts*)

Estimates the enthalpy of vaporization of an organic compound at its normal boiling point using the Joback method as a function of chemical structure only.

$$\Delta H_{vap} = 15.30 + \sum_i H_{vap,i}$$

In the above equation, enthalpy of fusion is calculated in kJ/mol; it is converted to J/mol here.

For 368 compounds tested by Joback, the absolute average error was 303.5 cal/mol and standard deviation was 429 cal/mol; the average relative error was 3.88%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Hvap [float] Estimated enthalpy of vaporization of the compound at its normal boiling point, [J/mol]

Examples

```
>>> Joback.Hvap({1: 2, 24: 1})
29018.0
```

static Pc(*counts*, *atom_count*)

Estimates the critcal pressure of an organic compound using the Joback method as a function of chemical structure only. This correlation was developed using the actual number of atoms forming the molecule as well.

$$P_c = \left[0.113 + 0.0032N_A - \sum_i P_{c,i}\right]^{-2}$$

In the above equation, critical pressure is calculated in bar; it is converted to Pa here.

392 compounds were used by Joback in this determination, with an absolute average error of 2.06 bar, standard devaition 3.2 bar, and AARE of 5.2%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

atom_count [int] Total number of atoms (including hydrogens) in the molecule, [-]

Returns

Pc [float] Estimated critical pressure, [Pa]

Examples

```
>>> Joback.Pc({1: 2, 24: 1}, 10)
4802499.604994407
```

static Tb(*counts*)

Estimates the normal boiling temperature of an organic compound using the Joback method as a function of chemical structure only.

$$T_b = 198.2 + \sum_i T_{b,i}$$

For 438 compounds tested by Joback, the absolute average error was 12.91 K and standard deviation was 17.85 K; the average relative error was 3.6%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Tb [float] Estimated normal boiling temperature, [K]

Examples

```
>>> Joback.Tb({1: 2, 24: 1})
322.11
```

static Tc(*counts*, *Tb=None*)

Estimates the critcal temperature of an organic compound using the Joback method as a function of chemical structure only, or optionally improved by using an experimental boiling point. If the experimental boiling point is not provided it will be estimated with the Joback method as well.

$$T_c = T_b \left[0.584 + 0.965 \sum_i T_{c,i} - \left(\sum_i T_{c,i}\right)^2 \right]^{-1}$$

For 409 compounds tested by Joback, the absolute average error was 4.76 K and standard deviation was 6.94 K; the average relative error was 0.81%.

Appendix BI of Joback's work lists 409 estimated critical temperatures.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Tb [float, optional] Experimental normal boiling temperature, [K]

Returns

Tc [float] Estimated critical temperature, [K]

Examples

```
>>> Joback.Tc({1: 2, 24: 1}, Tb=322.11)
500.5590049525365
```

static Tm(*counts*)

Estimates the melting temperature of an organic compound using the Joback method as a function of chemical structure only.

$$T_m = 122.5 + \sum_i T_{m,i}$$

For 388 compounds tested by Joback, the absolute average error was 22.6 K and standard deviation was 24.68 K; the average relative error was 11.2%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Tm [float] Estimated melting temperature, [K]

Examples

```
>>> Joback.Tm({1: 2, 24: 1})
173.5
```

static Vc(*counts*)

Estimates the critcal volume of an organic compound using the Joback method as a function of chemical structure only.

$$V_c = 17.5 + \sum_i V_{c,i}$$

In the above equation, critical volume is calculated in cm^3/mol; it is converted to m^3/mol here.

310 compounds were used by Joback in this determination, with an absolute average error of 7.54 cm^3/mol, standard devaition 13.16 cm^3/mol, and AARE of 2.27%.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

Vc [float] Estimated critical volume, [m^3/mol]

Examples

```
>>> Joback.Vc({1: 2, 24: 1})
0.0002095
```

calculated_Cpig_coeffs = None

calculated_mul_coeffs = None

estimate(*callables=True*)

Method to compute all available properties with the Joback method; returns their results as a dict. For the tempearture dependent values Cpig and mul, both the coefficients and objects to perform calculations are returned.

mul(*T*)

Computes liquid viscosity at a specified temperature of an organic compound using the Joback method as a function of chemical structure only.

$$\mu_{liq} = \text{MW} \exp\left(\frac{\sum_i \mu_a - 597.82}{T} + \sum_i \mu_b - 11.202 \right)$$

Parameters

T [float] Temperature, [K]

Returns

mul [float] Liquid viscosity, [Pa*s]

Examples

```
>>> J = Joback('CC(=O)C')
>>> J.mul(300)
0.0002940378347162687
```

static mul_coeffs(*counts*)

Computes the liquid phase viscosity Joback coefficients of an organic compound using the Joback method as a function of chemical structure only.

$$\mu_{liq} = \text{MW} \exp\left(\frac{\sum_i \mu_a - 597.82}{T} + \sum_i \mu_b - 11.202\right)$$

288 compounds were used by Joback in this determination. No overall error was reported.

The liquid viscosity data used was specified to be at "several temperatures for each compound" only. A small and unspecified number of compounds were used in this estimation.

Parameters

counts [dict] Dictionary of Joback groups present (numerically indexed) and their counts, [-]

Returns

coefficients [list[float]] Coefficients which will result in a liquid viscosity in in units of Pa*s, [-]

Examples

```
>>> mu_ab = Joback.mul_coeffs({1: 2, 24: 1})
>>> mu_ab
[839.1099999999998, -14.99]
>>> MW = 58.041864812
>>> mul = lambda T : MW*exp(mu_ab[0]/T + mu_ab[1])
>>> mul(300)
0.0002940378347162687
```

```
thermo.group_contribution.joback.J_BIGGS_JOBACK_SMARTS = [['Methyl', '-CH3', '[CX4H3]'], ['Secondary acyclic', '-CH2-', '[!R;CX4H2]'], ['Tertiary acyclic', '>CH-', '[!R;CX4H]'], ['Quaternary acyclic', '>C<', '[!R;CX4H0]'], ['Primary alkene', '=CH2', '[CX3H2]'], ['Secondary alkene acyclic', '=CH-', '[!R;CX3H1;!$([CX3H1](=O))]'], ['Tertiary alkene acyclic', '=C<', '[$([!R;CX3H0]);!$([!R;CX3H0]=[#8])]'], ['Cumulative alkene', '=C=', '[$([CX2H0](=*)=*)]'], ['Terminal alkyne', 'CH', '[$([CX2H1]#[!#7])]'], ['Internal alkyne', 'C-', '[$([CX2H0]#[!#7])]'], ['Secondary cyclic', '-CH2- (ring)', '[R;CX4H2]'], ['Tertiary cyclic', '>CH- (ring)', '[R;CX4H]'], ['Quaternary cyclic', '>C< (ring)', '[R;CX4H0]'], ['Secondary alkene cyclic', '=CH- (ring)', '[R;CX3H1,cX3H1]'], ['Tertiary alkene cyclic', '=C< (ring)', '[$([R;CX3H0]);!$([R;CX3H0]=[#8])]'], ['Fluoro', '-F', '[F]'], ['Chloro', '-Cl', '[Cl]'], ['Bromo', '-Br', '[Br]'], ['Iodo', '-I', '[I]'], ['Alcohol', '-OH (alcohol)', '[OX2H;!$([OX2H]-[#6]=[O]);!$([OX2H]-a)]'], ['Phenol', '-OH (phenol)', '[$([OX2H]-a)]'], ['Ether acyclic', '-O- (nonring)', '[OX2H0;!R;!$([OX2H0]-[#6]=[#8])]'], ['Ether cyclic', '-O- (ring)', '[#8X2H0;R;!$([#8X2H0]~[#6]=[#8])]'], ['Carbonyl acyclic', '>C=O (nonring)', '[$([CX3H0](=[OX1]));!$([CX3](=[OX1])-[OX2]);!R]=O'], ['Carbonyl cyclic', '>C=O (ring)', '[$([#6X3H0](=[OX1]));!$([#6X3](=[#8X1])~[#8X2]);R]=O'], ['Aldehyde', 'O=CH- (aldehyde)', '[CX3H1](=O)'], ['Carboxylic acid', '-COOH (acid)', '[OX2H]-[C]=O'], ['Ester', '-COO- (ester)', '[#6X3H0;!$([#6X3H0](~O)(~O)(~O))](=[#8X1])[#8X2H0]'], ['Oxygen double bond other', '=O (other than above)', '[OX1H0;!$([OX1H0]~[#6X3]);!$([OX1H0]~[#7X3]~[#8])]'], ['Primary amino', '-NH2', '[NX3H2]'], ['Secondary amino acyclic', '>NH (nonring)', '[NX3H1;!R]'], ['Secondary amino cyclic', '>NH (ring)', '[#7X3H1;R]'], ['Tertiary amino', '>N- (nonring)', '[#7X3H0;!$([#7](~O)~O)]'], ['Imine acyclic', '-N= (nonring)', '[#7X2H0;!R]'], ['Imine cyclic', '-N= (ring)', '[#7X2H0;R]'], ['Aldimine', '=NH', '[#7X2H1]'], ['Cyano', '-CN', '[#6X2]#[#7X1H0]'], ['Nitro', '-NO2', '[$([#7X3,#7X3+][!#8])](=[O])~[O-]'], ['Thiol', '-SH', '[SX2H]'], ['Thioether acyclic', '-S- (nonring)', '[#16X2H0;!R]'], ['Thioether cyclic', '-S- (ring)', '[#16X2H0;R]']]
```

Metadata for the Joback groups. The first element is the group name; the second is the group symbol; and the third is the SMARTS matching string.

```
thermo.group_contribution.joback.J_BIGGS_JOBACK_SMARTS_id_dict = {1: '[CX4H3]', 2: '[!R;CX4H2]', 3: '[!R;CX4H]', 4: '[!R;CX4H0]', 5: '[CX3H2]', 6: '[!R;CX3H1;!$([CX3H1](=O))]', 7: '[$([!R;CX3H0]);!$([!R;CX3H0]=[#8])]', 8: '[$([CX2H0](=*)=*)]', 9: '[$([CX2H1]#[!#7])]', 10: '[$([CX2H0]#[!#7])]', 11: '[R;CX4H2]', 12: '[R;CX4H]', 13: '[R;CX4H0]', 14: '[R;CX3H1,cX3H1]', 15: '[$([R;CX3H0]);!$([R;CX3H0]=[#8])]', 16: '[F]', 17: '[Cl]', 18: '[Br]', 19: '[I]', 20: '[OX2H;!$([OX2H]-[#6]=[O]);!$([OX2H]-a)]', 21: '[$([OX2H]-a)]', 22: '[OX2H0;!R;!$([OX2H0]-[#6]=[#8])]', 23: '[#8X2H0;R;!$([#8X2H0]~[#6]=[#8])]', 24: '[$([CX3H0](=[OX1]));!$([CX3](=[OX1])-[OX2]);!R]=O', 25: '[$([#6X3H0](=[OX1]));!$([#6X3](=[#8X1])~[#8X2]);R]=O', 26: '[CX3H1](=O)', 27: '[OX2H]-[C]=O', 28: '[#6X3H0;!$([#6X3H0](~O)(~O)(~O))](=[#8X1])[#8X2H0]', 29: '[OX1H0;!$([OX1H0]~[#6X3]);!$([OX1H0]~[#7X3]~[#8])]', 30: '[NX3H2]', 31: '[NX3H1;!R]', 32: '[#7X3H1;R]', 33: '[#7X3H0;!$([#7](~O)~O)]', 34: '[#7X2H0;!R]', 35: '[#7X2H0;R]', 36: '[#7X2H1]', 37: '[#6X2]#[#7X1H0]', 38: '[$([#7X3,#7X3+][!#8])](=[O])~[O-]', 39: '[SX2H]', 40: '[#16X2H0;!R]', 41: '[#16X2H0;R]'}
```

7.38 Fedors Group Contribution Method (thermo.group_contribution.fedors)

This module contains an implementation of the Fedors group-contribution method. This functionality requires the RDKit library to work.

`thermo.group_contribution.Fedors(mol)`

Estimate the critical volume of a molecule using the Fedors [1] method, which is a basic group contribution method that also uses certain bond count features and the number of different types of rings.

Parameters

mol [str or rdkit.Chem.rdchem.Mol, optional] Smiles string representing a chemical or a rdkit molecule, [-]

Returns

Vc [float] Estimated critical volume, [m^3/mol]

status [str] A string holding an explanation of why the molecule failed to be fragmented, if it fails; 'OK' if it suceeds, [-]

unmatched_atoms [bool] Whether or not all atoms in the molecule were matched successfully; if this is True, the results should not be trusted, [-]

unrecognized_bond [bool] Whether or not all bonds in the molecule were matched successfully; if this is True, the results should not be trusted, [-]

unrecognized_ring_size [bool] Whether or not all rings in the molecule were matched successfully; if this is True, the results should not be trusted, [-]

Notes

Raises an exception if rdkit is not installed, or *smi* or *rdkitmol* is not defined.

References

[1], [2]

Examples

Example for sec-butanol in [2]:

```
>>> Vc, status, _, _, _ = Fedors('CCC(C)O')
>>> Vc, status
(0.000274024, 'OK')
```

7.39 Wilson-Jasperson Group Contribution Method (thermo.group_contribution.wilson_jasperson)

This module contains an implementation of the Wilson-Jasperson group-contribution method. This functionality requires the RDKit library to work.

thermo.group_contribution.**Wilson_Jasperson**(*mol*, *Tb*, *second_order=True*)

Estimate the critical temperature and pressure of a molecule using the molecule itself, and a known or estimated boiling point using the Wilson-Jasperson method.

Parameters

mol [str or rdkit.Chem.rdchem.Mol, optional] Smiles string representing a chemical or a rdkit molecule, [-]

Tb [float] Known or estimated boiling point, [K]

second_order [bool] Whether to use the first order method (False), or the second order method, [-]

Returns

Tc [float] Estimated critical temperature, [K]

Pc [float] Estimated critical pressure, [Pa]

missing_Tc_increments [bool] Whether or not there were missing atoms for the *Tc* calculation, [-]

missing_Pc_increments [bool] Whether or not there were missing atoms for the *Pc* calculation, [-]

Notes

Raises an exception if rdkit is not installed, or *smi* or *rdkitmol* is not defined.

Calculated values were published in [3] for 448 compounds, as calculated by NIST TDE. There appear to be further modifications to the method in NIST TDE, as ~25% of values have differences larger than 5 K.

References

[1], [2], [3]

Examples

Example for 2-ethylphenol in [2]:

```
>>> Tc, Pc, _, _ = Wilson_Jasperson('CCC1=CC=CC=C1O', Tb=477.67)
>>> (Tc, Pc)
(693.567, 3743819.6667)
>>> Tc, Pc, _, _ = Wilson_Jasperson('CCC1=CC=CC=C1O', Tb=477.67, second_order=False)
>>> (Tc, Pc)
(702.883, 3794106.49)
```

CHAPTER

EIGHT

EXAMPLE USES OF THERMO

The following Jupyter notebooks show some of the many calculations that can be done with Thermo.

These problems often make use of fluids, ht, chemicals, and pint so make sure you have them installed! More details on the unit handling library can be found at fluids.units.

8.1 Working with Heat Transfer Fluids - Therminol LT

Heat transfer fluids are pure species or chemical mixtures with specially tailored properties that make them suitable for use in heat exchangers. Usually this means not fouling, requiring little heat transfer area because of a high heat capacity, thermal conductivity, and potentially high density and low flammability.

Therminol LT is a fluid chosen for the demonstration. It is in fact a pure chemical, 1,2-diethylbenzene.

The data comes from therminol itself, in the following PDF.

https://web.archive.org/web/20210615044602/https://www.therminol.com/sites/therminol/files/documents/TF-8726_Therminol_LT.pdf

[1]:
```
from fluids.core import C2K, F2K
from fluids.constants import R
import numpy as np
from chemicals import rho_to_Vm, Vm_to_rho, property_mass_to_molar, omega_definition,␣
↪simple_formula_parser, similarity_variable, molecular_weight
from thermo import (TDependentProperty, VaporPressure, VolumeLiquid,␣
↪ChemicalConstantsPackage, PropertyCorrelationsPackage,
                    HeatCapacityLiquid, HeatCapacityGas, ThermalConductivityLiquid,
                    ThermalConductivityGas, ViscosityGas, ViscosityLiquid,␣
↪EnthalpyVaporization,
                    SurfaceTension)
name = '1,2-diethylbenzene'
CAS = "25340-17-4"
formula = "C10H14"
atoms = simple_formula_parser(formula)
sv = similarity_variable(atoms)
MW = molecular_weight(atoms)

Tc = 377.0 + 273.15
Pc = 34.5e5
rhoc_mass = 298.0 # kg/m^3
Vc = rho_to_Vm(rhoc_mass, MW)
Zc = Pc*Vc/(R*Tc)
```

(continues on next page)

(continued from previous page)

```
Tm = C2K(-75.0)

Ts = [-73, -62, -51, -40, -29, -18, -7, 4, 16, 27, 38, 49, 60, 71, 82, 93, 104, 116, 127,
↪ 138, 149, 160, 171, 181, 182, 193, 204, 216, 227, 238, 249, 260, 271, 282, 293, 304,␣
↪316]
Ts = [C2K(v) for v in Ts]

Psats = [ 0.002, 0.006, 0.016, 0.038, 0.084, 0.175, 0.345, 0.649, 1.17, 2.02, 3.37, 5.43,
↪ 8.51, 13.0, 19.3, 28.1, 40.1, 56.1, 77.1, 101, 104, 139, 183, 237, 304, 386, 484, 601,
↪ 740, 904, 1090, 1310, 1570]
Psats = [v*1e3 for v in Psats] # kPa to Pa
Ts_Psats = Ts[len(Ts)-len(Psats):]

# Obtain the acentric factor from linear interpolation for convenience
Psat_07 = float(np.interp(0.7*Tc, Ts_Psats, Psats))
omega = omega_definition(Psat_07, Pc)

# Interpolate on pressure to find the normal boiling point
Tb = float(np.interp(101325.0, Psats, Ts_Psats))

rhols_mass = [938, 930, 921, 913, 904, 896, 887, 878, 869, 860, 852, 843, 833, 824, 815,␣
↪806, 796, 786, 776, 766, 756, 746, 735, 726, 724, 713, 702, 690, 678, 666, 652, 639,␣
↪625, 610, 594, 576, 558]
Vms = [rho_to_Vm(rho, MW) for rho in rhols_mass]

Cpls_mass = [1.44, 1.48, 1.53, 1.57, 1.61, 1.66, 1.70, 1.74, 1.78, 1.83, 1.87, 1.91, 1.
↪95, 1.99, 2.03, 2.07, 2.11, 2.15, 2.19, 2.23, 2.27, 2.30, 2.34, 2.38, 2.38, 2.42, 2.46,
↪ 2.50, 2.54, 2.58, 2.63, 2.67, 2.72, 2.78, 2.84, 2.91, 3.01]
Cpls_mass = [v*1000 for v in Cpls_mass] # kJ/(kg*K)
Cpls = [property_mass_to_molar(Cp, MW) for Cp in Cpls_mass] # J/(mol*K)

Cpgs_mass = [0.766, 0.813, 0.860, 0.908, 0.955, 1.002, 1.049, 1.095, 1.142, 1.188, 1.234,
↪ 1.280, 1.325, 1.370, 1.415, 1.459, 1.503, 1.547, 1.590, 1.634, 1.676, 1.719, 1.761, 1.
↪799, 1.803, 1.845, 1.886, 1.928, 1.970, 2.012, 2.055, 2.099, 2.144, 2.191, 2.241, 2.
↪297, 2.362]
Cpgs_mass = [v*1000 for v in Cpgs_mass] # kJ/(kg*K)
Cpgs = [property_mass_to_molar(Cp, MW) for Cp in Cpgs_mass] # J/(mol*K)


kls = [0.1426, 0.1405, 0.1384, 0.1362, 0.1341, 0.1320, 0.1298, 0.1277, 0.1255, 0.1233, 0.
↪1212, 0.1190, 0.1168, 0.1146, 0.1124, 0.1102, 0.1080, 0.1058, 0.1036, 0.1013, 0.0991,␣
↪0.0968, 0.0946, 0.0926, 0.0923, 0.0901, 0.0878, 0.0855, 0.0832, 0.0810, 0.0786, 0.0763,
↪ 0.0740, 0.0717, 0.0694, 0.0670, 0.0647]
muls = [10.09, 6.03, 3.99, 2.84, 2.13, 1.67, 1.35, 1.12, 0.947, 0.814, 0.708, 0.624, 0.
↪554, 0.496, 0.447, 0.405, 0.369, 0.338, 0.310, 0.286, 0.265, 0.246, 0.229, 0.215, 0.
↪213, 0.199, 0.187, 0.175, 0.165, 0.155, 0.146, 0.138, 0.131, 0.124, 0.117, 0.112, 0.
↪106]
muls = [v*1e-3 for v in muls] # mPa*s to Pa*s

Hvaps_mass = [492.7, 485.2, 477.8, 470.4, 463.0, 455.7, 448.5, 441.3, 434.1, 427.0, 420.
↪0, 412.9, 405.9, 399.0, 392.1, 385.2, 378.4, 371.6, 364.7, 357.9, 351.0, 344.1, 337.2,␣
↪330.9, 330.1, 323.0, 315.7, 308.2, 300.5, 292.5, 284.3, 275.6, 266.4, 256.7, 246.2,␣
↪234.7, 221.8]
```

