realgas

Release 1.1.2

Robert F. De Jaco

Jan 20, 2021

CONTENTS:

Simple integration of real-gas effects into open source Chemical Engineering modeling and simulation environments.

Chemical processes involving gas-phases, such as membrane separations, adsorption separations, and heterogeneous catalysis, are ubiquitous. These processes are typically modeled assuming that the gas-phase is ideal and that component heat capacities can be approximated by constant effective values. The purpose of this software package is to allow these assumptions to be relaxed by automating the tediousness of a real gas formulation.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

The package can be installed via `pip` as follows

```
pip install realgas
```

1.2 Single-Component Examples

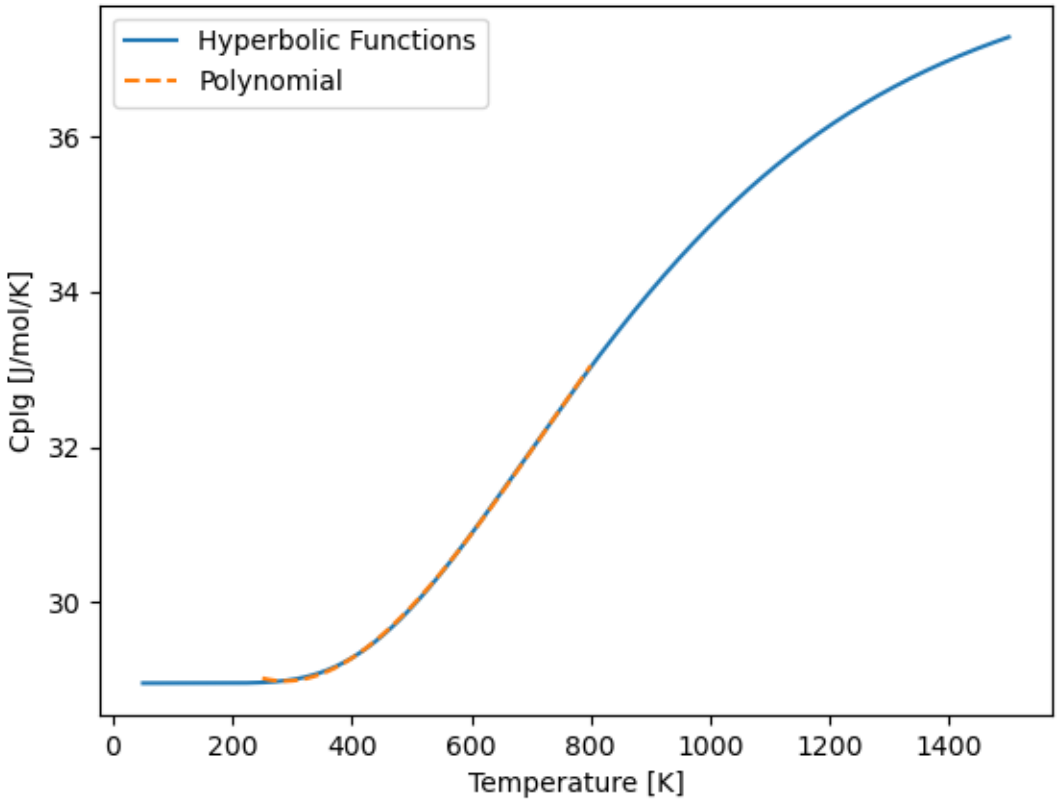
1.2.1 Heat Capacity

Determine the temperature dependence of the ideal gas heat capacity, C_p^{IG} for water

And we will get something that looks like the following

```
>>> import matplotlib.pyplot as plt
>>> from realgas.cp import CpIdealGas
>>> I = CpIdealGas(compound_name='Air', T_min_fit=250., T_max_fit=800., poly_order=3)
>>> I.eval(300.), I.Cp_units
(29.00369, 'J/mol/K')
>>> I.eval(300.)/I.MW
1.001508
>>> # we can then plot and visualize the resutls
>>> fig, ax = I.plot()
>>> fig.savefig('docs/source/air.png')
>>> del I
```

And we will get something that looks like the following

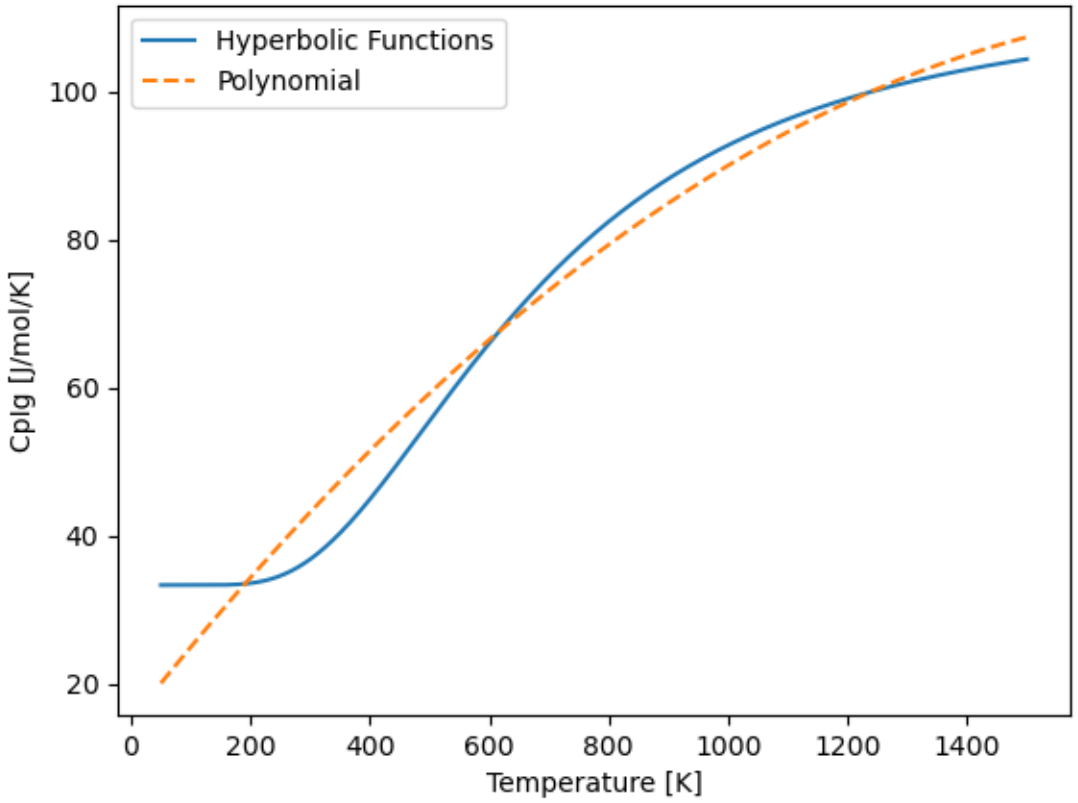


and we notice that the polynomial (orange dashed lines) fits the hyperbolic function well.

Automatically tries to raise errors if fit is not good enough:

```
>>> from realgas.cp import CpIdealGas
>>> I = CpIdealGas(compound_name='Methane')
Traceback (most recent call last):
    ...
Exception: Fit is too poor (not in (0.99,1)) too large!
Try using a smaller temperature range for fitting
and/or increasing the number of fitting points and polynomial degree.
See error path in error-*dir
```

This will lead to an error directory with a figure saved in it that looks like the following:



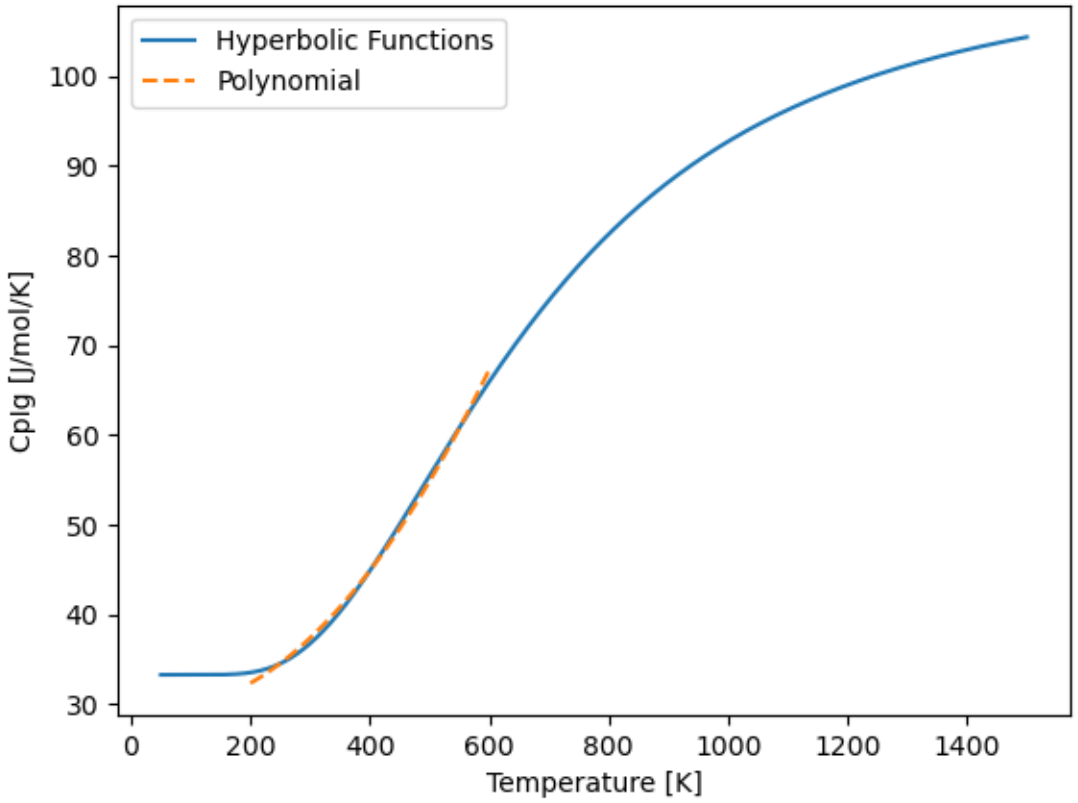
Usually, we wont need an accurate function over this entire temperature range. Lets imagine that we are interested instead in a temperature interval between 200 and 600 K. In this case

```
>>> I = CpIdealGas(compound_name='Methane', T_min_fit=200., T_max_fit=600.)
```

And then we can save the results to a file

```
>>> fig = plt.figure()
>>> fig, ax = I.plot(fig=fig)
>>> fig.savefig('docs/source/cp-methane-fixed.png')
```