(continues on next page)

(continued from previous page)

```
Hvaps_mass = [v*1000 for v in Hvaps_mass] # kJ/(kg)
Hvaps = [property_mass_to_molar(Hvap, MW) for Hvap in Hvaps_mass] # J/(mol)

kgs = [ 0.0051, 0.0057, 0.0063, 0.0069, 0.0075, 0.0082, 0.0088, 0.0095, 0.0101, 0.0108,␣
↪0.0115, 0.0122, 0.0130, 0.0137, 0.0144, 0.0152, 0.0160, 0.0168, 0.0176, 0.0184, 0.0192,
↪ 0.0200, 0.0209, 0.0216, 0.0217, 0.0226, 0.0235, 0.0244, 0.0253, 0.0262, 0.0272, 0.
↪0281, 0.0291, 0.0301, 0.0310, 0.0321, 0.0331]

mugs = [0.00434, 0.00458, 0.00482, 0.00506, 0.00530, 0.00554, 0.00578, 0.00603, 0.00628,␣
↪0.00652, 0.00677, 0.00702, 0.00727, 0.00752, 0.00777, 0.00802, 0.00828, 0.00853, 0.
↪00878, 0.00903, 0.00928, 0.00952, 0.00977, 0.01000, 0.01002, 0.01027, 0.01051, 0.01076,
↪ 0.01100, 0.01124, 0.01148, 0.01172, 0.01196, 0.01220, 0.01243, 0.01267, 0.01290]
mugs = [v*1e-3 for v in mugs] # mPa*s to Pa*s

sigmas = [0.028, 0.0]
sigma_Ts = [298.15, Tc]

prop_kwargs = {'Tc': Tc, 'Pc': Pc, 'Vc': Vc, 'Zc': Zc, 'omega': omega,
               'MW': MW, 'Tb': Tb, 'Tm': Tm, 'CASRN': CAS}
prop_kwargs = {} # Comment this out to show the estimation method results

plot_kwargs = {'pts': 30, 'Tmin': Ts[0]}
```

Now that the data has been added into Python objects, we can fit them to equations. The plots below show how good the fits are.

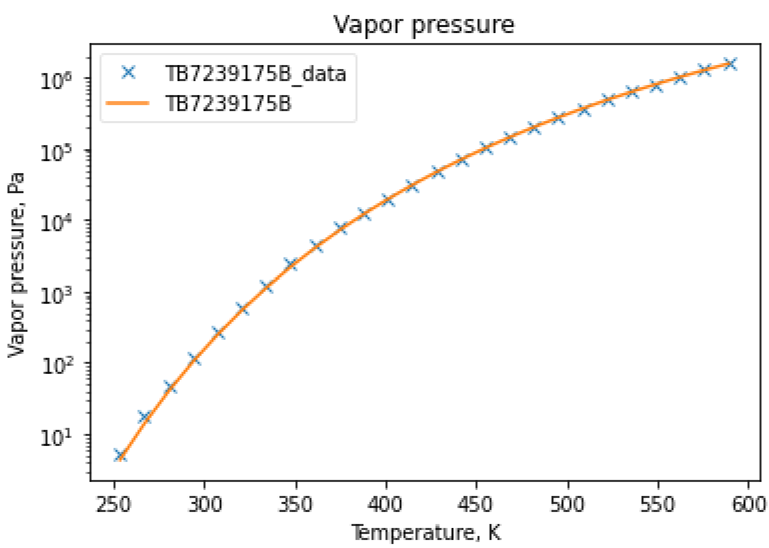
[2]:
```
source = 'TB7239175B'
plot = True
PsatObj = VaporPressure(**prop_kwargs)
PsatObj.fit_add_model(Ts=Ts_Psats, data=Psats, model='DIPPR101', name=source)

VolLiqObj = VolumeLiquid(**prop_kwargs)
VolLiqObj.fit_add_model(Ts=Ts, data=Vms, model='DIPPR100', name=source)

CpLiqObj = HeatCapacityLiquid(**prop_kwargs)
CpLiqObj.fit_add_model(Ts=Ts, data=Cpls, model='DIPPR100', name=source)

CpGasObj = HeatCapacityGas(**prop_kwargs)
CpGasObj.fit_add_model(Ts=Ts, data=Cpgs, model='DIPPR100', name=source)

MuLiqObj = ViscosityLiquid(**prop_kwargs)
MuLiqObj.fit_add_model(Ts=Ts, data=muls, model='mu_TDE', name=source)

MuGasObj = ViscosityGas(**prop_kwargs)
MuGasObj.fit_add_model(Ts=Ts, data=mugs, model='DIPPR100', name=source)

KGasObj = ThermalConductivityGas(**prop_kwargs)
KGasObj.fit_add_model(Ts=Ts, data=kgs, model='DIPPR100', name=source)

KLiqObj = ThermalConductivityLiquid(**prop_kwargs)
KLiqObj.fit_add_model(Ts=Ts, data=kls, model='DIPPR100', name=source)
```

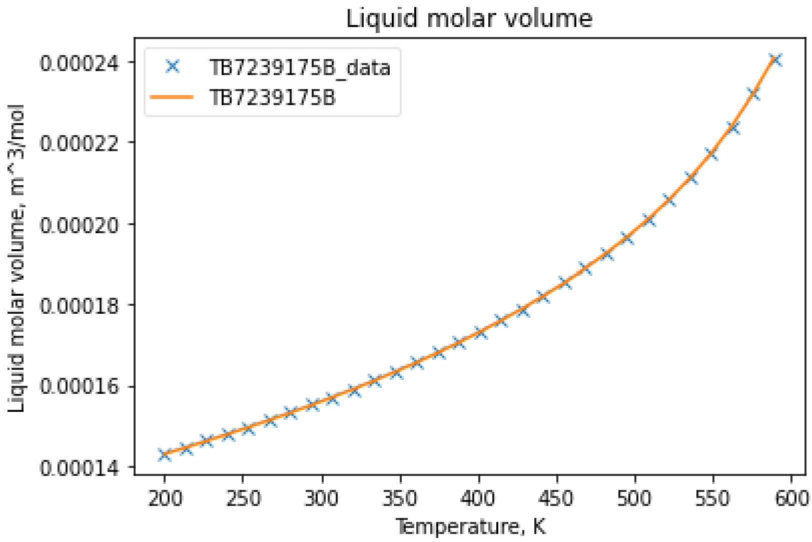
(continues on next page)

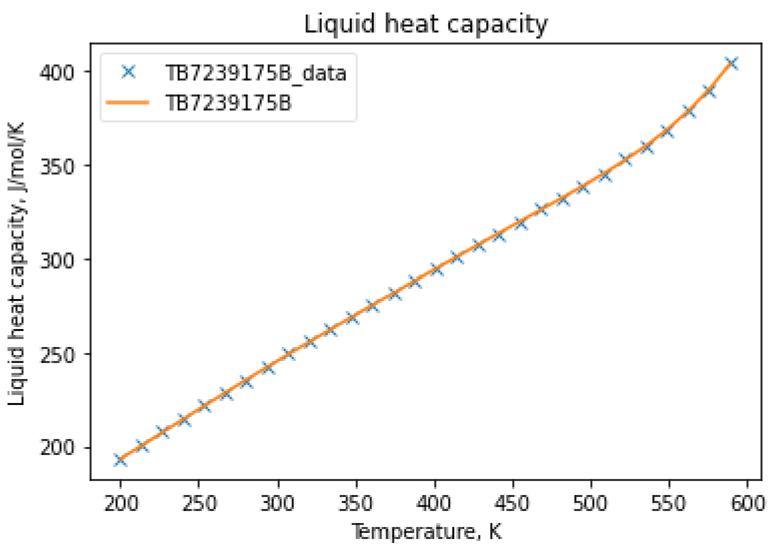
(continued from previous page)

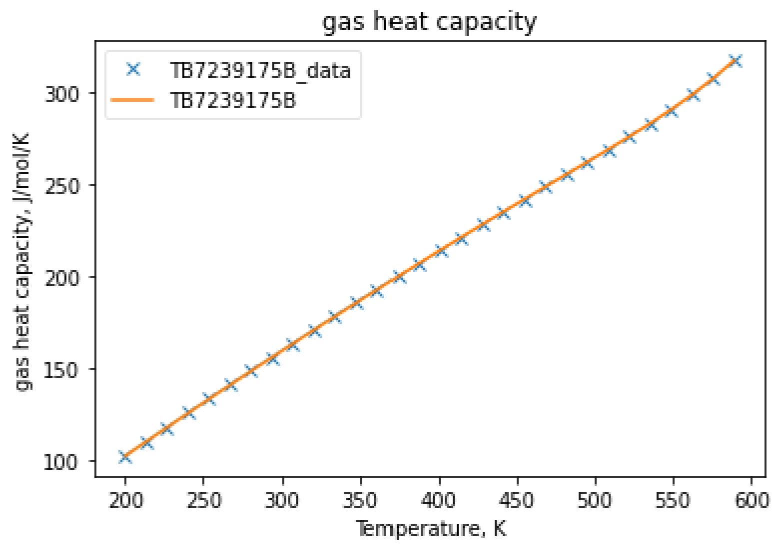
```
HvapObj = EnthalpyVaporization(**prop_kwargs)
HvapObj.fit_add_model(Ts=Ts, data=Hvaps, model_kwargs={'Tc': Tc}, model='PPDS12',␣
↪name=source)

SigmaObj = SurfaceTension(**prop_kwargs)
SigmaObj.fit_add_model(Ts=sigma_Ts, data=sigmas, model_kwargs={'Tc': Tc}, model='linear',
↪ name=source)

if plot:
    PsatObj.plot_T_dependent_property(axes='semilogy', **plot_kwargs)
    VolLiqObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    CpLiqObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    CpGasObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    MuLiqObj.plot_T_dependent_property(axes='semilogy', **plot_kwargs)
    MuGasObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    KGasObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    KLiqObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    HvapObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
    SigmaObj.plot_T_dependent_property(axes='plot', **plot_kwargs)
```

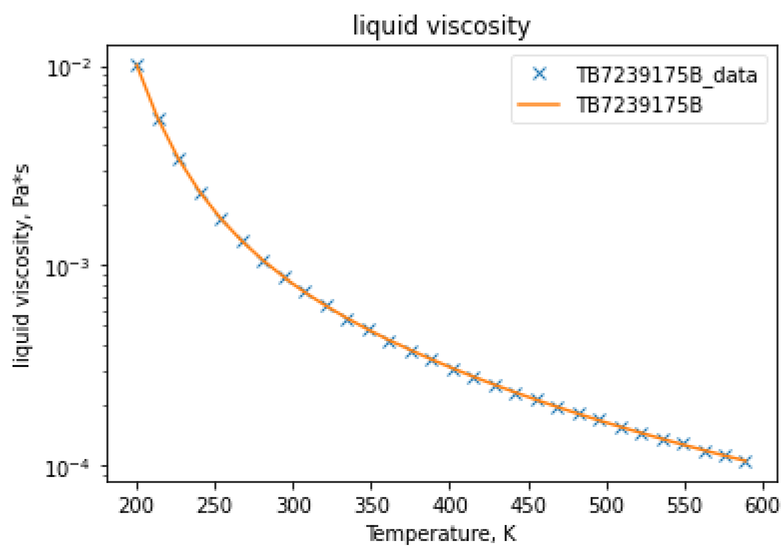
```
/home/caleb/.local/lib/python3.9/site-packages/scipy/optimize/minpack.py:475:␣
↪RuntimeWarning: Number of calls to function has reached maxfev = 500.
  warnings.warn(errors[info][0], RuntimeWarning)
```

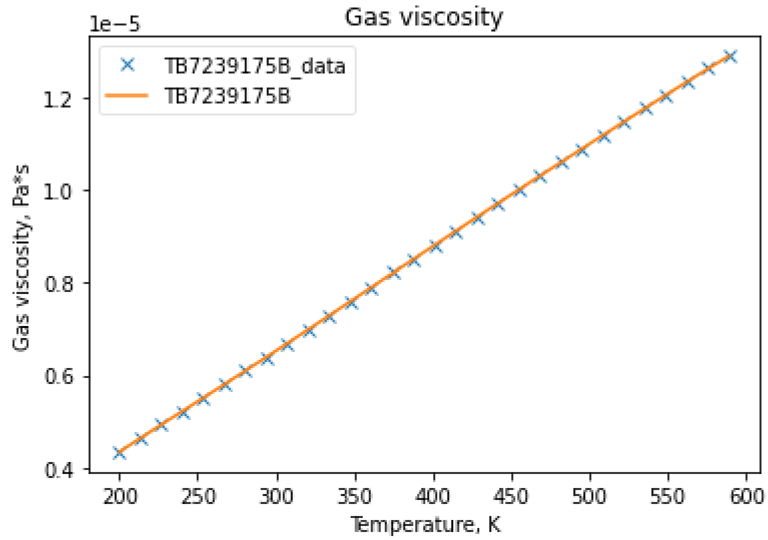


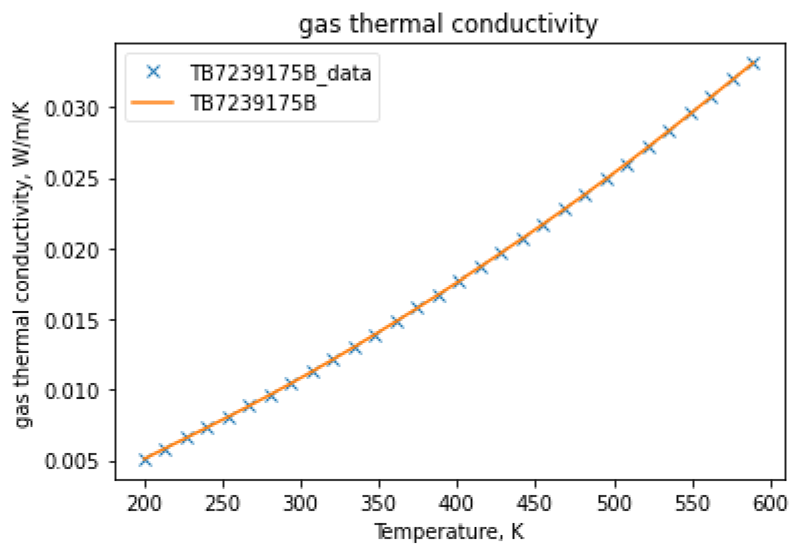

Liquid molar volume
TB7239175B_data
TB7239175B
Liquid molar volume, m^3/mol
0.00024
0.00022
0.00020
0.00018
0.00016
0.00014
200
250
300
350
400
450
500
550
600
Temperature, K

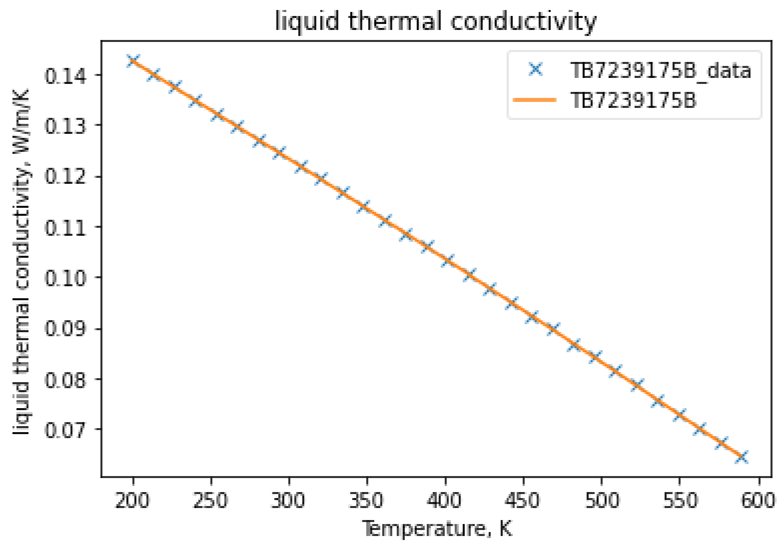

Liquid heat capacity
TB7239175B_data
TB7239175B
Liquid heat capacity, J/mol/K
400
350
300
250
200
200
250
300
350
400
450
500
550
600
Temperature, K

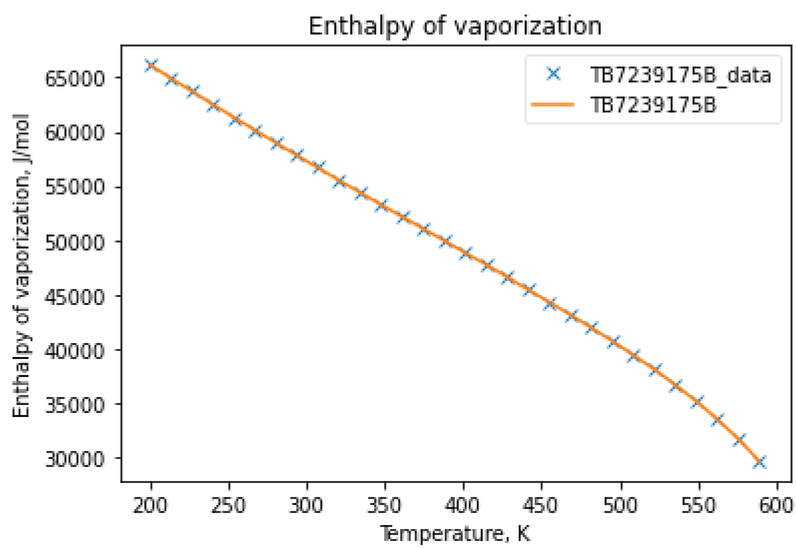

gas heat capacity
TB7239175B_data
TB7239175B
gas heat capacity, J/mol/K
Temperature, K