Which leads to a much better fit, as shown below



1.2.2 Cubic Equations of State

```
>>> from realgas.eos.cubic import PengRobinson, RedlichKwong, SoaveRedlichKwong
>>> P = 8e5  # Pa
>>> T = 300. # K
>>> PengRobinson(compound_name='Propane').iterate_to_solve_Z(P=P, T=T)
0.85682
>>> RedlichKwong(compound_name='Propane').iterate_to_solve_Z(P=P, T=T)
0.87124
>>> cls_srk = SoaveRedlichKwong(compound_name='Propane')
>>> Z = cls_srk.iterate_to_solve_Z(P=P, T=T)
>>> Z
0.86528
>>> # calculate residual properties
>>> from chem_util.chem_constants import gas_constant as R
>>> V = Z*R*T/P
>>> cls_srk.S_R_R_expr(P, V, T)
-0.30028
>>> cls_srk.H_R_RT_expr(P, V, T)
-0.42714
>>> cls_srk.G_R_RT_expr(P, V, T) - cls_srk.H_R_RT_expr(P, V, T) + cls_srk.S_R_R_
↪expr(P, V, T)
0.0
>>> P = 8e5  # Pa
```

(continues on next page)

(continued from previous page)

```
>>> T = 300. # K
>>> PengRobinson(compound_name='Propane').iterate_to_solve_Z(P=P, T=T)
0.85682
>>> RedlichKwong(compound_name='Propane').iterate_to_solve_Z(P=P, T=T)
0.87124
>>> cls_srk = SoaveRedlichKwong(compound_name='Propane')
>>> Z = cls_srk.iterate_to_solve_Z(P=P, T=T)
>>> Z
0.86528
>>> # calculate residual properties
>>> from chem_util.chem_constants import gas_constant as R
>>> V = Z*R*T/P
>>> cls_srk.S_R_R_expr(P, V, T)
-0.30028
>>> cls_srk.H_R_RT_expr(P, V, T)
-0.42714
>>> cls_srk.G_R_RT_expr(P, V, T) - cls_srk.H_R_RT_expr(P, V, T) + cls_srk.S_R_R_
↪expr(P, V, T)
0.0
```

1.2.3 Virial Equation of State

```
>>> from realgas.eos.virial import SecondVirial
>>> Iv2 = SecondVirial(compound_name='Propane')
>>> Iv2.calc_Z_from_units(P=8e5, T=300.)
0.87260
>>> Iv2 = SecondVirial(compound_name='Propane')
>>> Iv2.calc_Z_from_units(P=8e5, T=300.)
0.87260
```

1.2.4 Other Utilities

Determine whether a single real root of the cubic equation of state can be used for simple computational implementation. In some regimes, the cubic equation of state only has 1 real root–in this case, the compressibility factor can be obtained easily.

```
>>> from realgas.eos.cubic import PengRobinson
>>> pr = PengRobinson(compound_name='Propane')
>>> pr.num_roots(300., 5e5)
3
>>> pr.num_roots(100., 5e5)
1
```

Input custom thermodynamic critical properties

```
>>> from realgas.eos.cubic import PengRobinson
>>> dippr = PengRobinson(compound_name='Methane')
>>> custom = PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                       T_c=191.4, V_c=0.0001, Z_c=0.286, w=0.0115, MW=16.042, P_
↪c=0.286*8.314*191.4/0.0001)
>>> dippr.iterate_to_solve_Z(T=300., P=8e5)
0.9828233
```

(continues on next page)

(continued from previous page)

```
>>> custom.iterate_to_solve_Z(T=300., P=8e5)
0.9823877
```

If we accidentally input the wrong custom units, it is likely that *realgas.critical_constants.CriticalConstants* will catch it.

```
>>> from realgas.eos.cubic import PengRobinson
>>> PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                         T_c=273.-191.4, V_c=0.0001, Z_c=0.286, w=0.0115, MW=16.042,␣
↪P_c=0.286*8.314*191.4/0.0001)
Traceback (most recent call last):
...
AssertionError: Percent difference too high for T_c
>>> PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                         T_c=191.4, V_c=0.0001*100., Z_c=0.286, w=0.0115, MW=16.042,␣
↪P_c=0.286*8.314*191.4/0.0001)
Traceback (most recent call last):
...
AssertionError: Percent difference too high for V_c
>>> PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                         T_c=191.4, V_c=0.0001, Z_c=2.86, w=0.0115, MW=16.042, P_c=0.
↪286*8.314*191.4/0.0001)
Traceback (most recent call last):
...
AssertionError: Percent difference too high for Z_c
>>> PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                         T_c=191.4, V_c=0.0001, Z_c=0.286, w=1.115, MW=16.042, P_c=0.
↪286*8.314*191.4/0.0001)
Traceback (most recent call last):
...
AssertionError: Percent difference too high for w
>>> PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                         T_c=191.4, V_c=0.0001, Z_c=0.286, w=0.0115, MW=18.042, P_
↪c=0.286*8.314*191.4/0.0001)
Traceback (most recent call last):
...
AssertionError: Percent difference too high for MW
>>> PengRobinson(compound_name='Methane', cas_number='72-28-8',
...                         T_c=191.4, V_c=0.0001, Z_c=0.286, w=0.0115, MW=18.042, P_
↪c=0.286*0.008314*191.4/0.0001)
Traceback (most recent call last):
...
AssertionError: Percent difference too high for P_c
```