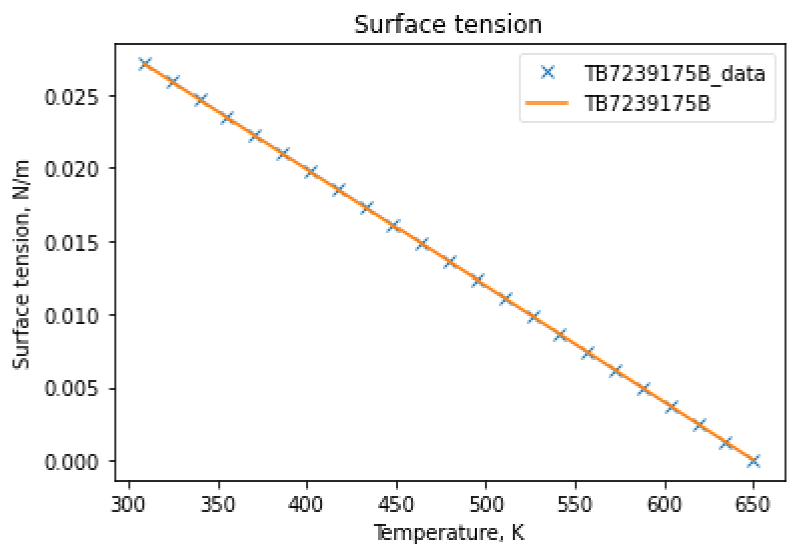

liquid viscosity
TB7239175B_data
TB7239175B
liquid viscosity, Pa*s
Temperature, K


Gas viscosity
1e−5
TB7239175B_data
TB7239175B
Gas viscosity, Pa*s
Temperature, K


gas thermal conductivity
TB7239175B_data
TB7239175B
gas thermal conductivity, W/m/K
Temperature, K


liquid thermal conductivity
TB7239175B_data
TB7239175B
liquid thermal conductivity, W/m/K
0.14
0.13
0.12
0.11
0.10
0.09
0.08
0.07
200
250
300
350
400
450
500
550
600
Temperature, K


Enthalpy of vaporization
TB7239175B_data
TB7239175B
Enthalpy of vaporization, J/mol
65000
60000
55000
50000
45000
40000
35000
30000
200
250
300
350
400
450
500
550
600
Temperature, K



[3]:
```
Vml_60F = VolLiqObj(F2K(60), None)
rhol_60Fs_mass = Vm_to_rho(Vml_60F, MW)

Vml_STP = VolLiqObj(298.15, None)
rhol_STPs_mass = Vm_to_rho(Vml_STP, MW)

constants = ChemicalConstantsPackage(Tcs=[Tc], Pcs=[Pc], Vcs=[Vc], Zcs=[Zc],␣
↪omegas=[omega], MWs=[MW],
                                     Vml_60Fs=[Vml_60F], rhol_60Fs=[1/Vml_60F], rhol_
↪60Fs_mass=[rhol_60Fs_mass],
                                     Vml_STPs=[Vml_STP], rhol_STPs_mass=[rhol_STPs_mass],
                                     similarity_variables=[sv])
correlations = PropertyCorrelationsPackage(constants=constants, VaporPressures=[PsatObj],
↪ VolumeLiquids=[VolLiqObj],
                                        HeatCapacityLiquids=[CpLiqObj],␣
↪HeatCapacityGases=[CpGasObj],
                                        ViscosityLiquids=[MuLiqObj],␣
↪ViscosityGases=[MuGasObj],
                                        ThermalConductivityGases=[KGasObj],␣
↪ThermalConductivityLiquids=[KLiqObj],
                                        EnthalpyVaporizations=[HvapObj],␣
↪SurfaceTensions=[SigmaObj])
```

Now that the ChemicalConstantsPackage and PropertyCorrelationsPackage have been created, we are ready to make packages and do calculations with them.

[4]:
```
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CEOSGas, FlashPureVLS
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)

liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher_PR = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])
```

(continues on next page)

(continued from previous page)

```
print(flasher_PR.flash(T=300, P=1e5))
```

```
<EquilibriumState, T=300.0000, P=100000.0000, zs=[1.0], betas=[1.0], phases=[<CEOSLiquid,
↪ T=300 K, P=100000 Pa>]>
```

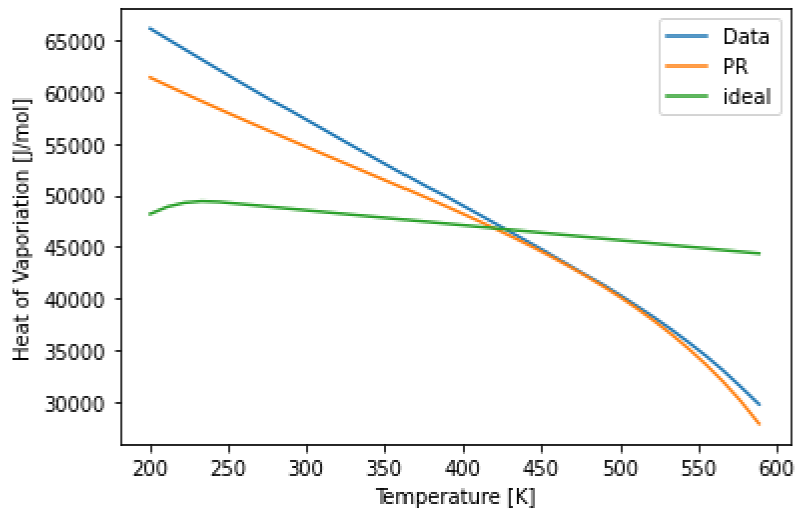
[5]:
```
from thermo.phases import GibbsExcessLiquid, IdealGas
liquid = GibbsExcessLiquid(VaporPressures=correlations.VaporPressures,␣
↪VolumeLiquids=correlations.VolumeLiquids,
                 HeatCapacityGases=correlations.HeatCapacityGases, equilibrium_basis=
↪'Psat')
gas = IdealGas(HeatCapacityGases=correlations.HeatCapacityGases)
flasher_ideal = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid],␣
↪solids=[])
print(flasher_ideal.flash(T=300, P=1e5))
```

```
<EquilibriumState, T=300.0000, P=100000.0000, zs=[1.0], betas=[1.0], phases=[
↪<GibbsExcessLiquid, T=300 K, P=100000 Pa>]>
```

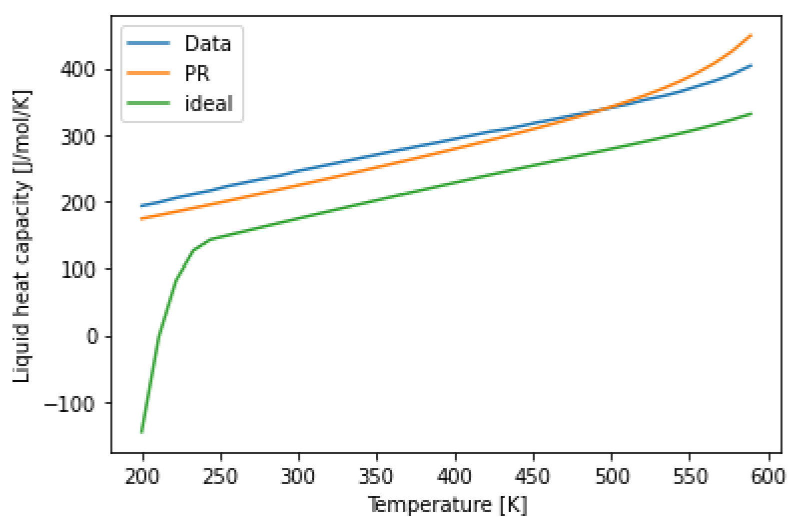
Using a thermodynamically consistent model is much more challenging than directly predicting a property. Liquid heat capacity, heat of vaporization, and density are all particularly challenging properties. The following plots show the accuracy of the models.

[9]:
```
Cpls_calc_PR = []
Cpls_calc_ideal = []
for T in Ts:
    Cpls_calc_PR.append(flasher_PR.flash(T=T, VF=0).Cp())
    Cpls_calc_ideal.append(flasher_ideal.flash(T=T, VF=0).Cp())

Hvaps_calc_PR = []
Hvaps_calc_ideal = []
for T in Ts:
    Hvaps_calc_PR.append(flasher_PR.flash(T=T, VF=1).H() - flasher_PR.flash(T=T, VF=0).
↪H())
    Hvaps_calc_ideal.append(flasher_ideal.flash(T=T, VF=1).H() - flasher_ideal.flash(T=T,
↪ VF=0).H())

Vl_calc_PR = []
Vl_calc_ideal = []
for T in Ts:
    Vl_calc_PR.append(flasher_PR.flash(T=T, VF=0).V())
    Vl_calc_ideal.append(flasher_ideal.flash(T=T, VF=0).V())
```

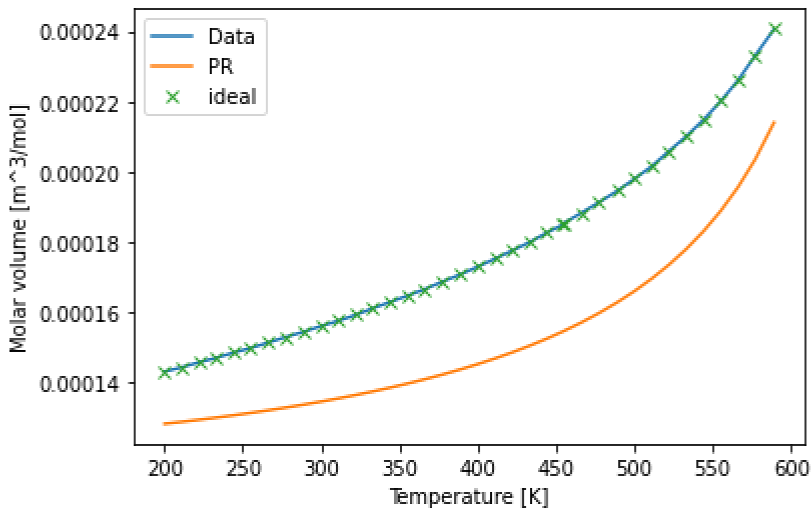
[7]:
```
import matplotlib.pyplot as plt
plt.plot(Ts, Hvaps, label='Data')
plt.plot(Ts, Hvaps_calc_PR, label='PR')
plt.plot(Ts, Hvaps_calc_ideal, label='ideal')
plt.xlabel("Temperature [K]")
plt.ylabel("Heat of Vaporiation [J/mol]")
plt.legend()
plt.show()
```




```
[8]: import matplotlib.pyplot as plt
plt.plot(Ts, Cpls, label='Data')
plt.plot(Ts, Cpls_calc_PR, label='PR')
plt.plot(Ts, Cpls_calc_ideal, label='ideal')
plt.xlabel("Temperature [K]")
plt.ylabel("Liquid heat capacity [J/mol/K]")
plt.legend()
plt.show()
```




```
[11]: import matplotlib.pyplot as plt
plt.plot(Ts, Vms, label='Data')
plt.plot(Ts, Vl_calc_PR, label='PR')
plt.plot(Ts, Vl_calc_ideal, 'x', label='ideal')
plt.xlabel("Temperature [K]")
plt.ylabel("Molar volume [m^3/mol]")
plt.legend()
plt.show()
```



8.2 Validating Flash Calculations

Finding the solution to multiphase equilibrium calculations is challenging and the topic of continuing research. Many commercial packages offer users a great deal of confidence in their answers, but can they be trusted? Thermo can be used to validate the results from other software or identify defects in them.

The following example uses a natural gas mixture two pseudocomponents C7-C16 and C17+. The properties of pure components are taken from Thermo. To do a perfect comparison, the critical properties from other software packages should be substituted into Thermo. This is example S3 from Fonseca-Pérez (2021). The kijs are from Harding and Floudas (2000), and the original pseudocomponents are from Nagarajan, Cullick, and Griewank (1991).

Fonseca-Pérez, R. M., A. Bonilla-Petriciolet, J. C. Tapia-Picazo, and J. E. Jaime-Leal. "A Reconsideration on the Resolution of Phase Stability Analysis Using Stochastic Global Optimization Methods: Proposal of a Reliable Set of Benchmark Problems." Fluid Phase Equilibria 548 (November 15, 2021): 113180. https://doi.org/10.1016/j.fluid.2021.113180.

Harding, S. T., and C. A. Floudas. "Phase Stability with Cubic Equations of State: Global Optimization Approach." AIChE Journal 46, no. 7 (July 1, 2000): 1422–40. https://doi.org/10.1002/aic.690460715.

Nagarajan, N. R., A. S. Cullick, and A. Griewank. "New Strategy for Phase Equilibrium and Critical Point Calculations by Thermodynamic Energy Analysis. Part I. Stability Analysis and Flash." Fluid Phase Equilibria 62, no. 3 (January 1, 1991): 191–210. https://doi.org/10.1016/0378-3812(91)80010-S.

[39]:
```
from thermo import *
from scipy.constants import atm
pure_constants = ChemicalConstantsPackage.constants_from_IDs(
    ['methane', 'ethane', 'propane', 'n-butane', 'n-pentane', 'n-hexane'])

pseudos = ChemicalConstantsPackage(Tcs=[606.28,825.67], Pcs=[25.42*atm, 14.39*atm],
                                   omegas=[0.4019, 0.7987], MWs=[140.0, 325.0])
constants = pure_constants + pseudos

properties = PropertyCorrelationsPackage(constants=constants)

T = 353
```

(continues on next page)

(continued from previous page)

```
P = 38500e3
zs = [0.7212, 0.09205, 0.04455, 0.03123, 0.01273, 0.01361, 0.07215, 0.01248]

kijs = [[0.0, 0.002, 0.017, 0.015, 0.02, 0.039, 0.05, 0.09],
[0.002, 0.0, 0.0, 0.025, 0.01, 0.056, 0.04, 0.055],
[0.017, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01, 0.01],
[0.015, 0.025, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.02, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.039, 0.056, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.05, 0.04, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.09, 0.055, 0.01, 0.0, 0.0, 0.0, 0.0, 0.0]]

eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas,␣
↪kijs=kijs)

gas = CEOSGas(PRMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases, T=T,␣
↪P=P, zs=zs)
liq = CEOSLiquid(PRMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases, T=T,␣
↪P=P, zs=zs)
liq2 = CEOSLiquid(PRMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases, T=T,
↪ P=P, zs=zs)
phase_list = [gas, liq, liq]


flashN = FlashVLN(constants, properties, liquids=[liq, liq2], gas=gas)
# flashN.PT_SS_TOL = 1e-18
res = flashN.flash(T=T, P=P, zs=zs)
print('There are %s phases present' %(res.phase_count))
print('Mass densities of each liquid are %f and %f kg/m^3' %(res.liquid0.rho_mass(),␣
↪res.liquid0.rho_mass()))
```

```
There are 2 phases present
Mass densities of each liquid are 527.867861 and 527.867861 kg/m^3
```

[45]:
```
import numpy as np
max_fugacity_err = np.max(np.abs(1-np.array(res.liquid0.fugacities())/res.liquid1.
↪fugacities()))
print('The maximum relative difference in fugacity is %e.' %(max_fugacity_err,))
```

```
The maximum relative difference in fugacity is 2.773985e-07.
```

8.3 High Molecular Weight Petroleum Pseudocomponents

Thermo is a general phase equilibrium engine, and if the user provides enough properties for the components, there is no issue adding your own components. In this basic example below, a made-up extended gas analysis is used to specify a gas consisting of the standard real components and three heavier fractions, C10+, C12+ and C15+.

A bare minimum of basic properties are estimated using the Kesler-Lee method (1976), and the estimated fraction molecular weights are turned into atomic compositions. The heat capacities of each pseudocomponent is found with the similarity variable concept of Lastovka and Shaw (2013) based on atomic composition.

This example ends with calculating a flash at 270 Kelvin and 1 bar.

```
[72]: from math import log, exp
import numpy as np
from scipy.constants import psi
from thermo import *
from chemicals import *

def Tc_Kesler_Lee_SG_Tb(SG, Tb):
    r'''Estimates critical temperature of a hydrocarbon compound or petroleum
    fraction using only its specific gravity and boiling point, from
    [1]_ as presented in [2]_.

    .. math::
        T_c = 341.7 + 811.1SG + [0.4244 + 0.1174SG]T_b
        + \frac{[0.4669 - 3.26238SG]10^5}{T_b}

    Parameters
    ----------
    SG : float
        Specific gravity of the fluid at 60 degrees Farenheight [-]
    Tb : float
        Boiling point the fluid [K]

    Returns
    -------
    Tc : float
        Estimated critical temperature [K]

    Notes
    -----
    Model shows predictions for Tc, Pc, MW, and omega.
    Original units in degrees Rankine.

    Examples
    --------
    Example 2.2 from [2]_, but with K instead of R.

    >>> Tc_Kesler_Lee_SG_Tb(0.7365, 365.555)
    545.0124354151242

    References
    ----------
    .. [1] Kesler, M. G., and B. I. Lee. "Improve Prediction of Enthalpy of
       Fractions." Hydrocarbon Processing (March 1976): 153-158.
    .. [2] Ahmed, Tarek H. Equations of State and PVT Analysis: Applications
       for Improved Reservoir Modeling. Gulf Pub., 2007.
    '''
    Tb = 9/5.*Tb # K to R
    Tc = 341.7 + 811.1*SG + (0.4244 + 0.1174*SG)*Tb + ((0.4669 - 3.26238*SG)*1E5)/Tb
    Tc = 5/9.*Tc # R to K
    return Tc

def Pc_Kesler_Lee_SG_Tb(SG, Tb):
    r'''Estimates critical pressure of a hydrocarbon compound or petroleum
```

(continues on next page)

(continued from previous page)

```
    fraction using only its specific gravity and boiling point, from
    [1]_ as presented in [2]_.

    .. math::
        \ln(P_c) = 8.3634 - \frac{0.0566}{SG} - \left[0.24244 + \frac{2.2898}
        {SG} + \frac{0.11857}{SG^2}\right]10^{-3}T_b
        + \left[1.4685 + \frac{3.648}{SG} + \frac{0.47227}{SG^2}\right]
        10^{-7}T_b^2-\left[0.42019 + \frac{1.6977}{SG^2}\right]10^{-10}T_b^3

    Parameters
    ----------
    SG : float
        Specific gravity of the fluid at 60 degrees Farenheight [-]
    Tb : float
        Boiling point the fluid [K]

    Returns
    -------
    Pc : float
        Estimated critical pressure [Pa]

    Notes
    -----
    Model shows predictions for Tc, Pc, MW, and omega.
    Original units in degrees Rankine and psi.

    Examples
    --------
    Example 2.2 from [2]_, but with K instead of R and Pa instead of psi.

    >>> Pc_Kesler_Lee_SG_Tb(0.7365, 365.555)
    3238323.346840464

    References
    ----------
    .. [1] Kesler, M. G., and B. I. Lee. "Improve Prediction of Enthalpy of
       Fractions." Hydrocarbon Processing (March 1976): 153-158.
    .. [2] Ahmed, Tarek H. Equations of State and PVT Analysis: Applications
       for Improved Reservoir Modeling. Gulf Pub., 2007.
    '''
    Tb = 9/5.*Tb # K to R
    Pc = exp(8.3634 - 0.0566/SG - (0.24244 + 2.2898/SG + 0.11857/SG**2)*1E-3*Tb
    + (1.4685 + 3.648/SG + 0.47227/SG**2)*1E-7*Tb**2
    -(0.42019 + 1.6977/SG**2)*1E-10*Tb**3)
    Pc = Pc*psi
    return Pc

def MW_Kesler_Lee_SG_Tb(SG, Tb):
    r'''Estimates molecular weight of a hydrocarbon compound or petroleum
    fraction using only its specific gravity and boiling point, from
    [1]_ as presented in [2]_.
```

(continues on next page)

(continued from previous page)

```
    .. math::
        MW = -12272.6 + 9486.4SG + [4.6523 - 3.3287SG]T_b + [1-0.77084SG
        - 0.02058SG^2]\left[1.3437 - \frac{720.79}{T_b}\right]\frac{10^7}{T_b}
        + [1-0.80882SG + 0.02226SG^2][1.8828 - \frac{181.98}{T_b}]
        \frac{10^{12}}{T_b^3}

    Parameters
    ----------
    SG : float
        Specific gravity of the fluid at 60 degrees Farenheight [-]
    Tb : float
        Boiling point the fluid [K]

    Returns
    -------
    MW : float
        Estimated molecular weight [g/mol]

    Notes
    -----
    Model shows predictions for Tc, Pc, MW, and omega.
    Original units in degrees Rankine.

    Examples
    --------
    Example 2.2 from [2]_, but with K instead of R and Pa instead of psi.

    >>> MW_Kesler_Lee_SG_Tb(0.7365, 365.555)
    98.70887589833501

    References
    ----------
    .. [1] Kesler, M. G., and B. I. Lee. "Improve Prediction of Enthalpy of
       Fractions." Hydrocarbon Processing (March 1976): 153-158.
    .. [2] Ahmed, Tarek H. Equations of State and PVT Analysis: Applications
       for Improved Reservoir Modeling. Gulf Pub., 2007.
    '''
    Tb = 9/5.*Tb # K to R
    MW = (-12272.6 + 9486.4*SG + (4.6523 - 3.3287*SG)*Tb + (1.-0.77084*SG - 0.
↪02058*SG**2)*
    (1.3437 - 720.79/Tb)*1E7/Tb + (1.-0.80882*SG + 0.02226*SG**2)*
    (1.8828 - 181.98/Tb)*1E12/Tb**3)
    return MW

def omega_Kesler_Lee_SG_Tb_Tc_Pc(SG, Tb, Tc=None, Pc=None):
    r'''Estimates accentric factor of a hydrocarbon compound or petroleum
    fraction using only its specific gravity and boiling point, from
    [1]_ as presented in [2]_. If Tc and Pc are provided, the Kesler-Lee
    routines for estimating them are not used.

    For Tbr > 0.8:
    .. math::
```

(continues on next page)

(continued from previous page)

```
        \omega = -7.904 + 0.1352K - 0.007465K^2 + 8.359T_{br}
        + ([1.408-0.01063K]/T_{br})

    Otherwise:
    .. math::
        \omega = \frac{-\ln\frac{P_c}{14.7} - 5.92714 + \frac{6.09648}{T_{br}}
        + 1.28862\ln T_{br} - 0.169347T_{br}^6}{15.2518 - \frac{15.6875}{T_{br}}
         - 13.4721\ln T_{br} + 0.43577T_{br}^6}

        K = \frac{T_b^{1/3}}{SG}

        T_{br} = \frac{T_b}{T_c}

    Parameters
    ----------
    SG : float
        Specific gravity of the fluid at 60 degrees Farenheight [-]
    Tb : float
        Boiling point the fluid [K]
    Tc : float, optional
        Estimated critical temperature [K]
    Pc : float, optional
        Estimated critical pressure [Pa]

    Returns
    -------
    omega : float
        Acentric factor [-]

    Notes
    -----
    Model shows predictions for Tc, Pc, MW, and omega.
    Original units in degrees Rankine and psi.

    Examples
    --------
    Example 2.2 from [2]_, but with K instead of R and Pa instead of psi.

    >>> omega_Kesler_Lee_SG_Tb_Tc_Pc(0.7365, 365.555, 545.012, 3238323.)
    0.306392118159797

    References
    ----------
    .. [1] Kesler, M. G., and B. I. Lee. "Improve Prediction of Enthalpy of
       Fractions." Hydrocarbon Processing (March 1976): 153-158.
    .. [2] Ahmed, Tarek H. Equations of State and PVT Analysis: Applications
       for Improved Reservoir Modeling. Gulf Pub., 2007.
    '''
    if Tc is None:
        Tc = Tc_Kesler_Lee_SG_Tb(SG, Tb)
    if Pc is None:
        Pc = Pc_Kesler_Lee_SG_Tb(SG, Tb)
```

(continues on next page)

(continued from previous page)

```
    Tb = 9/5.*Tb # K to R
    Tc = 9/5.*Tc # K to R
    K = Tb**(1/3.)/SG
    Tbr = Tb/Tc
    if Tbr > 0.8:
        omega = -7.904 + 0.1352*K - 0.007465*K**2 + 8.359*Tbr + ((1.408-0.01063*K)/Tbr)
    else:
        omega = ((-log(Pc/101325.) - 5.92714 + 6.09648/Tbr + 1.28862*log(Tbr)
        - 0.169347*Tbr**6) / (15.2518 - 15.6875/Tbr - 13.4721*log(Tbr) +0.43577*Tbr**6))
    return omega
```

[112]:
```
# Basic composition and names. All pure component properties are obtained from Chemicals␣
↪and Thermo.
pure_constants = ChemicalConstantsPackage.constants_from_IDs(
    ['water', 'hydrogen', 'helium', 'nitrogen', 'carbon dioxide', 'hydrogen sulfide',
↪'methane',
    'ethane', 'propane', 'isobutane', 'n-butane', 'isopentane', 'n-pentane', 'hexane',
    'heptane', 'octane', 'nonane'])
pure_fractions = [.02, .00005, .00018, .009, .02, .002, .82, .08, .031,
                  .009, .0035, .0033, .0003, .0007, .0004, .00005, .00002]
```

[105]:
```
pseudo_names = ['C10-C11', 'C12-C14', 'C15+']
pseudo_carbon_numbers = [10.35, 12.5, 16.9]
pseudo_SGs = [.73, .76, .775] # Specific gravity values are based of the alkane series
pseudo_Tbs = [447, 526, 589]

# Using the estimation methods defined earlier, we obtain some critical properties
pseudo_Tcs = [Tc_Kesler_Lee_SG_Tb(SG, Tb) for SG, Tb in zip(pseudo_SGs, pseudo_Tbs)]
pseudo_Pcs = [Pc_Kesler_Lee_SG_Tb(SG, Tb) for SG, Tb in zip(pseudo_SGs, pseudo_Tbs)]
pseudo_MWs = [MW_Kesler_Lee_SG_Tb(SG, Tb) for SG, Tb in zip(pseudo_SGs, pseudo_Tbs)]
pseudo_omegas = [omega_Kesler_Lee_SG_Tb_Tc_Pc(SG, Tb) for SG, Tb in zip(pseudo_SGs,␣
↪pseudo_Tbs)]
# Estimate the hydroen counts
hydrogen_counts = [(MW - C*periodic_table.C.MW)/periodic_table.H.MW
                   for C, MW in zip(pseudo_carbon_numbers, pseudo_MWs)]
# Get the atomic compositions
pseudo_atoms = [{'C': C, 'H': H} for C, H in zip(pseudo_carbon_numbers, hydrogen_counts)]
# Calculate the similarity variable of each species
similarity_variables = [similarity_variable(atoms=atoms) for atoms in pseudo_atoms]

pseudo_fractions = [.0003, .00015, .00005]
```

[113]:
```
pseudos = ChemicalConstantsPackage(names=pseudo_names, MWs=pseudo_MWs, Tbs=pseudo_Tbs,
                                   atomss=pseudo_atoms,
                                  Tcs=pseudo_Tcs, Pcs=pseudo_Pcs, omegas=pseudo_omegas,
                                  similarity_variables=similarity_variables)
# Add the pure components and the pseudocomponents to create a new package of constant␣
↪values
# which will be used by the phase and flash objects
constants = pure_constants + pseudos
```

(continues on next page)

(continued from previous page)

```
# Obtain the temperature and pressure dependent objects
properties = PropertyCorrelationsPackage(constants=constants)
# This is the feed composition
zs = normalize(pure_fractions + pseudo_fractions)
T = 270 # K
P = 1e5 # bar
```

[132]:
```
kijs = np.zeros((constants.N, constants.N)).tolist() # kijs left as zero in this example
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas,␣
↪kijs=kijs)

# The API SRK equation of state is used, but other cubic equations of state can be uesd␣
↪instead
gas = CEOSGas(APISRKMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases, T=T,
↪ P=P, zs=zs)
liq = CEOSLiquid(APISRKMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases,␣
↪T=T, P=P, zs=zs)
liq2 = CEOSLiquid(APISRKMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases,␣
↪T=T, P=P, zs=zs)
phase_list = [gas, liq, liq]

# Set up the three phase flash engine
flashN = FlashVLN(constants, properties, liquids=[liq, liq2], gas=gas)
```

[133]:
```
# Do the flash, and get some properties
res = flashN.flash(T=T, P=P, zs=zs)
res.phase_count, res.gas_beta, res.liquids_betas
```

[133]:
```
(3, 0.9827041561275568, [0.01683884003998437, 0.0004570038324588659])
```

[134]:
```
res.H(), res.Cp_mass(), res.MW(), res.gas.mu(), res.gas.k()
```

[134]:
```
(-1961.508963322489,
 1989.3915447041693,
 19.675910651652533,
 1.0011888443404098e-05,
 0.027073401138714016)
```

[135]:
```
res.heaviest_liquid.rho_mass(), res.lightest_liquid.rho_mass()
```

[135]:
```
(769.2525386053419, 599.2086838769083)
```

8.4 Performing Large Numbers of Calculations with Thermo in Parallel

A common request is to obtain a large number of properties from Thermo at once. Thermo is not NumPy - it cannot just automatically do all of the calculations in parallel.

If you have a specific property that does not require phase equilibrium calculations to obtain, it is possible to use the `chemicals.numba` interface to in your own numba-accelerated code. https://chemicals.readthedocs.io/chemicals.numba.html

For those cases where lots of flashes are needed, your best bet is to brute force it - use multiprocessing (and maybe a beefy machine) to obtain the results faster. The following code sample uses `joblib` to facilitate the calculation. Note that joblib won't show any benefits on sub-second calculations. Also note that the `threading` backend of joblib will not offer any performance improvements due to the CPython GIL.

[1]:
```
import numpy as np
from thermo import *
from chemicals import *

constants, properties = ChemicalConstantsPackage.from_IDs(
    ['methane', 'ethane', 'propane', 'isobutane', 'n-butane', 'isopentane',
     'n-pentane', 'hexane', 'heptane', 'octane', 'nonane', 'nitrogen'])
T, P = 200, 5e6
zs = [.8, .08, .032, .00963, .0035, .0034, .0003, .0007, .0004, .00005, .00002, .07]
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
gas = CEOSGas(SRKMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases, T=T,␣
↪P=P, zs=zs)
liq = CEOSLiquid(SRKMIX, eos_kwargs, HeatCapacityGases=properties.HeatCapacityGases, T=T,
↪ P=P, zs=zs)
# Set up a two-phase flash engine, ignoring kijs
flasher = FlashVL(constants, properties, liquid=liq, gas=gas)

# Set a composition - it could be modified in the inner loop as well
# Do a test flash
flasher.flash(T=T, P=P, zs=zs).gas_beta
```

[1]:
```
0.4595970727935113
```

[2]:
```
def get_properties(T, P):
    # This is the function that will be called in parallel
    # note that Python floats are faster than numpy floats
    res = flasher.flash(T=float(T), P=float(P), zs=zs)
    return [res.rho_mass(), res.Cp_mass(), res.gas_beta]
```

[3]:
```
from joblib import Parallel, delayed
pts = 30
Ts = np.linspace(200, 400, pts)
Ps = np.linspace(1e5, 1e7, pts)
Ts_grid, Ps_grid = np.meshgrid(Ts, Ps)
# processed_data = Parallel(n_jobs=16)(delayed(get_properties)(T, P) for T, P in zip(Ts_
↪grid.flat, Ps_grid.flat))
```

[4]:
```
# Naive loop in Python
%timeit -r 1 -n 1 processed_data = [get_properties(T, P) for T, P in zip(Ts_grid.flat,␣
↪Ps_grid.flat)]
```

```
15.3 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[5]:
```
# Use the threading feature of Joblib
# Because the calculation is CPU-bound, the threads do not improve speed and Joblib's␣
↪overhead slows down the calculation
%timeit -r 1 -n 1 processed_data = Parallel(n_jobs=16, prefer="threads")(delayed(get_
↪properties)(T, P) for T, P in zip(Ts_grid.flat, Ps_grid.flat))
```

```
43.9 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[7]:
```
# Use the multiprocessing feature of joblib
# We were able to improve the speed by 5x
%timeit -r 1 -n 1 processed_data = Parallel(n_jobs=16, batch_size=30)(delayed(get_
↪properties)(T, P) for T, P in zip(Ts_grid.flat, Ps_grid.flat))
```

```
3.55 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

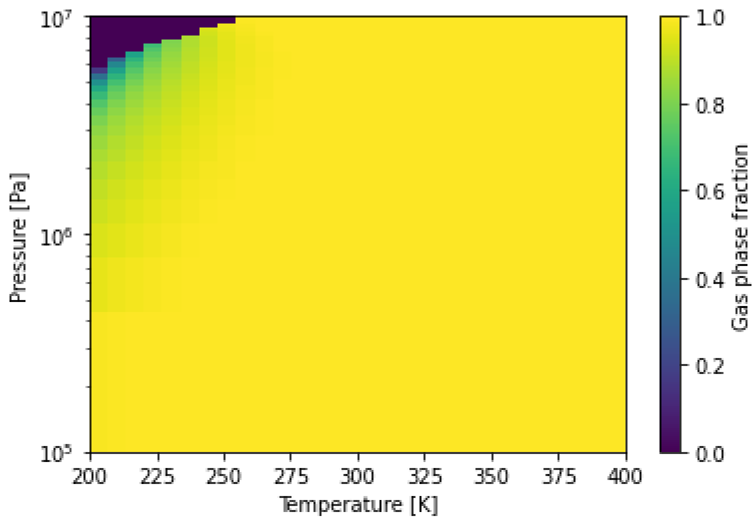
[8]:
```
# For small multiprocessing jobs, the slowest job can cause a significant delay
# For longer and larger jobs the full benefit of using all cores is shown better.
%timeit -r 1 -n 1 processed_data = Parallel(n_jobs=8, batch_size=30)(delayed(get_
↪properties)(T, P) for T, P in zip(Ts_grid.flat, Ps_grid.flat))
```

```
4.42 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[9]:
```
# Joblib returns the data as a flat structure, but we can re-construct it into a grid
processed_data = Parallel(n_jobs=16, batch_size=30)(delayed(get_properties)(T, P) for T,␣
↪P in zip(Ts_grid.flat, Ps_grid.flat))
phase_fractions = np.array([[processed_data[j*pts+i][2] for j in range(pts)] for i in␣
↪range(pts)])
```

[10]:
```
# Make a plot to show the results

import matplotlib.pyplot as plt
from matplotlib import ticker, cm
from matplotlib.colors import LogNorm
fig, ax = plt.subplots()
color_map = cm.viridis
im = ax.pcolormesh(Ts_grid, Ps_grid, phase_fractions.T, cmap=color_map)
cbar = fig.colorbar(im, ax=ax)
cbar.set_label('Gas phase fraction')

ax.set_yscale('log')
ax.set_xlabel('Temperature [K]')
ax.set_ylabel('Pressure [Pa]')
plt.show()
```

```
<ipython-input-10-719d0a113f9b>:8: MatplotlibDeprecationWarning: shading='flat' when X␣
↪and Y have the same dimensions as C is deprecated since 3.3.  Either specify the␣
↪corners of the quadrilaterals with X and Y, or pass shading='auto', 'nearest' or
↪'gouraud', or set rcParams['pcolor.shading'].  This will become an error two minor␣
↪releases later.
  im = ax.pcolormesh(Ts_grid, Ps_grid, phase_fractions.T, cmap=color_map)
```



8.5 Example 14.2 Joule-Thomson Effect

A stream of nitrogen is expanded from T1 = 300 K, P1 = 200 bar, to 1 bar by a throttling valve. An ideal throttling valve has the conditions of being adiabatic (no heat loss, energy is conserved); and is either solved using a valve Cv to solve for pressure or solved with the outlet pressure directly specified.

Calculate the outlet temperature using:

(1) A high precision (helmholtz fundamental) equation of state

(2) The Peng-Robinson equation of state

[1]:
```
# Set the conditions and imports
from scipy.constants import bar
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CoolPropLiquid, CEOSGas,␣
↪CoolPropGas, FlashPureVLS
fluid = 'nitrogen'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 300.0
P1 = 200*bar
P2 = 1*bar
zs = [1]
```

[2]:
```
# Thermo can use CoolProp to provide properties of one or all phases
# For pure species this is quite reliable within the temperature,
# pressure, etc. limits of the EOSs implemented by CoolProp

backend = 'HEOS'
gas = CoolPropGas(backend, fluid, T=T1, P=P1, zs=zs)
liquid = CoolPropLiquid(backend, fluid, T=T1, P=P1, zs=zs)

flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])
```

(continues on next page)

(continued from previous page)

```
state_1 = flasher.flash(T=T1, P=P1)
state_2 = flasher.flash(H=state_1.H(), P=P2)
T2_precise = state_2.T
T2_precise
```

[2]:
```
269.1866854380218
```

[3]:
```
# Use the default originally published Peng-Robinson models
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

state_1 = flasher.flash(T=T1, P=P1)
state_2 = flasher.flash(H=state_1.H(), P=P2)
T2_PR = state_2.T
T2_PR
```

[3]:
```
265.50610736019723
```

The outlet temperature anwsers given in the book are 269.19 K for the high-precision EOS, and for the PR EOS they used a very low precision Cp of 1 J/(g*K) and obtained an outlet temperature of 283.05 K.

The book textbook cites this 14 K difference as coming from the cubic EOS's lack of precision but the above calculation shows that if an accurate heat capacity is used the difference is only ~ 4K.

8.6 Example 14.3 Adiabatic Compression and Expansion

A heat pump using the refrigerant R-22 operates with a mass flow rate of 100 kg/hr. The fluid enters the compressor at T1 = 300 K and P1 = 1 bar. The compressor heat loss is neglected. The outlet pressure of the compressor is 5 bar. If the isentropic efficiency of the compressor is 0.7 and the mechanical efficiency is 0.9, what is the power draw of the compressor and how how is the refrigerant when it exits the compressor?

The textbook uses the Peng-Robinson EOS, so to compare, use that as well.

[1]:
```
# Set the conditions and imports
from scipy.constants import bar, hour
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CEOSGas, FlashPureVLS
fluid = 'R-22'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 300.0
P1 = 1*bar
P2 = 5*bar
eta_isentropic = 0.7
eta_mechanical = 0.9
```

[2]:
```
# Use the default originally published Peng-Robinson models
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
```

(continues on next page)

(continued from previous page)

```
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

# Flash at inlet conditions to obtain initial enthalpy
state_1 = flasher.flash(T=T1, P=P1)
# Flash at outlet condition - entropy is conserved by compressors and expanders!
state_2_ideal = flasher.flash(S=state_1.S(), P=P2)
# Compute the change in enthalpy
delta_H_ideal = (state_2_ideal.H()-state_1.H())
# The definition of isentropic efficiency means that the actual amount of heat added is
# dH_actual = dH_idea/eta_isentropic
H_added_to_fluid_actual = delta_H_ideal/eta_isentropic

state_2 = flasher.flash(H=state_1.H() + H_added_to_fluid_actual, P=P2)

# To compute the actual power, itis more convinient to use the mass enthalpy
actual_power_per_kg = (state_2.H_mass() - state_1.H_mass())/(eta_mechanical) # W/kg
actual_power = actual_power_per_kg*100/hour
print(f'The actual power is {actual_power:.0f} W')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
The actual power is 2252 W
The actual outlet temperature is  406.60 K
```

The power given in the textbook is 2257 W and 405.68 K out. No details as to the liquid heat capacity are given. As refrigerants are well defined substances, it is recommended for anyone doing modeling with them to use a high-accuracy model wherever possible.