It performs the checks by comparing to the DIPPR [RWO+07] database and asserting that the values are within a reasonable tolerance

1.3 Mixture Examples

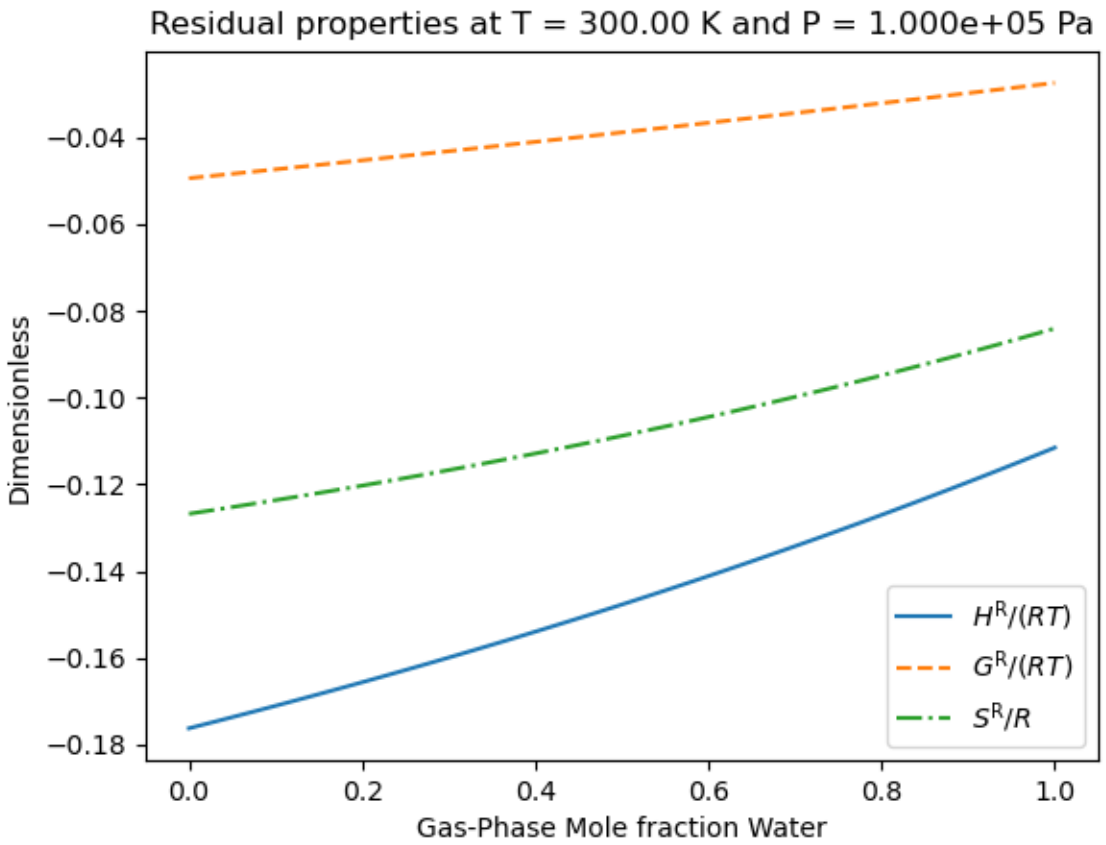
Note: For non-ideal gases, currently only implemented for virial equation of state

1.3.1 Residual Properties

Below, an example is shown for calculating residual properties of THF/Water mixtures

```
>>> from realgas.eos.virial import SecondVirialMixture
>>> P, T = 1e5, 300.
>>> mixture = SecondVirialMixture(compound_names=['Water', 'Tetrahydrofuran'], k_ij=0.
↪)
>>> import matplotlib.pyplot as plt
>>> fig, ax = mixture.plot_residual_HSG(P, T)
>>> fig.savefig('docs/source/THF-WATER.png')
```

So that the results look like the following



We note that the residual properties will not always vanish in the limit of pure components like excess properties since the pure-components may not be perfect gases.

1.3.2 Partial Molar Properties

```
>>> from realgas.partial_molar_properties import Mixture
>>> cp_kwargs = dict(T_min_fit=200., T_max_fit=600.)
>>> I = Mixture(
...     [dict(compound_name='Methane', **cp_kwargs), dict(compound_name='Ethane',␣
↪**cp_kwargs)],
...      compound_names=['Methane', 'Ethane'],
...      ideal=False,
...     )
>>> I.T_cs
[190.564, 305.32]
>>> I.cas_numbers
['74-82-8', '74-84-0']
```

The reference state is the pure component at

```
>>> cp_kwargs = dict(T_min_fit=200., T_max_fit=600.)
>>> I = Mixture(
...     [dict(compound_name='Methane', **cp_kwargs), dict(compound_name='Ethane',␣
↪**cp_kwargs)],
...      compound_names=['Methane', 'Ethane'],
...      ideal=False,
...     )
>>> I.T_cs
[190.564, 305.32]
>>> I.cas_numbers
['74-82-8', '74-84-0']
```

The reference state is the pure component at $P = 0$ and $T = T_{\text{ref}}$. The reference temperature is T_{ref} and defaults to 0 K. But different values can be used, as shown below

```
>>> I.enthalpy(ys=[0.5, 0.5], P=1e5, T=300.)
9037.3883
>>> I.enthalpy(ys=[0.5, 0.5], P=1e5, T=300., T_ref=0.)
9037.3883
>>> I.enthalpy(ys=[0.5, 0.5], P=1e5, T=300., T_ref=300.)
-31.33905
>>> I.ideal = True
>>> I.enthalpy(ys=[0.5, 0.5], P=1e5, T=300., T_ref=300.)
0.0
```

And we observe that the enthalpy can be non-zero for real gases when the reference temperature is chosen to be the same as the temperature of interest, since the enthalpy departure function is non-zero.

However, for a real gas,

```
>>> I.ideal = False
```

in the limit that the gas has low pressure and high temperature,

```
>>> I.enthalpy(ys=[0.5, 0.5], P=1., T=500., T_ref=500.)
-0.000111958
```

In the limit that the gas becomes a pure mixture, we recover the limit that $\bar{H}_i^{\text{pure}} = H^{\text{pure}}$ or $\bar{H}_i^{\text{pure}} - H^{\text{pure}} = 0$.

```
>>> kwargs = dict(ys=[1., 0.], P=1e5, T=300.)
>>> I.enthalpy(**kwargs)-I.bar_Hi(I.cas_numbers[0], **kwargs)
0.0
>>> kwargs = dict(ys=[0., 1.], P=1e5, T=300.)
>>> I.enthalpy(**kwargs)-I.bar_Hi(I.cas_numbers[1], **kwargs)
0.0
```

Using the second order virial equation of state we can perform these same calculations on multicomponent mixtures, as shown below

Note: all units are SI units, so the enthalpy here is in J/mol

```
>>> cp_kwargs = dict(T_min_fit=200., T_max_fit=600.)
>>> M = Mixture(
...     [dict(compound_name='Methane', **cp_kwargs),
...      dict(compound_name='Ethane', **cp_kwargs), dict(compound_name='Ethylene',␣
↪**cp_kwargs),
...      dict(compound_name='Carbon dioxide', **cp_kwargs)],
...      compound_names=['Methane', 'Ethane', 'Ethylene', 'Carbon dioxide'],
...      ideal=False,
...     )
>>> M.enthalpy(ys=[0.1, 0.2, 0.5, 0.2], P=10e5, T=300.)
7432.66593
>>> M.enthalpy(ys=[1.0, 0.0, 0.0, 0.0], P=10e5, T=300.) - M.bar_Hi(M.cas_numbers[0],␣
↪ys=[1.0, 0.0, 0.0, 0.0], P=10e5, T=300.)
0.0
```

Another simple check is to ensure that we get the same answer regardless of the order of the compounds

```
>>> N = Mixture(
...     [dict(compound_name='Ethane', **cp_kwargs),
...      dict(compound_name='Methane', **cp_kwargs), dict(compound_name='Ethylene',␣
↪**cp_kwargs),
...      dict(compound_name='Carbon dioxide', **cp_kwargs)],
...      compound_names=['Ethane', 'Methane', 'Ethylene', 'Carbon dioxide'],
...      ideal=False,
...     )
>>> M.enthalpy(ys=[0.4, 0.3, 0.17, 0.13], P=5e5, T=300.) - N.enthalpy(ys=[0.3, 0.4, 0.
↪17, 0.13], P=5e5, T=300.)
0.0
```

And that, further, a mixture with an extra component that is not present (mole fraction 0.) converges to an $N-1$ mixture

```
>>> Nm1 = Mixture(  # take out CO2
...     [dict(compound_name='Ethane', **cp_kwargs),
...      dict(compound_name='Methane', **cp_kwargs), dict(compound_name='Ethylene',␣
↪**cp_kwargs)],
...      compound_names=['Ethane', 'Methane', 'Ethylene'],
...      ideal=False,
...     )
>>> N.enthalpy(ys=[0.4, 0.3, 0.3, 0.], P=5e5, T=300.) - Nm1.enthalpy(ys=[0.4, 0.3, 0.
↪3], P=5e5, T=300.)
0.0
```

1.4 Gotchas

- All units are SI units

CHAPTER

TWO

DEFINITIONS

Residual properties for a given thermodynamic property M are defined as

$$M = M^{\mathrm{IG}} + M^{\mathrm{R}}$$

where M^{IG} is the value of the property in the ideal gas state and M^{R} is the residual value of the property.

More information on residual properties can be found in standard texts [SVanNessA05]

2.1 Nomenclature

Code	Symbol	Description
P	P	Pressure in Pa
V	V	Molar Volume in $\mathrm{m}^3/\mathrm{mol}$
R	R	gas constant SI units ($\mathrm{m}^3 \times \mathrm{Pa}/\mathrm{mol}/\mathrm{K}$)
T	T	temperature in K
T_c	T_{c}	critical temperature in K
P_c	P_{c}	critical pressure in Pa
T_r	T_{r}	reduced temperature (dimensionless)
V_c	V_{c}	Critical volume $\mathrm{m}^3/\mathrm{mol}$
w	ω	Accentric factor
y_i	y_i	mole fraction of component *i*
–	n_i	number of moles of component *i*
S	S	Molar entropy
H	H	Molar enthalpy
G	G	Molar Gibbs free energy

CHAPTER

THREE

PARTIAL MOLAR PROPERTIES

We **define** partial molar property $\bar{M}_i$ of species *i* in a mixture as

$$\bar{M}_i = \left(\frac{\partial(nM)}{\partial n_i}\right)_{P,T,n_j} \tag{3.1}$$