8.7 Problem 14.02 Work and Temperature Change Upon Isentropic Compression of Oxygen

A stream of gaseous oxygen is compressed from 1 bar to 10 bar. The inlet temperature is 25 °C. Calculate the specific work and the temperature of the outlet gas if the process as an isentropic efficiency of 1, using both the ideal gas law and the SRK equation of state.

8.7.1 Solution

This requires a PT and then a PS flash only. This problem is also good for contrasting simple engineering formulas for compression vs. rigorous thermodynamics.

[1]:
```
# Set the conditions and imports
from scipy.constants import bar, hour
from thermo import ChemicalConstantsPackage, SRKMIX, IdealGas, CEOSLiquid, CEOSGas,␣
↪FlashPureVLS
fluid = 'oxygen'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 298.15
```

(continues on next page)

(continued from previous page)

```
P1 = 1*bar
P2 = 10*bar
```

[2]:
```
# Use the Ideal-Gas EOS
gas = IdealGas(HeatCapacityGases=correlations.HeatCapacityGases)
# Note that we can set-up a flasher object with only a gas phase
# This obviously has much more performance!
flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[], solids=[])

# Flash at inlet conditions to obtain initial enthalpy
state_1 = state_1_ideal = flasher.flash(T=T1, P=P1)
# Flash at outlet condition - entropy is conserved by compressors and expanders!
state_2 = state_2_ideal = flasher.flash(S=state_1.S(), P=P2)

actual_power = (state_2.H() - state_1.H()) # W/mol
print('With the ideal-gas EOS:')
print(f'The actual power is {actual_power:.4f} J/mol')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
With the ideal-gas EOS:
The actual power is 7991.2798 J/mol
The actual outlet temperature is  560.70 K
```

[3]:
```
# SRK
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(SRKMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
gas = CEOSGas(SRKMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

# Flash at inlet conditions to obtain initial enthalpy
state_1 = flasher.flash(T=T1, P=P1)
# Flash at outlet condition - entropy is conserved by compressors and expanders!
state_2 = state_2_ideal = flasher.flash(S=state_1.S(), P=P2)

actual_power = (state_2.H() - state_1.H()) # W/mol
print('With the SRK EOS:')
print(f'The actual power is {actual_power:.4f} J/mol')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
With the SRK EOS:
The actual power is 8000.1749 J/mol
The actual outlet temperature is  561.06 K
```

These calculations make use of the full power of the Thermo engine. It is also possible to use simpler calculations to calculate, as shown below.

[4]:
```
from fluids import isentropic_work_compression, isentropic_T_rise_compression
k = state_1_ideal.isentropic_exponent()
Z = state_1_ideal.Z()
print(f'Using the ideal isentropic exponent {k:.3f}')
print(f'Using the ideal compressibility {Z:.3f}')
```

(continues on next page)

(continued from previous page)

```
molar_work = isentropic_work_compression(T1=T1, k=state_1_ideal.isentropic_exponent(),␣
↪Z=state_1_ideal.Z(), P1=P1, P2=P2, eta=1)
T2 = isentropic_T_rise_compression(T1=T1, P1=P1, P2=P2, k=k, eta=1)
print(f'The simple power is {molar_work:.4f} J/mol')
print(f'The simple outlet temperature is {T2: .2f} K')
```

```
Using the ideal isentropic exponent 1.395
Using the ideal compressibility 1.000
The simple power is 8047.9387 J/mol
The simple outlet temperature is  572.15 K
```

From these results, we can see that for small pressure increases, the ideal-gas and SRK equations work quite similarly. There is also a very large difference in outlet temperature between the simplified equations given in many textbooks, and the real isentropic calculations when a temperature-dependent heat capacity is used. Therefore, there are substantial advantages to rigorous modeling, regardless of the complexity of the EOS for the gas phase.

8.8 Problem 14.03 Reversible and Isothermal Compression of Liquid Water

A flow of 2000 kg/h liquid water at 25 °C and 1 bar is pumped to a pressure of 100 bar. The pump is “cooled”, so the process is reversible and isothermal. What is the duty of the pump shaft, and the energy that must be removed from the water being compressed?

8.8.1 Solution

We can use the high-accuracy IAPWS-95 implementation of the properties of water to easily and extremely accurately calculate these values.

[87]:
```
from scipy.constants import bar, hour
import numpy as np
from thermo import FlashPureVLS, IAPWS95Liquid, IAPWS95Gas, iapws_constants, iapws_
↪correlations
from scipy.integrate import quad
import numpy as np

T1 = T2 = 25 + 273.15
P1 = 1*bar
P2 = 100*bar


liquid = IAPWS95Liquid(T=T1, P=P1, zs=[1])
gas = IAPWS95Gas(T=T1, P=P1, zs=[1])
flasher = FlashPureVLS(iapws_constants, iapws_correlations, gas, [liquid], [])

mass_flow = 2000/hour
mole_flow = property_molar_to_mass(mass_flow, MW=iapws_constants.MWs[0])

entry = flasher.flash(T=T1, P=P1)
leaving = flasher.flash(T=T2, P=P2)
```

(continues on next page)

(continued from previous page)

```
def to_int(P, flasher):
    state = flasher.flash(T=T1, P=P)
    return state.V()
integral_result = quad(to_int, P1, P2, args=(flasher,))[0]
shaft_duty = integral_result*mole_flow

cooling_duty = shaft_duty - (leaving.H() - entry.H())*mole_flow

print(f'The shaft power is {shaft_duty:.8f} W')
print(f'The cooling duty is {cooling_duty:.4f} W')
```

```
The shaft power is 5504.05633851 W
The cooling duty is 431.1770 W
```

The above shows the numerical integral calculation. That is the correct formulation.

However, it can be a little unintuitive. We can contrast this with another calculation - a series of tiny isentropic compression, then cooling steps.

[86]:
```
cooling_duty = 0
compressing_duty = 0
increments = 30 # Number of increments
dP = (P2 - P1)/increments

old_state = entry
for i in range(increments):
    P = P1+(i+1)*dP

    # Compress another increment of pressure
    new_compressed_state = flasher.flash(S=old_state.S(), P=P)
    compressing_duty += (new_compressed_state.H() - old_state.H())*mole_flow

    # Cool back to T1 at new pressure
    new_cooled_state = flasher.flash(T=T1, P=P)
    cooling_duty += (new_compressed_state.H() - new_cooled_state.H())*mole_flow

    old_state = new_cooled_state

print(f'The shaft power is {compressing_duty:.4f} W')
print(f'The cooling duty is {cooling_duty:.4f} W')
```

```
The shaft power is 5504.0608 W
The cooling duty is 431.1815 W
```

8.9 Problem 14.04 Heat Effect Upon Mixing of Methane and Dodecane at Elevated Temperature and Pressure Using SRK

1600 kg/hr of methane is mixed with 170 kg/hr of dodecane. The inlet temperature of both streams is 160 °C, and each enter at a pressure of 2 MPa. The mixing process is isobaric. What is the temperature of the combined stream? Use the SRK EOS with no binary interaction parameters.

8.9.1 Solution

This is a straightforward calculation. The energy of both streams is combined; and the outlet pressure is known. The calculation only requires calculating the inlet energy of both streams, adding it up, and finding the mole fractions of the outlet.

[12]:
```
from thermo import ChemicalConstantsPackage, SRKMIX, FlashVL, CEOSLiquid, CEOSGas
from chemicals import ws_to_zs, mixing_simple

constants, correlations = ChemicalConstantsPackage.from_IDs(['methane', 'dodecane'])
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(SRKMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
gas = CEOSGas(SRKMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher = FlashVL(constants, correlations, liquid=liquid, gas=gas)

P1 = P2 = 2e6
T1 = 160+273.15

ws = [1600, 170]
zs = ws_to_zs(ws=ws, MWs=constants.MWs)

methane_H = flasher.flash(T=T1, P=P1, zs=[1, 0]).H()
dodecane_H = flasher.flash(T=T1, P=P1, zs=[0, 1]).H()
H = zs[0]*methane_H + zs[1]*dodecane_H

res = flasher.flash(P=P2, H=H, zs=zs)
print(f'The outlet temperature is {res.T-273.15:.4f} °C')
```

```
The outlet temperature is 150.2259 °C
```

8.10 Problem 14.05 Required Power for R134a Compression Using a High Precision Equation of State

Refrigerant R134a is compressed from a saturated vapor at 5 °C to an outlet pressure of 1 MPa. Calculate the power of the compressor, using a high-precision EOS.

The mechanical efficiency is 0.95, and the isentropic efficiency 0.7; the mass flow rate is 3000 kg/hr.

8.10.1 Solution

This is straightforward.

[8]:
```
# Set the conditions and imports
from scipy.constants import bar, hour
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CoolPropLiquid, CEOSGas,␣
↪CoolPropGas, FlashPureVLS
fluid = 'R134a'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 5 + 273.15
VF1 = 1
P2 = 10*bar
zs = [1]
eta_isentropic = 0.7
eta_mechanical = 0.9
```

[10]:
```
backend = 'HEOS'
gas = CoolPropGas(backend, fluid, T=T1, P=1e5, zs=zs)
liquid = CoolPropLiquid(backend, fluid, T=T1, P=1e5, zs=zs)

flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

# Flash at inlet conditions to obtain initial enthalpy
state_1 = flasher.flash(T=T1, VF=VF1)
# Flash at outlet condition - entropy is conserved by compressors and expanders!
state_2_ideal = flasher.flash(S=state_1.S(), P=P2)
# Compute the change in enthalpy
delta_H_ideal = (state_2_ideal.H()-state_1.H())
# The definition of isentropic efficiency means that the actual amount of heat added is
# dH_actual = dH_idea/eta_isentropic
H_added_to_fluid_actual = delta_H_ideal/eta_isentropic

state_2 = flasher.flash(H=state_1.H() + H_added_to_fluid_actual, P=P2)

# To compute the actual power, itis more convinient to use the mass enthalpy
actual_power_per_kg = (state_2.H_mass() - state_1.H_mass())/(eta_mechanical) # W/kg
actual_power = actual_power_per_kg*3000/hour
print(f'The actual power is {actual_power:.0f} W')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
The actual power is 28858 W
The actual outlet temperature is  324.80 K
```

8.11 Problem 14.06 Required Volume for a Gas Storage Tank for Ammonia

50 m^3 of liquid ammonia is stored at the conditions 50 °C and 100 bar. The vessel fails, and the contents empties into a backup containment vessel. The backup vessel has a maximum pressure of 10 bar. What volume must be vessel be to not exceed this pressure?

8.11.1 Solution

This is straightforward; energy is conserved and a pressure is specified. Find the amount of ammonia in the original vessel; find the molar volume of ammonia in the new vessel; and multiply that by the amount of ammonia.

Ammonia is a highly non-ideal fluid, so we use a high-precision EOS.

[10]:
```
# Set the conditions and imports
from scipy.constants import bar
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CoolPropLiquid, CEOSGas,␣
↪CoolPropGas, FlashPureVLS
fluid = 'ammonia'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 50 + 273.15
P1 = 100*bar
P2 = 10*bar
zs = [1]
volume_1 = 50

backend = 'HEOS'
gas = CoolPropGas(backend, fluid, T=T1, P=1e5, zs=zs)
liquid = CoolPropLiquid(backend, fluid, T=T1, P=1e5, zs=zs)

flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])
```

[12]:
```
# Flash at inlet conditions to obtain initial enthalpy
state_1 = flasher.flash(T=T1, P=P1)
moles = volume_1/state_1.V()
state_2 = flasher.flash(P=P2, H=state_1.H())

volume_2 = moles*state_2.V()
print(f'The thermodynamically required secondary containment volume is {volume_2: .2f} m^
↪3')
```

```
The thermodynamically required secondary containment volume is  433.83 m^3
```

8.12 Problem 14.07 Liquid Nitrogen Production Via Volume Expansion of the Compressed Gas

Nitrogen at -104 °C and 250 bar flows through a valve to a pressure of 1 bar. What fraction of the stream becomes liquid?

8.12.1 Solution

This is straightforward; energy is conserved and outlet presure is specified, making this a PH flash. This problem is also an important application that can show the results of different equations of state and how important good thermodynamics are.

We can compare many different EOSs with Thermo easily.

[1]:
```
from thermo import *
from thermo.interaction_parameters import SPDB
fluid = 'nitrogen'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = -104 + 273.15
P1 = 240*1e5
zs = [1]
P2 = 1e5
```

[2]:
```
flasher_objects = []
flasher_names = []

gas = CoolPropGas('HEOS', fluid, T=T1, P=P1, zs=zs)
liquid = CoolPropLiquid('HEOS', fluid, T=T1, P=P1, zs=zs)
high_precision = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid],␣
↪solids=[])
flasher_objects.append(high_precision)
flasher_names.append('High-Precision')

# Add the Peng-Robinson Pina-Martinez parameters EOS
Ls = SPDB.get_parameter_vector(name='PRTwu_PinaMartinez', CASs=constants.CASs, parameter=
↪'TwuPRL')
Ms = SPDB.get_parameter_vector(name='PRTwu_PinaMartinez', CASs=constants.CASs, parameter=
↪'TwuPRM')
Ns = SPDB.get_parameter_vector(name='PRTwu_PinaMartinez', CASs=constants.CASs, parameter=
↪'TwuPRN')
cs = SPDB.get_parameter_vector(name='PRTwu_PinaMartinez', CASs=constants.CASs, parameter=
↪'TwuPRc')
alpha_coeffs = [(Ls[i], Ms[i], Ns[i]) for i in range(constants.N)]
eos_kwargs = {'Pcs': constants.Pcs, 'Tcs': constants.Tcs, 'omegas': constants.omegas,
'cs': cs, 'alpha_coeffs':alpha_coeffs}

gas = CEOSGas(PRMIXTranslatedConsistent, eos_kwargs=eos_kwargs,␣
↪HeatCapacityGases=correlations.HeatCapacityGases)
liquid = CEOSLiquid(PRMIXTranslatedConsistent, eos_kwargs=eos_kwargs,␣
↪HeatCapacityGases=correlations.HeatCapacityGases)
eos_obj = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])
```
(continues on next page)

(continued from previous page)

```
flasher_objects.append(eos_obj)
flasher_names.append('PR-Pina-Martinez')

# Add the SRK Pina-Martinez parameters EOS
Ls = SPDB.get_parameter_vector(name='SRKTwu_PinaMartinez', CASs=constants.CASs,␣
↪parameter='TwuSRKL')
Ms = SPDB.get_parameter_vector(name='SRKTwu_PinaMartinez', CASs=constants.CASs,␣
↪parameter='TwuSRKM')
Ns = SPDB.get_parameter_vector(name='SRKTwu_PinaMartinez', CASs=constants.CASs,␣
↪parameter='TwuSRKN')
cs = SPDB.get_parameter_vector(name='SRKTwu_PinaMartinez', CASs=constants.CASs,␣
↪parameter='TwuSRKc')
alpha_coeffs = [(Ls[i], Ms[i], Ns[i]) for i in range(constants.N)]
eos_kwargs = {'Pcs': constants.Pcs, 'Tcs': constants.Tcs, 'omegas': constants.omegas,
'cs': cs, 'alpha_coeffs':alpha_coeffs}

gas = CEOSGas(SRKMIXTranslatedConsistent, eos_kwargs=eos_kwargs,␣
↪HeatCapacityGases=correlations.HeatCapacityGases)
liquid = CEOSLiquid(SRKMIXTranslatedConsistent, eos_kwargs=eos_kwargs,␣
↪HeatCapacityGases=correlations.HeatCapacityGases)
eos_obj = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])
flasher_objects.append(eos_obj)
flasher_names.append('SRK-Pina-Martinez')

# Add a bunch of EOSs that don't require any parameters
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)

cubic_EOSs = [('PR', PRMIX), ('SRK', SRKMIX),
              ('VDW', VDWMIX),
             ('PRSV', PRSVMIX), ('PRSV2', PRSV2MIX),
             ('TWUPR', TWUPRMIX), ('TWUSRK', TWUSRKMIX),
             ('PRTranslatedConsistent', PRMIXTranslatedConsistent),
             ('SRKTranslatedConsistent', SRKMIXTranslatedConsistent)]
for eos_name, eos_obj in cubic_EOSs:
    liquid = CEOSLiquid(eos_obj, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
    gas = CEOSGas(eos_obj, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
    eos_obj = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

    flasher_objects.append(eos_obj)
    flasher_names.append(eos_name)
```

[3]:
```
for obj, obj_name in zip(flasher_objects, flasher_names):
    state_1 = obj.flash(T=T1, P=P1, zs=zs)
    state_2 = obj.flash(P=P2, H=state_1.H(), zs=zs)
    print(f'The {obj_name} EOS predicted liquid molar fraction is {state_2.LF:.8f}.')
```

```
The High-Precision EOS predicted liquid molar fraction is 0.03887228.
The PR-Pina-Martinez EOS predicted liquid molar fraction is 0.05536129.
```

(continues on next page)

(continued from previous page)

```
The SRK-Pina-Martinez EOS predicted liquid molar fraction is 0.06765522.
The PR EOS predicted liquid molar fraction is 0.05963486.
The SRK EOS predicted liquid molar fraction is 0.04341557.
The VDW EOS predicted liquid molar fraction is 0.00000000.
The PRSV EOS predicted liquid molar fraction is 0.06011654.
The PRSV2 EOS predicted liquid molar fraction is 0.06011654.
The TWUPR EOS predicted liquid molar fraction is 0.05491152.
The TWUSRK EOS predicted liquid molar fraction is 0.04670591.
The PRTranslatedConsistent EOS predicted liquid molar fraction is 0.05860220.
The SRKTranslatedConsistent EOS predicted liquid molar fraction is 0.07069564.
```

As can be see, the equation of state used changes the results drastically. Even the best of the cubic equations of state given results 30-50% off from the high-precision equation of state. This problem was admittedly constructed to show off the importance of using higher precision models, but the point applies elsewhere also.

8.13 Problem 14.08 Required Compressor Power for Isothermal and Adiabatic Compression of a Gas Mixture (CO2, O2) Using the Ideal Gas Law

A stream of 1000 mol/hour CO2 and 1000 mol/hour O2 is compressed from 290 K and 1 bar to 5 bar. Calculate the compression power for both adiabatic compression, and isothermal compression. The compression is reversible (assumed) in each case - no efficiencies are necessary.

8.13.1 Solution

This is a straightforward calculation. Using Thermo, working with complicated mixtures can be about as easy as pure components - if binary interaction parameters are zero. In this case, we try to load a parameter from a sample ChemSep database, but no values are available.

The values in that database are just a sample - it is entirely the user's responsibility to provide the correct data to Thermo. If garbage is put in, garbage will come out!

The problem says to use the ideal-gas law, so we can do that too and see how the answers compare.

[1]:
```
from scipy.constants import hour
T1 = 290
P1 = 1e5
P2 = 5e5
flow = 2000/hour # mol/s

from thermo import ChemicalConstantsPackage, PRMIX, IGMIX, FlashVL, CEOSLiquid, CEOSGas
from thermo.interaction_parameters import IPDB

constants, correlations = ChemicalConstantsPackage.from_IDs(['CO2', 'O2'])
kijs = IPDB.get_ip_asymmetric_matrix('ChemSep PR', constants.CASs, 'kij')
print(f'The PR kij matrix is {kijs}')

eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas,
                  kijs=kijs)
liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
```

(continues on next page)

(continued from previous page)

```
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher = FlashVL(constants, correlations, liquid=liquid, gas=gas)
zs = [.5, .5]

liquid = CEOSLiquid(IGMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
gas = CEOSGas(IGMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
flasher_ideal = FlashVL(constants, correlations, liquid=liquid, gas=gas)
```

```
The PR kij matrix is [[0.0, 0], [0, 0.0]]
```

Adiabatic compression

[2]:
```
# Solve with Peng-Robinson
state_1 = flasher.flash(T=T1, P=P1, zs=zs)
state_2 = flasher.flash(S=state_1.S(), P=P2, zs=zs)
shaft_duty = (state_2.H() - state_1.H())*flow

print(f'The shaft power with Peng-Robinson is {shaft_duty:.4f} W')


state_1 = flasher_ideal.flash(T=T1, P=P1, zs=zs)
state_2 = flasher_ideal.flash(S=state_1.S(), P=P2, zs=zs)
shaft_duty = (state_2.H() - state_1.H())*flow
print(f'The shaft power with ideal-gas is {shaft_duty:.4f} W')
```

```
The shaft power with Peng-Robinson is 2632.7895 W
The shaft power with ideal-gas is 2639.9248 W
```

Isothermal Compression

This problem is more interesting, because there is the cooling duty as well as the compressing duty.

From theory, in an ideal gas, the cooling duty will be exactly equal to the compressing duty.

For a real-gas, it will be different as enthalpy is pressure-dependent.

In both cases, the evaluation of the following integral is required.

$$\text{duty} = \text{flow} \int_{P1}^{P2} V \partial P$$

[3]:
```
from scipy.integrate import quad

def to_int(P, flasher):
    state = flasher.flash(T=T1, P=P, zs=zs)
    return state.V()
shaft_duty = cooling_duty = quad(to_int, P1, P2, args=(flasher_ideal,))[0]*flow
```

(continues on next page)

(continued from previous page)

```
print(f'The shaft power with ideal-gas is {shaft_duty:.4f} W')
print(f'The cooling duty with ideal-gas is {cooling_duty:.4f} W')

entry = flasher.flash(T=T1, P=P1, zs=zs)
exit = flasher.flash(T=T1, P=P2, zs=zs)

shaft_duty = quad(to_int, P1, P2, args=(flasher,))[0]*flow
cooling_duty = shaft_duty - (exit.H() - entry.H())*flow

print(f'The shaft power with Peng-Robinson is {shaft_duty:.8f} W')
print(f'The cooling duty with Peng-Robinson is {cooling_duty:.8f} W')
```

```
The shaft power with ideal-gas is 2155.9263 W
The cooling duty with ideal-gas is 2155.9263 W
The shaft power with Peng-Robinson is 2139.44610002 W
The cooling duty with Peng-Robinson is 2192.57596810 W
```

The above shows the numerical integral calculation. That is the correct formulation.

However, it can be a little unintuitive. We can contrast this with another calculation - a series of tiny isentropic compression, then cooling steps.

[4]:

```
cooling_duty = 0
compressing_duty = 0
increments = 3 # Number of increments
dP = (P2 - P1)/increments
old_state = entry
for i in range(increments):
    P = P1+(i+1)*dP

    # Compress another increment of pressure
    new_compressed_state = flasher.flash(S=old_state.S(), P=P, zs=zs)
    compressing_duty += (new_compressed_state.H() - old_state.H())*flow

    # Cool back to T1 at new pressure
    new_cooled_state = flasher.flash(T=T1, P=P, zs=zs)
    cooling_duty += (new_compressed_state.H() - new_cooled_state.H())*flow

    old_state = new_cooled_state

print(f'The shaft power is {compressing_duty:.8f} W')
print(f'The cooling duty is {cooling_duty:.8f} W')
```

```
The shaft power is 2322.61227046 W
The cooling duty is 2375.74213854 W
```

8.14 Problem 14.09 Temperature Change Upon Ethylene Expansion in Throttle Valves Using a High Precision EOS

Ethylene is expanded from P1 = 3000 bar, T1 = 600 K to P2 = 300 bar by a first valve, and then to P3 = 1 bar by a second valve. What are the temperatures T2 and T3? Neglect the velocity term in the solution.

8.14.1 Solution

This is straightforward - an initial PT flash calculation, followed by two separate PH flash calculations.

[1]:
```
# Set the conditions and imports
from scipy.constants import bar, hour
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CoolPropLiquid, CEOSGas,␣
↪CoolPropGas, FlashPureVLS
fluid = 'ethylene'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 600
P1 = 3000*bar
P2 = 300*bar
P3 = 1*bar
zs = [1]
```

[2]:
```
backend = 'HEOS'
gas = CoolPropGas(backend, fluid, T=T1, P=P1, zs=zs)
liquid = CoolPropLiquid(backend, fluid, T=T1, P=P1, zs=zs)

flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

# Flash at inlet conditions to obtain initial enthalpy
state_1 = flasher.flash(T=T1, P=P1)
state_2 = flasher.flash(H=state_1.H(), P=P2)
state_3 = flasher.flash(H=state_1.H(), P=P3)

print(f'The second temperature is {state_2.T: .2f} K')
print(f'The third temperature is {state_3.T: .2f} K')
```

```
The second temperature is  676.94 K
The third temperature is  651.47 K
```

8.15 Problem 14.10 Leakage Rate Change in Vacuum Distillation When Lowering the Column Pressure

In sub-atmospheric pressure distillation columns, a vacuum system removes entering air by removing a vapor stream, usually near the top of the column. If air is not removed the pressure will continue to increase, as the air itself won't condense through the condenser (unless it is cryogenic). Air can also pose a fire hazard in some cases.

How will the leakage rate into the column change if the pressure of the column is lowered from 0.4 bar to 0.1 bar? Assume the ambient pressure is 1.013 bar.

8.15.1 Solution

Leaks into a column are usually around flanges, through valve or pump packings, inspection or sampling ports, or manholes.

There are a variety of empirical correlations that can be used to estimate leakage depending on pressure. The first answer uses one of those. These are not truly `mechanistic`, however.

We can also imagine a single hole, and treat the flow as through an orifice. This is the second answer.

We can also treat the hole as an isothermal compressible gas flow problem. The third answer uses that.

[18]:
```
from math import pi
from scipy.constants import hour
from fluids import *
V = 10
P1 = 0.4*1e5
P2 = 0.1*1e5
P_ambient = 101325

rho = 1.2

D = .8
H = 15
V = pi/4*D**2*H

m1 = vacuum_air_leakage_Seider(V=V, P=P1)*hour
m2 = vacuum_air_leakage_Seider(V=V, P=P2)*hour
m_ratio = m2/m1
print(f'Using an emperical correlation, the ratio of air increase is {m_ratio: .3f}.')
```

```
Using an emperical correlation, the ratio of air increase is  1.029.
```

[21]:
```
# Imagine a 0.1 m hole in the tower
D_hole = 1e-7
beta = D_hole/D

m1 = differential_pressure_meter_solver(D=D_hole/beta, D2=D_hole, P1=P_ambient, P2=P1,
                                        rho=rho, mu=1e-3, k=1.3, meter_type='ISO 5167␣
↪orifice', taps='D')
m2 = differential_pressure_meter_solver(D=D_hole/beta, D2=D_hole, P1=P_ambient, P2=P2,
                                        rho=rho, mu=1e-3, k=1.3, meter_type='ISO 5167␣
↪orifice', taps='D')
m_ratio = m2/m1
print(f'Using a flow meter correlation, the ratio of air increase is {m_ratio: .3f}.')
```

```
Using a flow meter correlation, the ratio of air increase is  1.031.
```

[22]:
```
t_hole = 0.008 # 0.8 mm thick wall
m1 = isothermal_gas(rho=rho, fd=0.01, P1=P_ambient, P2=P1, L=t_hole, D=D_hole)
m2 = isothermal_gas(rho=rho, fd=0.01, P1=P_ambient, P2=P2, L=t_hole, D=D_hole)
m_ratio = m2/m1
print(f'Using isothermal compressible gas flow, the ratio of air increase is {m_ratio: .
↪3f}.')
```

```
Using isothermal compressible gas flow, the ratio of air increase is  1.081.
```

8.16 Problem 14.11 Pressure Rise In a Storage Tank Upon Heating

500 kg of propylene is contained in a 1 m^3 vessel stored at 30 °C. The vessel is heated - from solar radiation in the problem statement. What is the initial pressure?

The safety valve of the tank activates at 60 bar. If the cooling system is disabled, what temperature will the contents of the vessel be when the valve actuates?

8.16.1 Solution

This is straightforward - an initial solution with total volume, mass, and temperature specified, followed by solving for the end temperature to obtain a specified pressure.

From experience the vessel is known to be liquid. Because of that, we can skip the flash calculations and work directly with the liquid phase object. That is normally much faster than the flash calculations.

[1]:
```
from scipy.constants import bar, hour
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CoolPropLiquid, CEOSGas,␣
↪CoolPropGas, FlashPureVLS
fluid = 'propylene'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 30 + 273.15
P2 = 60*bar
zs = [1]
V_total = 1 # m^3
m = 500 # kg

backend = 'HEOS'
gas = CoolPropGas(backend, fluid, T=T1, P=1e5, zs=zs)
liquid = CoolPropLiquid(backend, fluid, T=T1, P=1e5, zs=zs)

flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[liquid], solids=[])

# Calculate the total number of moles
moles = m/(1e-3*constants.MWs[0])
# Calculate the molar volume
Vm_initial = V_total/moles

# We know the phase is liquid, so we can skip the flash and solve for the liquid at this␣
↪state
state_1 = liquid.to(T=T1, V=Vm_initial, zs=zs)
print(f'The initial pressure is {state_1.P/1e6: .3f} MPa')

state_2 = liquid.to(P=P2, V=Vm_initial, zs=zs)
print(f'The end tempererature is {state_2.T: .3f} K')
```

```
The initial pressure is  1.979 MPa
The end tempererature is  311.102 K
```

8.17 Problem 14.12 Work and Temperature Change Upon Adiabatic Compression of Oxygen

A stream of oxygen is compressed by a compressor from a pressure P1 = 1 bar to P2 = 10 bar. The flow rate of the oxygen stream is 250 kg/h and the temperature is 25°C.

What is the power of the compressor, and the outlet temperature of the gas?

8.17.1 Solution

This is a series of PH, PS and PT flashes.

[1]:
```
from scipy.constants import bar, hour
from thermo import ChemicalConstantsPackage, SRKMIX, IGMIX, CEOSGas, CEOSLiquid,␣
↪FlashPureVLS
fluid = 'oxygen'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 25 + 273.15
P1 = 1*bar
P2 = 10*bar
zs = [1]
eta_isentropic = 0.75
eta_mechanical = 0.95
```

[2]:
```
eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(SRKMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_
↪kwargs=eos_kwargs)
gas = CEOSGas(SRKMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
SRK_flasher = FlashPureVLS(constants, correlations, liquids=[liquid], gas=gas, solids=[])

gas = CEOSGas(IGMIX, HeatCapacityGases=correlations.HeatCapacityGases, eos_kwargs=eos_
↪kwargs)
ideal_flasher = FlashPureVLS(constants, correlations, gas=gas, liquids=[], solids=[])
```

[3]:
```
# Flash at inlet conditions to obtain initial enthalpy
state_1 = SRK_flasher.flash(T=T1, P=P1)
# Flash at outlet condition - entropy is conserved by compressors and expanders!
state_2_ideal = SRK_flasher.flash(S=state_1.S(), P=P2)
# Compute the change in enthalpy
delta_H_ideal = (state_2_ideal.H()-state_1.H())
# The definition of isentropic efficiency means that the actual amount of heat added is
# dH_actual = dH_idea/eta_isentropic
H_added_to_fluid_actual = delta_H_ideal/eta_isentropic

state_2 = SRK_flasher.flash(H=state_1.H() + H_added_to_fluid_actual, P=P2)

# To compute the actual power, itis more convinient to use the mass enthalpy
actual_power_per_kg = (state_2.H_mass() - state_1.H_mass())/(eta_mechanical) # W/kg
```

(continues on next page)

(continued from previous page)

```
actual_power = actual_power_per_kg*250/hour
print('With the SRK EOS:')
print(f'The actual power is {actual_power:.0f} W')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
With the SRK EOS:
The actual power is 24368 W
The actual outlet temperature is  643.85 K
```

[4]:
```
# Flash at inlet conditions to obtain initial enthalpy
state_1 = ideal_flasher.flash(T=T1, P=P1)
# Flash at outlet condition - entropy is conserved by compressors and expanders!
state_2_ideal = ideal_flasher.flash(S=state_1.S(), P=P2)
# Compute the change in enthalpy
delta_H_ideal = (state_2_ideal.H()-state_1.H())
# The definition of isentropic efficiency means that the actual amount of heat added is
# dH_actual = dH_idea/eta_isentropic
H_added_to_fluid_actual = delta_H_ideal/eta_isentropic

state_2 = ideal_flasher.flash(H=state_1.H() + H_added_to_fluid_actual, P=P2)

# To compute the actual power, itis more convinient to use the mass enthalpy
actual_power_per_kg = (state_2.H_mass() - state_1.H_mass())/(eta_mechanical) # W/kg
actual_power = actual_power_per_kg*250/hour
print('With the ideal EOS:')
print(f'The actual power is {actual_power:.0f} W')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
With the ideal EOS:
The actual power is 24341 W
The actual outlet temperature is  643.68 K
```

8.18 Problem 14.13 Thermodynamic Cycle Calculation Using a High-Precision EOS

A thermodynamic cycle with water as the working fluid consists of the following steps:

- Constant-pressure heating to P1 = 100 bar and T1 = 350 °C
- Isentropic expansion of the gas in a turbine to P2 = 1 bar (reversible; efficiency = 100%)
- Constant pressure condensation
- Isentropic compression of the liquid to P4 = 100 bar

What is the thermal efficiency of the process?

$$\eta_{th} = -\frac{P_{12} + P_{34}}{Q_{41}}$$

8.18.1 Solution

This is quite straightforward.

[1]:
```
import numpy as np
from thermo import FlashPureVLS, IAPWS95Liquid, IAPWS95Gas, iapws_constants, iapws_
↪correlations
from scipy.integrate import quad
import numpy as np

T1 = 350 + 273.15
P1 = 100*1e5
P2 = 1e5
# Entropy conserved in step 2 as well
VF3 = 0
P3 = P2

P4 = P1
# entropy conserved in step 5 as well


liquid = IAPWS95Liquid(T=T1, P=P1, zs=[1])
gas = IAPWS95Gas(T=T1, P=P1, zs=[1])
flasher = FlashPureVLS(iapws_constants, iapws_correlations, gas, [liquid], [])


stage_1 = flasher.flash(P=P1, T=T1)
stage_2 = flasher.flash(P=P2, S=stage_1.S())
stage_3 = flasher.flash(VF=VF3, P=P3)
stage_4 = flasher.flash(P=P4, S=stage_3.S())

expander_duty = stage_2.H() - stage_1.H()
pump_duty = stage_4.H() - stage_3.H()
heating_duty = stage_1.H() - stage_4.H()
cooling_duty = stage_3.H() - stage_2.H()
heating_duty, cooling_duty, expander_duty, pump_duty
```

[1]:
```
(44969.97634439414,
 -31180.343551697508,
 -13975.281899345828,
 185.64910664919353)
```

[2]:
```
# it is easy to check the cycle converged
cycle_error = sum([heating_duty, cooling_duty, expander_duty, pump_duty])
cycle_error
```

[2]:
```
-9.094947017729282e-13
```

[3]:
```
# Not quite sure what definition is being suggested by the textbook
eta_th = -expander_duty/heating_duty
print(f'The thermal efficiency is {eta_th*100:.2f} %')
```

```
The thermal efficiency is 31.08 %
```

8.19 Problem 14.14 Refrigeration Cycle Calculation Using the Peng-Robinson EOS

A refrigerator uses the refrigerant R-12, dichlorodifluoromethane. The steps and conditions of the cycle are as follows:

- Isobaric condensation to saturation temperature 30°C
- Adiabatic let-down to P2 = 20 degrees subcooling
- Isobaric evaporation to saturation temperature of 20 °C
- Isentropic compression to P4 = 30 °C

Use the Peng-Robinson EOS.

8.19.1 Solution

This is quite straightforward, with the only complication coming from the degrees of subcooling.

[1]:
```python
# Set the conditions and imports
from scipy.constants import bar
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CEOSGas, FlashPureVLS
fluid = 'dichlorodifluoromethane'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

zs = [1]

eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,
                    eos_kwargs=eos_kwargs)
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,
              eos_kwargs=eos_kwargs)
flasher = FlashPureVLS(constants, correlations, liquids=[liquid], gas=gas, solids=[])

T1 = 273.15+30
state_1 = flasher.flash(VF=0, T=T1)
saturation_state_1 = flasher.flash(T=-20+273.15, VF=1)
# Wording is unclear for state 2 but thermodynamically his is what makes sense
state_2 = flasher.flash(H=state_1.H(), P=saturation_state_1.P)
# Check the flash lowers the pressure
assert state_2.P < state_1.P
state_3 = flasher.flash(P=state_2.P, VF=1)
saturation_state_2 = flasher.flash(T=30+273.15, VF=1)
state_4 = flasher.flash(P=saturation_state_2.P, S=state_3.S())
states = [state_1, state_2, state_3, state_4]

condensation_duty = (state_1.H() - state_4.H())
heating_duty = state_3.H() - state_2.H()
compressing_duty = state_4.H() - state_3.H()
condensation_duty, heating_duty, compressing_duty
```

[1]:
```
(-17242.594866461008, 13841.52397663936, 3401.0708898216462)
```

[2]:
```
# Check the cycle convergence
cycle_error = sum([condensation_duty, heating_duty, compressing_duty])
cycle_error
```

[2]:
```
-9.094947017729282e-13
```

[3]:
```
for state in states:
    print(f'T={state.T:.2f} K, P={state.P:.2f} Pa, VF={state.VF:.2f}, S={state.S():.2f}␣
↪J/(mol*K), H={state.H():.2f} J/(mol)')
```

```
T=303.15 K, P=746445.43 Pa, VF=0.00, S=-72.22 J/(mol*K), H=-17223.28 J/(mol)
T=253.15 K, P=152387.52 Pa, VF=0.29, S=-70.06 J/(mol*K), H=-17223.28 J/(mol)
T=253.15 K, P=152387.52 Pa, VF=1.00, S=-15.38 J/(mol*K), H=-3381.75 J/(mol)
T=312.16 K, P=746445.43 Pa, VF=1.00, S=-15.38 J/(mol*K), H=19.32 J/(mol)
```

8.20 Problem 14.15 Joule-Thomson Coefficient for Methane Using the Peng-Robinson EOS

Calculate the Joule-Thomson coefficient of methane at 300 K and 30 bar, using the Peng Robinson model.

8.20.1 Solution

This is straightforward.

[1]:
```
# Set the conditions and imports
from scipy.constants import bar
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CEOSGas, FlashPureVLS
fluid = 'methane'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T = 300
P = 30*bar
zs = [1]

eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,
                    eos_kwargs=eos_kwargs)
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,
              eos_kwargs=eos_kwargs)
flasher = FlashPureVLS(constants, correlations, liquids=[liquid], gas=gas, solids=[])

res = flasher.flash(T=T, P=P, zs=zs)
print(f'The JT coefficient at the specified conditions is {res.Joule_Thomson():.4g} K/Pa
↪')
```

```
The JT coefficient at the specified conditions is 4.652e-06 K/Pa
```

8.21 Problem 14.16 Compressor Duty and State Properties after Ammonia Compression

Ammonia at 100 °C and 5 bar is compressed to a pressure of 10 bar. The thermal efficiency of the process is 0.8; and the mechanical efficiency is 0.9. What is the compressor duty per mole and the temperature of the outlet?