The mixture property is related to the partial molar property as

$$nM = \sum_i n_i \bar{M}_i$$

or, in terms of gas-phase mole fractions y_i,

$$M = \sum_i y_i \bar{M}_i$$

The following relationships also hold

$$\bar{M}_i = \bar{M}^{\mathrm{IG}} + \bar{M}^{\mathrm{R}} \tag{3.2}$$

$$M^{\mathrm{R}} = \sum_i y_i \bar{M}^{\mathrm{R}} \tag{3.3}$$

3.1 Ideal Gas

The *Gibbs theorem* is

> A partial molar property (other than volume) of a contituent species in an ideal-gas mixture is equal to the corresponding molar property of the species as a pure ideal gas at the mixture temperature but at a pressure equal to its partial pressure in the mixture

The partial molar volume of an ideal gas, $\bar{V}_i^{\mathrm{IG}}$, is

$$\bar{V}_i^{\mathrm{IG}} = \frac{RT}{P} \tag{3.4}$$

The partial molar enthalpy of an ideal gas, $\bar{H}_i^{\mathrm{IG}}$, is

$$\bar{H}_i^{\mathrm{IG}} = H_i^{\mathrm{IG}}$$

which results from the enthalpy of an ideal gas being independent of pressure. Therefore, we can compute the ideal gas partial molar enthalpy if we have the ideal gas heat capacities, as follows

$$\bar{H}_i^{\mathrm{IG}} = H_i^{\mathrm{IG}} = \int_{T_{\mathrm{ref}}}^{T} C_{\mathrm{p},i}^{\mathrm{IG}} \mathrm{d}T' \tag{3.5}$$

where T_{ref} is a reference temperature and T' is a dummy variable for integration. Often times, we want to compute these quantities in dimensionless units,

$$\bar{H}_i^{\text{IG},\star} = \frac{\bar{H}_i^{\text{IG}}}{RT_{\text{ref}}} \tag{3.6}$$

$$= \int_1^{T^\star} \frac{C_{\text{p},i}^{\text{IG}}}{R} \mathrm{d}T' \tag{3.7}$$

(3.8)

or

$$\bar{H}_i^{\text{IG},\star} = \int_1^{T^\star} C_{\text{p},i}^{\text{IG},\star} \mathrm{d}T' \tag{3.9}$$

where

$$T^\star = T/T_{\text{ref}}$$

is a dimensionless variable and

$$C_{\text{p},i}^{\text{IG},\star} = C_{\text{p},i}^{\text{IG}}/R$$

is a dimensionless parameter that is a function of $T^\star$.

We note that the thermodynamic integration reference temperature does not have to be the same as the temperature for scaling, but we have made them the same here for simplicity.

Todo: Implement ideal gas partial molar entropy?? Implement ideal gas partial molar free energy?? This might not be useful though because these values seem to *depend* on mixture properties

---

3.2 Residual

The residual partial molar volume of component *i*, $\bar{V}_i^{\text{R}}$, can be calculated as

$$\bar{V}_i^{\text{R}} = RT \left(\frac{\partial \ln \hat{\phi}_i}{\partial P} \right)_{T,y} \tag{3.10}$$

Defining the dimensionless quantity

$$\bar{V}_i^{\text{R},\star} = \frac{\bar{V}_i^{\text{R}}}{RT_{\text{ref}}}$$

The expression in dimensionless units can be simplified to

$$\bar{V}_i^{\text{R},\star} = T^\star \left(\frac{\partial \ln \hat{\phi}_i}{\partial P} \right)_{T^\star,y} \tag{3.11}$$

The residual partial molar enthalpy of component *i*, $\bar{H}_i^{\text{R}}$, can be calculated as

$$\bar{H}_i^{\text{R}} = -RT^2 \left(\frac{\partial \ln \hat{\phi}_i}{\partial T} \right) \tag{3.12}$$

Defining the dimensionless quantity

$$\bar{H}_i^{\mathrm{R},\star} = \frac{\bar{H}_i^{\mathrm{R}}}{RT_{\mathrm{ref}}} \tag{3.13}$$

$$= -\frac{RT^2}{RT_{\mathrm{ref}}}\left(\frac{\partial \ln \hat{\phi}_i}{\partial T}\right) \tag{3.14}$$

$$= -\frac{RT^2}{RT_{\mathrm{ref}}^2}\left(\frac{\partial \ln \hat{\phi}_i}{\partial T^\star}\right) \tag{3.15}$$

(3.16)

The expression in dimensionless units is computed as

$$\bar{H}_i^{\mathrm{R},\star} = -\left(T^\star\right)^2\left(\frac{\partial \ln \hat{\phi}_i}{\partial T^\star}\right) \tag{3.17}$$

The residual partial molar free energy of component *i*, $\bar{G}_i^{\mathrm{R}}$, which *defines* the fugacity coefficient $\hat{\phi}_i$, is

$$\bar{G}_i^{\mathrm{R}} = RT \ln \hat{\phi}_i \tag{3.18}$$

With these definitions, however, we note that we need an equation of state to calculate the partial molar properties. In this package, the second-order virial equation of state currently implements the necessary derivatives.

class realgas.partial_molar_properties.**Mixture**(*cp_args: List[dict]*, *ideal=True*, ***kwargs*)

Parameters

- **ideal** (*bool, optional*) – whether or not ideal gas, defaults to True
- **kwargs** – key-word arguments for *realgas.eos.virial.SecondVirialMixture*

bar_Hi_IG(*cas_i: str*, *T*, *T_ref=0*)

Parameters

- **T_ref** – reference temperature in K for enthalpy, defaults to 0
- **T** – temperautre in K

Returns $arH_i^e xtIG$, see Equation (3.5)

bar_Vi_IG(*T*, *P*)

Parameters

- **T** – temperature in K
- **P** – pressure in Pa

Returns $arV_i^e xtIG$, see Equation (3.4)

enthalpy(*ys: List[Union[float, Any]]*, *P: float*, *T: float*, *T_ref=0.0*)

Residual property of X for mixture.