8.21.1 Solution

This is just another compression problem.

[1]:
```
# Set the conditions and imports
from scipy.constants import bar
from thermo import ChemicalConstantsPackage, PRMIX, CEOSLiquid, CEOSGas, FlashPureVLS
fluid = 'ammonia'
constants, correlations = ChemicalConstantsPackage.from_IDs([fluid])

T1 = 100 + 273.15
P1 = 5*bar
P2 = 10*bar
zs = [1]

eta_isentropic = 0.8
eta_mechanical = 0.9

eos_kwargs = dict(Tcs=constants.Tcs, Pcs=constants.Pcs, omegas=constants.omegas)
liquid = CEOSLiquid(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,
                    eos_kwargs=eos_kwargs)
gas = CEOSGas(PRMIX, HeatCapacityGases=correlations.HeatCapacityGases,
              eos_kwargs=eos_kwargs)
flasher = FlashPureVLS(constants, correlations, liquids=[liquid], gas=gas, solids=[])

state_1 = flasher.flash(T=T1, P=P1)
state_2_ideal = flasher.flash(S=state_1.S(), P=P2)
# Compute the change in enthalpy
delta_H_ideal = (state_2_ideal.H()-state_1.H())
H_added_to_fluid_actual = delta_H_ideal/eta_isentropic

state_2 = flasher.flash(H=state_1.H() + H_added_to_fluid_actual, P=P2)

specific_power = (state_2.H() - state_1.H())/(eta_mechanical)
print(f'The actual power is {specific_power:.0f} W/mol')
print(f'The actual outlet temperature is {state_2.T: .2f} K')
```

```
The actual power is 3148 W/mol
The actual outlet temperature is  448.20 K
```

CHAPTER NINE

INSTALLATION

Get the latest version of Thermo from https://pypi.python.org/pypi/thermo/

If you have an installation of Python with pip, simple install it with:

> $ pip install thermo

Alternatively, if you are using conda as your package management, you can simply install thermo in your environment from conda-forge channel with:

> $ conda install -c conda-forge thermo

To get the git version, run:

> $ git clone git://github.com/CalebBell/thermo.git

CHAPTER

TEN

LATEST SOURCE CODE

The latest development version of Thermo's sources can be obtained at

> https://github.com/CalebBell/thermo

CHAPTER

ELEVEN

BUG REPORTS

To report bugs, please use the Thermo's Bug Tracker at:

> https://github.com/CalebBell/thermo/issues

If you have further questions about the usage of the library, feel free to contact the author at Caleb.Andrew.Bell@gmail.com.

CHAPTER TWELVE

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 Thermo license, if it is convenient for you, please cite Thermo if used in your work. Please also consider contributing any changes you make back, and benefit the community.

CHAPTER

THIRTEEN

CITATION

To cite Thermo in publications use:

```
Caleb Bell and Contributors (2016-2021). Thermo: Chemical properties component of␣
↪Chemical Engineering Design Library (ChEDL)
https://github.com/CalebBell/thermo.
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[2] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] 14:00-17:00. "ISO 10780:1994." ISO. Accessed March 29, 2021. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/01/88/18855.html.

[2] 14:00-17:00. "ISO 13443:1996." ISO. Accessed March 29, 2021. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/04/20461.html.

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Laliberte, Marc. "A Model for Calculating the Heat Capacity of Aqueous Solutions, with Updated Density and Viscosity Data." Journal of Chemical & Engineering Data 54, no. 6 (June 11, 2009): 1725-60. doi:10.1021/je8008123

[1] Magomedov, U. B. "The Thermal Conductivity of Binary and Multicomponent Aqueous Solutions of Inorganic Substances at High Parameters of State." High Temperature 39, no. 2 (March 1, 2001): 221-26. doi:10.1023/A:1017518731726.

[1] Magomedov, U. B. "The Thermal Conductivity of Binary and Multicomponent Aqueous Solutions of Inorganic Substances at High Parameters of State." High Temperature 39, no. 2 (March 1, 2001): 221-26. doi:10.1023/A:1017518731726.

[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] McCleskey, R. Blaine. "Electrical Conductivity of Electrolytes Found In Natural Waters from (5 to 90) °C." Journal of Chemical & Engineering Data 56, no. 2 (February 10, 2011): 317-27. doi:10.1021/je101012n.

[1] Chen, Chau-Chyun, H. I. Britt, J. F. Boston, and L. B. Evans. "Local Composition Model for Excess Gibbs Energy of Electrolyte Systems. Part I: Single Solvent, Single Completely Dissociated Electrolyte Systems." AIChE Journal 28, no. 4 (July 1, 1982): 588-96. doi:10.1002/aic.690280410

[2] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Speight, James. Lange's Handbook of Chemistry. 16 edition. McGraw-Hill Professional, 2005.

[1] Arcis, Hugues, Jane P. Ferguson, Jenny S. Cox, and Peter R. Tremaine. "The Ionization Constant of Water at Elevated Temperatures and Pressures: New Data from Direct Conductivity Measurements and Revised Formulations from T = 273 K to 674 K and p = 0.1 MPa to 31 MPa." Journal of Physical and Chemical Reference Data 49, no. 3 (July 23, 2020): 033103. https://doi.org/10.1063/1.5127662.

[1] Bandura, Andrei V., and Serguei N. Lvov. "The Ionization Constant of Water over Wide Ranges of Temperature and Density." Journal of Physical and Chemical Reference Data 35, no. 1 (March 1, 2006): 15-30. doi:10.1063/1.1928231

[1] Bandura, Andrei V., and Serguei N. Lvov. "The Ionization Constant of Water over Wide Ranges of Temperature and Density." Journal of Physical and Chemical Reference Data 35, no. 1 (March 1, 2006): 15-30. doi:10.1063/1.1928231

[1] Marshall, William L., and E. U. Franck. "Ion Product of Water Substance, 0-1000 degree C, 1010,000 Bars New International Formulation and Its Background." Journal of Physical and Chemical Reference Data 10, no. 2 (April 1, 1981): 295-304. doi:10.1063/1.555643.

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Soave, G. "Direct Calculation of Pure-Compound Vapour Pressures through Cubic Equations of State." Fluid Phase Equilibria 31, no. 2 (January 1, 1986): 203-7. doi:10.1016/0378-3812(86)90013-0.

[1] Thorade, Matthis, and Ali Saadat. "Partial Derivatives of Thermodynamic State Properties for Dynamic Simulation." Environmental Earth Sciences 70, no. 8 (April 10, 2013): 3497-3503. doi:10.1007/s12665-013-2394-z.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Johansson, Fredrik. Mpmath: A Python Library for Arbitrary-Precision Floating-Point Arithmetic, 2010.

[1] Peng, Ding-Yu, and Donald B. Robinson. "A New Two-Constant Equation of State." Industrial & Engineering Chemistry Fundamentals 15, no. 1 (February 1, 1976): 59-64. doi:10.1021/i160057a011.

[2] Robinson, Donald B., Ding-Yu Peng, and Samuel Y-K Chung. "The Development of the Peng - Robinson Equation and Its Application to Phase Equilibrium in a System Containing Methanol." Fluid Phase Equilibria 24, no. 1 (January 1, 1985): 25-41. doi:10.1016/0378-3812(85)87035-7.

[3] Privat, R., and J.-N. Jaubert. "PPR78, a Thermodynamic Model for the Prediction of Petroleum Fluid-Phase Behaviour," 11. EDP Sciences, 2011. doi:10.1051/jeep/201100011.

[1] Robinson, Donald B, and Ding-Yu Peng. The Characterization of the Heptanes and Heavier Fractions for the GPA Peng-Robinson Programs. Tulsa, Okla.: Gas Processors Association, 1978.

[2] Robinson, Donald B., Ding-Yu Peng, and Samuel Y-K Chung. "The Development of the Peng - Robinson Equation and Its Application to Phase Equilibrium in a System Containing Methanol." Fluid Phase Equilibria 24, no. 1 (January 1, 1985): 25-41. doi:10.1016/0378-3812(85)87035-7.

[1] Stryjek, R., and J. H. Vera. "PRSV: An Improved Peng-Robinson Equation of State for Pure Compounds and Mixtures." The Canadian Journal of Chemical Engineering 64, no. 2 (April 1, 1986): 323-33. doi:10.1002/cjce.5450640224.

[2] Stryjek, R., and J. H. Vera. "PRSV - An Improved Peng-Robinson Equation of State with New Mixing Rules for Strongly Nonideal Mixtures." The Canadian Journal of Chemical Engineering 64, no. 2 (April 1, 1986): 334-40. doi:10.1002/cjce.5450640225.

[3] Stryjek, R., and J. H. Vera. "Vapor-liquid Equilibrium of Hydrochloric Acid Solutions with the PRSV Equation of State." Fluid Phase Equilibria 25, no. 3 (January 1, 1986): 279-90. doi:10.1016/0378-3812(86)80004-8.

[4] Proust, P., and J. H. Vera. "PRSV: The Stryjek-Vera Modification of the Peng-Robinson Equation of State. Parameters for Other Pure Compounds of Industrial Interest." The Canadian Journal of Chemical Engineering 67, no. 1 (February 1, 1989): 170-73. doi:10.1002/cjce.5450670125.

[1] Stryjek, R., and J. H. Vera. "PRSV2: A Cubic Equation of State for Accurate Vapor-liquid Equilibria Calculations." The Canadian Journal of Chemical Engineering 64, no. 5 (October 1, 1986): 820-26. doi:10.1002/cjce.5450640516.

[1] Twu, Chorng H., John E. Coon, and John R. Cunningham. "A New Generalized Alpha Function for a Cubic Equation of State Part 1. Peng-Robinson Equation." Fluid Phase Equilibria 105, no. 1 (March 15, 1995): 49-59. doi:10.1016/0378-3812(94)02601-V.

[1] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Twu, Chorng H., David Bluck, John R. Cunningham, and John E. Coon. "A Cubic Equation of State with a New Alpha Function and a New Mixing Rule." Fluid Phase Equilibria 69 (December 10, 1991): 33-50. doi:10.1016/0378-3812(91)90024-2.

[1] Le Guennec, Yohann, Romain Privat, and Jean-Noël Jaubert. "Development of the Translated-Consistent Tc-PR and Tc-RK Cubic Equations of State for a Safe and Accurate Prediction of Volumetric, Energetic and Saturation Properties of Pure Compounds in the Sub- and Super-Critical Domains." Fluid Phase Equilibria 429 (December 15, 2016): 301-12. https://doi.org/10.1016/j.fluid.2016.09.003.

[1] Pina-Martinez, Andrés, Romain Privat, Jean-Noël Jaubert, and Ding-Yu Peng. "Updated Versions of the Generalized Soave -Function Suitable for the Redlich-Kwong and Peng-Robinson Equations of State." Fluid Phase Equilibria, December 7, 2018. https://doi.org/10.1016/j.fluid.2018.12.007.

[1] Soave, Giorgio. "Equilibrium Constants from a Modified Redlich-Kwong Equation of State." Chemical Engineering Science 27, no. 6 (June 1972): 1197-1203. doi:10.1016/0009-2509(72)80096-4.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Twu, Chorng H., John E. Coon, and John R. Cunningham. "A New Generalized Alpha Function for a Cubic Equation of State Part 2. Redlich-Kwong Equation." Fluid Phase Equilibria 105, no. 1 (March 15, 1995): 61-69. doi:10.1016/0378-3812(94)02602-W.

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Le Guennec, Yohann, Romain Privat, and Jean-Noël Jaubert. "Development of the Translated-Consistent Tc-PR and Tc-RK Cubic Equations of State for a Safe and Accurate Prediction of Volumetric, Energetic and Saturation Properties of Pure Compounds in the Sub- and Super-Critical Domains." Fluid Phase Equilibria 429 (December 15, 2016): 301-12. https://doi.org/10.1016/j.fluid.2016.09.003.

[1] Pina-Martinez, Andrés, Romain Privat, Jean-Noël Jaubert, and Ding-Yu Peng. "Updated Versions of the Generalized Soave -Function Suitable for the Redlich-Kwong and Peng-Robinson Equations of State." Fluid Phase Equilibria, December 7, 2018. https://doi.org/10.1016/j.fluid.2018.12.007.

[1] Soave, G. "Rigorous and Simplified Procedures for Determining the Pure-Component Parameters in the Redlich—Kwong—Soave Equation of State." Chemical Engineering Science 35, no. 8 (January 1, 1980): 1725-30. https://doi.org/10.1016/0009-2509(80)85007-X.

[2] Sandarusi, Jamal A., Arthur J. Kidnay, and Victor F. Yesavage. "Compilation of Parameters for a Polar Fluid Soave-Redlich-Kwong Equation of State." Industrial & Engineering Chemistry Process Design and Development 25, no. 4 (October 1, 1986): 957-63. https://doi.org/10.1021/i200035a020.

[3] Valderrama, Jose O., Héctor De la Puente, and Ahmed A. Ibrahim. "Generalization of a Polar-Fluid Soave-Redlich-Kwong Equation of State." Fluid Phase Equilibria 93 (February 11, 1994): 377-83. https://doi.org/10.1016/0378-3812(94)87021-7.

[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.

[1] Redlich, Otto., and J. N. S. Kwong. "On the Thermodynamics of Solutions. V. An Equation of State. Fugacities of Gaseous Solutions." Chemical Reviews 44, no. 1 (February 1, 1949): 233-44. doi:10.1021/cr60137a013.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Smith, J. M, H. C Van Ness, and Michael M Abbott. Introduction to Chemical Engineering Thermodynamics. Boston: McGraw-Hill, 2005.

[1] Hu, Jiawen, Rong Wang, and Shide Mao. "Some Useful Expressions for Deriving Component Fugacity Coefficients from Mixture Fugacity Coefficient." Fluid Phase Equilibria 268, no. 1-2 (June 25, 2008): 7-13. doi:10.1016/j.fluid.2008.03.007.

[2] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Watson, Harry A. J., and Paul I. Barton. "Reliable Flash Calculations: Part 3. A Nonsmooth Approach to Density Extrapolation and Pseudoproperty Evaluation." Industrial & Engineering Chemistry Research, November 11, 2017. https://doi.org/10.1021/acs.iecr.7b03233.

[2] Mathias P. M., Boston J. F., and Watanasiri S. "Effective Utilization of Equations of State for Thermodynamic Properties in Process Simulation." AIChE Journal 30, no. 2 (June 17, 2004): 182-86. https://doi.org/10.1002/aic.690300203.

[1] Peng, Ding-Yu, and Donald B. Robinson. "A New Two-Constant Equation of State." Industrial & Engineering Chemistry Fundamentals 15, no. 1 (February 1, 1976): 59-64. doi:10.1021/i160057a011.

[2] Robinson, Donald B., Ding-Yu Peng, and Samuel Y-K Chung. "The Development of the Peng - Robinson Equation and Its Application to Phase Equilibrium in a System Containing Methanol." Fluid Phase Equilibria 24, no. 1 (January 1, 1985): 25-41. doi:10.1016/0378-3812(85)87035-7.

[1] Chang, Yih-Bor. "Development and Application of an Equation of State Compositional Simulator" 1990. https://repositories.lib.utexas.edu/handle/2152/80585.

[1] Peng, Ding-Yu, and Donald B. Robinson. "A New Two-Constant Equation of State." Industrial & Engineering Chemistry Fundamentals 15, no. 1 (February 1, 1976): 59-64. doi:10.1021/i160057a011.

[2] Robinson, Donald B., Ding-Yu Peng, and Samuel Y-K Chung. "The Development of the Peng - Robinson Equation and Its Application to Phase Equilibrium in a System Containing Methanol." Fluid Phase Equilibria 24, no. 1 (January 1, 1985): 25-41. doi:10.1016/0378-3812(85)87035-7.

[1] Stryjek, R., and J. H. Vera. "PRSV: An Improved Peng-Robinson Equation of State for Pure Compounds and Mixtures." The Canadian Journal of Chemical Engineering 64, no. 2 (April 1, 1986): 323-33. doi:10.1002/cjce.5450640224.

[2] Stryjek, R., and J. H. Vera. "PRSV - An Improved Peng-Robinson Equation of State with New Mixing Rules for Strongly Nonideal Mixtures." The Canadian Journal of Chemical Engineering 64, no. 2 (April 1, 1986): 334-40. doi:10.1002/cjce.5450640225.

[3] Stryjek, R., and J. H. Vera. "Vapor-liquid Equilibrium of Hydrochloric Acid Solutions with the PRSV Equation of State." Fluid Phase Equilibria 25, no. 3 (January 1, 1986): 279-90. doi:10.1016/0378-3812(86)80004-8.

[4] Proust, P., and J. H. Vera. "PRSV: The Stryjek-Vera Modification of the Peng-Robinson Equation of State. Parameters for Other Pure Compounds of Industrial Interest." The Canadian Journal of Chemical Engineering 67, no. 1 (February 1, 1989): 170-73. doi:10.1002/cjce.5450670125.

[1] Stryjek, R., and J. H. Vera. "PRSV2: A Cubic Equation of State for Accurate Vapor-liquid Equilibria Calculations." The Canadian Journal of Chemical Engineering 64, no. 5 (October 1, 1986): 820-26. doi:10.1002/cjce.5450640516.

[1] Twu, Chorng H., John E. Coon, and John R. Cunningham. "A New Generalized Alpha Function for a Cubic Equation of State Part 1. Peng-Robinson Equation." Fluid Phase Equilibria 105, no. 1 (March 15, 1995): 49-59. doi:10.1016/0378-3812(94)02601-V.

[1] Peng, Ding-Yu, and Donald B. Robinson. "A New Two-Constant Equation of State." Industrial & Engineering Chemistry Fundamentals 15, no. 1 (February 1, 1976): 59-64. doi:10.1021/i160057a011.

[2] Robinson, Donald B., Ding-Yu Peng, and Samuel Y-K Chung. "The Development of the Peng - Robinson Equation and Its Application to Phase Equilibrium in a System Containing Methanol." Fluid Phase Equilibria 24, no. 1 (January 1, 1985): 25-41. doi:10.1016/0378-3812(85)87035-7.

[1] Le Guennec, Yohann, Romain Privat, and Jean-Noël Jaubert. "Development of the Translated-Consistent Tc-PR and Tc-RK Cubic Equations of State for a Safe and Accurate Prediction of Volumetric, Energetic and Saturation Properties of Pure Compounds in the Sub- and Super-Critical Domains." Fluid Phase Equilibria 429 (December 15, 2016): 301-12. https://doi.org/10.1016/j.fluid.2016.09.003.

[1] Pina-Martinez, Andrés, Romain Privat, Jean-Noël Jaubert, and Ding-Yu Peng. "Updated Versions of the Generalized Soave -Function Suitable for the Redlich-Kwong and Peng-Robinson Equations of State." Fluid Phase Equilibria, December 7, 2018. https://doi.org/10.1016/j.fluid.2018.12.007.

[1] Soave, Giorgio. "Equilibrium Constants from a Modified Redlich-Kwong Equation of State." Chemical Engineering Science 27, no. 6 (June 1972): 1197-1203. doi:10.1016/0009-2509(72)80096-4.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Twu, Chorng H., John E. Coon, and John R. Cunningham. "A New Generalized Alpha Function for a Cubic Equation of State Part 2. Redlich-Kwong Equation." Fluid Phase Equilibria 105, no. 1 (March 15, 1995): 61-69. doi:10.1016/0378-3812(94)02602-W.

[1] API Technical Data Book: General Properties & Characterization. American Petroleum Institute, 7E, 2005.

[1] Le Guennec, Yohann, Romain Privat, and Jean-Noël Jaubert. "Development of the Translated-Consistent Tc-PR and Tc-RK Cubic Equations of State for a Safe and Accurate Prediction of Volumetric, Energetic and Saturation Properties of Pure Compounds in the Sub- and Super-Critical Domains." Fluid Phase Equilibria 429 (December 15, 2016): 301-12. https://doi.org/10.1016/j.fluid.2016.09.003.

[1] Soave, G. "Rigorous and Simplified Procedures for Determining the Pure-Component Parameters in the Redlich—Kwong—Soave Equation of State." Chemical Engineering Science 35, no. 8 (January 1, 1980): 1725-30. https://doi.org/10.1016/0009-2509(80)85007-X.

[2] Sandarusi, Jamal A., Arthur J. Kidnay, and Victor F. Yesavage. "Compilation of Parameters for a Polar Fluid Soave-Redlich-Kwong Equation of State." Industrial & Engineering Chemistry Process Design and Development 25, no. 4 (October 1, 1986): 957-63. https://doi.org/10.1021/i200035a020.

[3] Valderrama, Jose O., Héctor De la Puente, and Ahmed A. Ibrahim. "Generalization of a Polar-Fluid Soave-Redlich-Kwong Equation of State." Fluid Phase Equilibria 93 (February 11, 1994): 377-83. https://doi.org/10.1016/0378-3812(94)87021-7.

[1] Holderbaum, T., and J. Gmehling. "PSRK: A Group Contribution Equation of State Based on UNIFAC." Fluid Phase Equilibria 70, no. 2-3 (December 30, 1991): 251-65. https://doi.org/10.1016/0378-3812(91)85038-V.

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Walas, Stanley M. Phase Equilibria in Chemical Engineering. Butterworth-Heinemann, 1985.

[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. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Zhi, Yun, and Huen Lee. "Fallibility of Analytic Roots of Cubic Equations of State in Low Temperature Region." Fluid Phase Equilibria 201, no. 2 (September 30, 2002): 287-94. https://doi.org/10.1016/S0378-3812(02)00072-9.

[1] Reid, Robert C.; Prausnitz, John M.; Poling, Bruce E. Properties of Gases and Liquids. McGraw-Hill Companies, 1987.

[1] Johansson, Fredrik. Mpmath: A Python Library for Arbitrary-Precision Floating-Point Arithmetic, 2010.

[1] Meurer, Aaron, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, and Sartaj Singh. "SymPy: Symbolic Computing in Python." PeerJ Computer Science 3 (2017): e103.

[1] Almeida, G. S., M. Aznar, and A. S. Telles. "Uma Nova Forma de Dependência Com a Temperatura Do Termo Atrativo de Equações de Estado Cúbicas." RBE, Rev. Bras. Eng., Cad. Eng. Quim 8 (1991): 95.

[1] Androulakis, I. P., N. S. Kalospiros, and D. P. Tassios. "Thermophysical Properties of Pure Polar and Nonpolar Compounds with a Modified VdW-711 Equation of State." Fluid Phase Equilibria 45, no. 2 (April 1, 1989): 135-63. doi:10.1016/0378-3812(89)80254-7.

[1] Chen, Zehua, and Daoyong Yang. "Optimization of the Reduced Temperature Associated with Peng–Robinson Equation of State and Soave-Redlich-Kwong Equation of State To Improve Vapor Pressure Prediction for Heavy Hydrocarbon Compounds." Journal of Chemical & Engineering Data, August 31, 2017. doi:10.1021/acs.jced.7b00496.

[1] Coquelet, C., A. Chapoy, and D. Richon. "Development of a New Alpha Function for the Peng–Robinson Equation of State: Comparative Study of Alpha Function Models for Pure Gases (Natural Gas Components) and Water-Gas Systems." International Journal of Thermophysics 25, no. 1 (January 1, 2004): 133-58. doi:10.1023/B:IJOT.0000022331.46865.2f.

[1] Gasem, K. A. M, W Gao, Z Pan, and R. L Robinson Jr. "A Modified Temperature Dependence for the Peng-Robinson Equation of State." Fluid Phase Equilibria 181, no. 1–2 (May 25, 2001): 113-25. doi:10.1016/S0378-3812(01)00488-5.

[1] Gibbons, Richard M., and Andrew P. Laughton. "An Equation of State for Polar and Non-Polar Substances and Mixtures" 80, no. 9 (January 1, 1984): 1019-38. doi:10.1039/F29848001019.

[1] Haghtalab, A., M. J. Kamali, S. H. Mazloumi, and P. Mahmoodi. "A New Three-Parameter Cubic Equation of State for Calculation Physical Properties and Vapor-liquid Equilibria." Fluid Phase Equilibria 293, no. 2 (June 25, 2010): 209-18. doi:10.1016/j.fluid.2010.03.029.

[1] Harmens, A., and H. Knapp. "Three-Parameter Cubic Equation of State for Normal Substances." Industrial & Engineering Chemistry Fundamentals 19, no. 3 (August 1, 1980): 291-94. doi:10.1021/i160075a010.

[1] Heyen, G. Liquid and Vapor Properties from a Cubic Equation of State. In "Proceedings of the 2nd International Conference on Phase Equilibria and Fluid Properties in the Chemical Industry". DECHEMA: Frankfurt, 1980; p 9-13.

[1] Mathias, Paul M. "A Versatile Phase Equilibrium Equation of State." Industrial & Engineering Chemistry Process Design and Development 22, no. 3 (July 1, 1983): 385-91. doi:10.1021/i200022a008.

[1] Mathias, Paul M., and Thomas W. Copeman. "Extension of the Peng-Robinson Equation of State to Complex Mixtures: Evaluation of the Various Forms of the Local Composition Concept." Fluid Phase Equilibria 13 (January 1, 1983): 91-108. doi:10.1016/0378-3812(83)80084-3.

[1] Mathias, Paul M., and Thomas W. Copeman. "Extension of the Peng-Robinson Equation of State to Complex Mixtures: Evaluation of the Various Forms of the Local Composition Concept." Fluid Phase Equilibria 13 (January 1, 1983): 91-108. doi:10.1016/0378-3812(83)80084-3.

[1] Melhem, Georges A., Riju Saini, and Bernard M. Goodwin. "A Modified Peng-Robinson Equation of State." Fluid Phase Equilibria 47, no. 2 (August 1, 1989): 189-237. doi:10.1016/0378-3812(89)80176-1.

[1] Saffari, Hamid, and Alireza Zahedi. "A New Alpha-Function for the Peng-Robinson Equation of State: Application to Natural Gas." Chinese Journal of Chemical Engineering 21, no. 10 (October 1, 2013): 1155-61. doi:10.1016/S1004-9541(13)60581-9.

[1] J. Schwartzentruber, H. Renon, and S. Watanasiri, "K-values for Non-Ideal Systems:An Easier Way," Chem. Eng., March 1990, 118-124.

[1] Soave, Giorgio. "Equilibrium Constants from a Modified Redlich- Kwong Equation of State." Chemical Engineering Science 27, no. 6 (June 1972): 1197-1203. doi:10.1016/0009-2509(72)80096-4.

[2] Young, André F., Fernando L. P. Pessoa, and Victor R. R. Ahón. "Comparison of 20 Alpha Functions Applied in the Peng–Robinson Equation of State for Vapor Pressure Estimation." Industrial & Engineering Chemistry Research 55, no. 22 (June 8, 2016): 6506-16. doi:10.1021/acs.iecr.6b00721.

[1] Soave, G. "Improvement of the Van Der Waals Equation of State." Chemical Engineering Science 39, no. 2 (January 1, 1984): 357-69. doi:10.1016/0009-2509(84)80034-2.

[1] Soave, G. "Rigorous and Simplified Procedures for Determining the Pure-Component Parameters in the Redlich—Kwong—Soave Equation of State." Chemical Engineering Science 35, no. 8 (January 1, 1980): 1725-30. https://doi.org/10.1016/0009-2509(80)85007-X.

[1] Soave, G. "Improving the Treatment of Heavy Hydrocarbons by the SRK EOS." Fluid Phase Equilibria 84 (April 1, 1993): 339-42. doi:10.1016/0378-3812(93)85131-5.

[1] Trebble, M. A., and P. R. Bishnoi. "Development of a New Four- Parameter Cubic Equation of State." Fluid Phase Equilibria 35, no. 1 (September 1, 1987): 1-18. doi:10.1016/0378-3812(87)80001-8.

[1] Twu, Chorng H., David Bluck, John R. Cunningham, and John E. Coon. "A Cubic Equation of State with a New Alpha Function and a New Mixing Rule." Fluid Phase Equilibria 69 (December 10, 1991): 33-50. doi:10.1016/0378-3812(91)90024-2.

[1] Yu, Jin-Min, and Benjamin C. -Y. Lu. "A Three-Parameter Cubic Equation of State for Asymmetric Mixture Density Calculations." Fluid Phase Equilibria 34, no. 1 (January 1, 1987): 1-19. doi:10.1016/0378-3812(87)85047-1.

[1] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Michelsen, Michael L., and Jørgen M. Mollerup. Thermodynamic Models: Fundamentals & Computational Aspects. Tie-Line Publications, 2007.

[2] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Michelsen, Michael L., and Jørgen M. Mollerup. Thermodynamic Models: Fundamentals & Computational Aspects. Tie-Line Publications, 2007.

[2] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[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] 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.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[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.

[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] 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] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[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.

[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] 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.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[2] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[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] 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] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[1] Government of Canada.. "Substances Lists" Feb 11, 2015. https://www.ec.gc.ca/subsnouvelles-newsubs/default.asp?n=47F768FE-1.

[2] US EPA. "TSCA Chemical Substance Inventory." Accessed April 2016. https://www.epa.gov/tsca-inventory.

[3] ECHA. "EC Inventory". Accessed March 2015. http://echa.europa.eu/information-on-chemicals/ec-inventory.

[4] SPIN. "SPIN Substances in Products In Nordic Countries." Accessed March 2015. http://195.215.202.233/DotNetNuke/default.aspx.

[1] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Renon, Henri, and J. M. Prausnitz. "Local Compositions in Thermodynamic Excess Functions for Liquid Mixtures." AIChE Journal 14, no. 1 (1968): 135-144. doi:10.1002/aic.690140124.

[2] Gmehling, Jurgen, Barbel Kolbe, Michael Kleiber, and Jurgen Rarey. Chemical Thermodynamics for Process Simulation. 1st edition. Weinheim: Wiley-VCH, 2012.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[1] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[3] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[4] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[5] Gharagheizi, Farhad, Poorandokht Ilani-Kashkouli, William E. Acree Jr., Amir H. Mohammadi, and Deresh Ramjugernath. "A Group Contribution Model for Determining the Vaporization Enthalpy of Organic Compounds at the Standard Reference Temperature of 298 K." Fluid Phase Equilibria 360 (December 25, 2013): 279-92. doi:10.1016/j.fluid.2013.09.021.

[6] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[7] 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. doi:10.1016/j.fluid.2016.10.013.

[1] Gharagheizi, Farhad, Poorandokht Ilani-Kashkouli, William E. Acree Jr., Amir H. Mohammadi, and Deresh Ramjugernath. "A Group Contribution Model for Determining the Sublimation Enthalpy of Organic Compounds at the Standard Reference Temperature of 298 K." Fluid Phase Equilibria 354 (September 25, 2013): 265-doi:10.1016/j.fluid.2013.06.046.

[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] Shen, V.K., Siderius, D.W., Krekelberg, W.P., and Hatch, H.W., Eds., NIST WebBook, NIST, http://doi.org/10.18434/T4M88Q

[1] Bennett, Jim, and Kurt A. G. Schmidt. "Comparison of Phase Identification Methods Used in Oil Industry Flow Simulations." Energy & Fuels 31, no. 4 (April 20, 2017): 3370-79. https://doi.org/10.1021/acs.energyfuels.6b02316.

[1] Bennett, Jim, and Kurt A. G. Schmidt. "Comparison of Phase Identification Methods Used in Oil Industry Flow Simulations." Energy & Fuels 31, no. 4 (April 20, 2017): 3370-79. https://doi.org/10.1021/acs.energyfuels.6b02316.

[1] Chang, Yih-Bor. "Development and Application of an Equation of State Compositional Simulator," 1990. https://repositories.lib.utexas.edu/handle/2152/80585.

[1] Poling, Bruce E., Edward A. Grens, and John M. Prausnitz. "Thermodynamic Properties from a Cubic Equation of State: Avoiding Trivial Roots and Spurious Derivatives." Industrial & Engineering Chemistry Process Design and Development 20, no. 1 (January 1, 1981): 127-30. https://doi.org/10.1021/i200012a019.

[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.

[1] Bennett, Jim, and Kurt A. G. Schmidt. "Comparison of Phase Identification Methods Used in Oil Industry Flow Simulations." Energy & Fuels 31, no. 4 (April 20, 2017): 3370-79. https://doi.org/10.1021/acs.energyfuels.6b02316.

[1] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[3] Elliott, J., and Carl Lira. Introductory Chemical Engineering Thermodynamics. 2nd edition. Upper Saddle River, NJ: Prentice Hall, 2012.

[4] Kooijman, Harry A., and Ross Taylor. The ChemSep Book. Books on Demand Norderstedt, Germany, 2000.

[1] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[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] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[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] 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., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[2] Fredenslund, Aage, Russell L. Jones, and John M. Prausnitz. "Group Contribution Estimation of Activity Coefficients in Nonideal Liquid Mixtures." AIChE Journal 21, no. 6 (November 1, 1975): 1086-99. doi:10.1002/aic.690210607.

[3] Jakob, Antje, Hans Grensemann, Jürgen Lohmann, and Jürgen Gmehling. "Further Development of Modified UNIFAC (Dortmund): Revision and Extension 5." Industrial & Engineering Chemistry Research 45, no. 23 (November 1, 2006): 7924-33. doi:10.1021/ie060355c.

[4] Kang, Jeong Won, Vladimir Diky, and Michael Frenkel. "New Modified UNIFAC Parameters Using Critically Evaluated Phase Equilibrium Data." Fluid Phase Equilibria 388 (February 25, 2015): 128-41. doi:10.1016/j.fluid.2014.12.042.

[5] Jäger, Andreas, Ian H. Bell, and Cornelia Breitkopf. "A Theoretically Based Departure Function for Multi-Fluid Mixture Models." Fluid Phase Equilibria 469 (August 15, 2018): 56-69. https://doi.org/10.1016/j.fluid.2018.04.015.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[2] Fredenslund, Aage, Russell L. Jones, and John M. Prausnitz. "Group Contribution Estimation of Activity Coefficients in Nonideal Liquid Mixtures." AIChE Journal 21, no. 6 (November 1, 1975): 1086-99. doi:10.1002/aic.690210607.

[1] Gmehling, Jurgen. Chemical Thermodynamics: For Process Simulation. Weinheim, Germany: Wiley-VCH, 2012.

[1] Wei, James, Morton M. Denn, John H. Seinfeld, Arup Chakraborty, Jackie Ying, Nicholas Peppas, and George Stephanopoulos. Molecular Modeling and Theory in Chemical Engineering. Academic Press, 2001.

[1] Wei, James, Morton M. Denn, John H. Seinfeld, Arup Chakraborty, Jackie Ying, Nicholas Peppas, and George Stephanopoulos. Molecular Modeling and Theory in Chemical Engineering. Academic Press, 2001.

[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] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[4] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[5] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[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] 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.

[1] Viswanath, Dabir S., and G. Natarajan. Databook On The Viscosity Of Liquids. New York: Taylor & Francis, 1989

[2] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[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] 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] Trachenko, K., and V. V. Brazhkin. "Minimal Quantum Viscosity from Fundamental Physical Constants." Science Advances, April 2020. https://doi.org/10.1126/sciadv.aba3747.

[1] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[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.

[4] Trachenko, K., and V. V. Brazhkin. "Minimal Quantum Viscosity from Fundamental Physical Constants." Science Advances, April 2020. https://doi.org/10.1126/sciadv.aba3747.

[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] 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] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[6] Gesellschaft, V. D. I., ed. VDI Heat Atlas. 2nd edition. Berlin; New York:: Springer, 2010.

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[2] Bell, Ian H., Jorrit Wronski, Sylvain Quoilin, and Vincent Lemort. "Pure and Pseudo-Pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp." Industrial & Engineering Chemistry Research 53, no. 6 (February 12, 2014): 2498-2508. doi:10.1021/ie4033999. http://www.coolprop.org/

[1] Haynes, W.M., Thomas J. Bruno, and David R. Lide. CRC Handbook of Chemistry and Physics. [Boca Raton, FL]: CRC press, 2014.

[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] Smith, H. C. Van Ness Joseph M. Introduction to Chemical Engineering Thermodynamics 4th Edition, Joseph M. Smith, H. C. Van Ness, 1987.

[2] Kooijman, Harry A., and Ross Taylor. The ChemSep Book. Books on Demand Norderstedt, Germany, 2000.

[3] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Wilson, Grant M. "Vapor-Liquid Equilibrium. XI. A New Expression for the Excess Free Energy of Mixing." Journal of the American Chemical Society 86, no. 2 (January 1, 1964): 127-130. doi:10.1021/ja01056a002.

[2] Gmehling, Jurgen, Barbel Kolbe, Michael Kleiber, and Jurgen Rarey. Chemical Thermodynamics for Process Simulation. 1st edition. Weinheim: Wiley-VCH, 2012.

[1] Poling, Bruce E., John M. Prausnitz, and John P. O'Connell. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[2] Gmehling, Jürgen, Michael Kleiber, Bärbel Kolbe, and Jürgen Rarey. Chemical Thermodynamics for Process Simulation. John Wiley & Sons, 2019.

[1] Abrams, Denis S., and John M. Prausnitz. "Statistical Thermodynamics of Liquid Mixtures: A New Expression for the Excess Gibbs Energy of Partly or Completely Miscible Systems." AIChE Journal 21, no. 1 (January 1, 1975): 116-28. doi:10.1002/aic.690210115.

[2] Gmehling, Jurgen, Barbel Kolbe, Michael Kleiber, and Jurgen Rarey. Chemical Thermodynamics for Process Simulation. 1st edition. Weinheim: Wiley-VCH, 2012.

[3] Maurer, G., and J. M. Prausnitz. "On the Derivation and Extension of the Uniquac Equation." Fluid Phase Equilibria 2, no. 2 (January 1, 1978): 91-99. doi:10.1016/0378-3812(78)85002-X.

[1] Joback, Kevin G. "A Unified Approach to Physical Property Estimation Using Multivariate Statistical Techniques." Thesis, Massachusetts Institute of Technology, 1984.

[2] 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.

[1] Fedors, R. F. "A Method to Estimate Critical Volumes." AIChE Journal 25, no. 1 (1979): 202-202. https://doi.org/10.1002/aic.690250129.

[2] Green, Don, and Robert Perry. Perry's Chemical Engineers' Handbook, Eighth Edition. McGraw-Hill Professional, 2007.

[1] 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.

[2] Poling, Bruce E. The Properties of Gases and Liquids. 5th edition. New York: McGraw-Hill Professional, 2000.

[3] Yan, Xinjian, Qian Dong, and Xiangrong Hong. "Reliability Analysis of Group-Contribution Methods in Predicting Critical Temperatures of Organic Compounds." Journal of Chemical & Engineering Data 48, no. 2 (March 1, 2003): 374-80. https://doi.org/10.1021/je025596f.

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V