Similar to Equation (3.3) but in dimensionless form

Parameters **method** (*callable*) – function to compute partial molar property of compound

class realgas.partial_molar_properties.**MixtureDimensionless**(*cp_args: List[dict]*, *ideal=True*, ***kwargs*)

Todo: add docs!

Parameters

- **ideal** (*bool, optional*) – whether or not ideal gas, defaults to True
- **kwargs** – key-word arguments for *realgas.eos.virial.SecondVirialMixture*

bar_Hi_IG_star(*cas_i*, *T_star*, *T_ref_star=0*)

Parameters

- **T_ref_star** – dimensionless reference temperature in K for enthalpy, defaults to 0
- **T_star** – dimensionless temperature

Returns arH_i^exIG, see Equation (3.9)

enthalpy_star(*ys: List[Union[float, Any]]*, *P: float*, *T: float*)

Residual property of X for mixture.

Similar to Equation (3.3) but in dimensionless form

Parameters **method** (*callable*) – function to compute partial molar property of compound

CHAPTER

FOUR

EQUATIONS OF STATE

4.1 Single Component

4.1.1 Virial

Todo: merge docs with those in `realgas.partial_molar_properties`. There, the definitions of residual properties are displayed and here wee need only write simplified forms for the specific equation of state.

Theory

The second order virial equation of state is [GP07]

$$Z = 1 + B\rho = 1 + \frac{BP}{RT} \tag{4.1}$$

Where the composition dependency of B is given by the *exact* mixing rule

$$B = \sum_i \sum_j y_i y_j B_{ij} \tag{4.2}$$

where $B_{ij} = B_{ji}$, and B_{ii} and B_{jj} are virial coefficients for the pure species

In this package, the useful correlation

$$\frac{BP_\mathrm{c}}{RT_\mathrm{c}} = B^0 + \omega B^1$$

or

$$B = \frac{RT_\mathrm{c}}{P_\mathrm{c}} \left(B^0 + \omega B^1\right) \tag{4.3}$$

So that, combining Equations (4.1) and (4.3), the compressibility can be calculated from dimensionless quantities as

$$Z = 1 + \left(B^0 + \omega B^1\right) \frac{P_\mathrm{r}}{T_\mathrm{r}} \tag{4.4}$$

where [SVanNessA05]

$$B^0 = 0.083 - \frac{0.422}{T_\mathrm{r}^{1.6}} \tag{4.5}$$

$$B^1 = 0.139 - \frac{0.172}{T_\mathrm{r}^{4.2}} \tag{4.6}$$

so that

so that the following derivatives can be computed as

$$\frac{dB^0}{dT_\mathrm{r}} = \frac{0.675}{T_\mathrm{r}^{2.6}} \tag{4.7}$$

$$\frac{dB^1}{dT_\mathrm{r}} = \frac{0.722}{T_\mathrm{r}^{5.2}} \tag{4.8}$$

Which allow the H^R, S^R, and G^R to be readily computed as follows [GP07]

$$\frac{G^\mathrm{R}}{RT} = \left(B^0 + \omega B^1\right)\frac{P_\mathrm{r}}{T_\mathrm{r}} \tag{4.9}$$

$$\frac{H^\mathrm{R}}{RT} = P_\mathrm{r}\left[\frac{B^0}{T_\mathrm{r}} - \frac{\mathrm{d}B^0}{\mathrm{d}T_\mathrm{r}} + \omega\left(\frac{B^1}{T_\mathrm{r}} - \frac{\mathrm{d}B^1}{\mathrm{d}T_\mathrm{r}}\right)\right] \tag{4.10}$$

$$\frac{S^\mathrm{R}}{R} = -P_\mathrm{r}\left(\frac{dB^0}{dT_\mathrm{r}} - \omega\frac{dB^1}{dT_\mathrm{r}}\right) \tag{4.11}$$

The cross coefficients are calculated as

$$B_{ij} = \frac{RT_{\mathrm{c}ij}}{P_{\mathrm{c}ij}}\left(B^0 + \omega_{ij}B^1\right) \tag{4.12}$$

so that the cross derivatives can be computed as

$$\frac{dB_{ij}}{dT} = \frac{RT_{\mathrm{c}ij}}{P_{\mathrm{c}ij}}\left(\frac{dB^0}{dT} + \omega_{ij}\frac{dB^1}{dT}\right)$$

$$\frac{dB_{ij}}{dT} = \frac{R}{P_{\mathrm{c}ij}}\left(\frac{dB^0}{dT_{\mathrm{r}ij}} + \omega_{ij}\frac{dB^1}{dT_{\mathrm{r}ij}}\right)$$

or, equivalently,

$$\frac{dB_{ij}}{dT_{\mathrm{r}ij}} = \frac{RT_{\mathrm{c}ij}}{P_{\mathrm{c}ij}}\left(\frac{dB^0}{dT_{\mathrm{r}ij}} + \omega_{ij}\frac{dB^1}{dT_{\mathrm{r}ij}}\right) \tag{4.13}$$

Fugacity Coefficients

The Fugacity coefficients are calculated as

$$\ln\hat{\phi}_i = \left(\frac{\partial(nG^\mathrm{R}/R/T)}{\partial n_i}\right)_{P,T,n_j}$$

For the virial equation of state, this becomes [VanNessA82]

$$\ln\hat{\phi}_k = \frac{P}{RT}\left[B_{kk} + \frac{1}{2}\sum_i\sum_j y_i y_j\left(2\delta_{ik} - \delta_{ij}\right)\right] \tag{4.14}$$

where *both* i and j indices run over all species

$$\delta_{ik} = 2B_{ik} - B_{ii} - B_{kk} = \delta_{ki} \tag{4.15}$$

and

$$\delta_{ii} = 0$$

Residual Partial Molar Properties

The partial molar residual free energy of component *k* is

$$\frac{\bar{G}_k^{\mathrm{R}}}{RT} = \ln \hat{\phi}_k = \frac{P}{RT}\left[B_{kk} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\delta_{ik} - \delta_{ij}\right)\right] \tag{4.16}$$

The partial molar residual enthalpy of component *k* is

$$\frac{\bar{H}_k^{\mathrm{R}}}{RT} = -T\left(\frac{\partial \ln \hat{\phi}_k}{\partial T}\right)_{P,y} \tag{4.17}$$

$$= -\frac{T}{T_c}\left(\frac{\partial \ln \hat{\phi}_k}{\partial T_r}\right)_{P,y} \tag{4.18}$$

$$= -\frac{T}{T_c}\left\{\frac{P}{RT}\left[\frac{\partial B_{kk}}{\partial T_r} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\frac{\partial \delta_{ik}}{\partial T_r} - \frac{\partial \delta_{ij}}{\partial T_r}\right)\right] - \frac{T_c \ln \hat{\phi}_i}{T}\right\} \tag{4.19}$$

where $\frac{\partial \delta_{ij}}{\partial T_r}$ is given by (4.47) so that we obtain

$$\frac{\bar{H}_k^{\mathrm{R}}}{RT} = -\frac{P}{RT_{\mathrm{c}}}\left[\frac{\partial B_{kk}}{\partial T_r} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\frac{\partial \delta_{ik}}{\partial T_r} - \frac{\partial \delta_{ij}}{\partial T_r}\right)\right] + \ln \hat{\phi}_i \tag{4.20}$$

Since

$$G^{\mathrm{R}} = H^{\mathrm{R}} - TS^{\mathrm{R}}$$

In terms of partial molar properties, then

$$\bar{S}_i^{\mathrm{R}} = \frac{\bar{H}_i^{\mathrm{R}} - \bar{G}_i^{\mathrm{R}}}{T} \tag{4.21}$$

$$\frac{\bar{S}_i^{\mathrm{R}}}{R} = \frac{\bar{H}_i^{\mathrm{R}}}{RT} - \frac{\bar{G}_i^{\mathrm{R}}}{RT} \tag{4.22}$$

(4.23)

By comparing Equation (4.16) and (4.20) it is observed that

$$\frac{\bar{S}_k^{\mathrm{R}}}{R} = -\frac{P}{RT_{\mathrm{c}}}\left[\frac{\partial B_{kk}}{\partial T_r} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\frac{\partial \delta_{ik}}{\partial T_r} - \frac{\partial \delta_{ij}}{\partial T_r}\right)\right] \tag{4.24}$$

where $\frac{\partial \delta_{ij}}{\partial T_r}$ is given by (4.47)

The partial molar residual volume of component *i* is calculated as

$$\frac{\bar{V}_k^{\mathrm{R}}}{RT} = \left(\frac{\partial \ln \hat{\phi}_i}{\partial P}\right)_{T,y} \tag{4.25}$$

$$= \frac{\partial}{\partial P}\left\{\frac{P}{RT}\left[B_{kk} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\delta_{ik} - \delta_{ij}\right)\right]\right\} \tag{4.26}$$

which simplifies to

$$\frac{\bar{V}_k^{\mathrm{R}}}{RT} = \frac{1}{RT}\left[B_{kk} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\delta_{ik} - \delta_{ij}\right)\right] \tag{4.27}$$

from which we obtain the intuitive result of

$$\bar{V}_k^{\mathrm{R}} = B_{kk} + \frac{1}{2}\sum_i \sum_j y_i y_j \left(2\delta_{ik} - \delta_{ij}\right)$$

class realgas.eos.virial.**Virial**(*pow: callable = <ufunc 'power'>, exp: callable = <ufunc 'exp'>*)

Parameters

- **pow** (*callable, optional*) – function for computing power, defaults to numpy.power
- **exp** (*callable, optional*) – function for computing logarithm, defaults to numpy.exp

B0_expr(*T_r*)

Parameters **T_r** – Reduced temperature

Returns Equation (4.5)

B1_expr(*T_r*)

Parameters **T_r** – reduced temperature

Returns Equation eq:*B1_expr*

B_expr(*T_r*, *w*, *T_c*, *P_c*)

Parameters

- **T_r** – reduced temperature
- **w** – accentric factor
- **T_c** – critical temperautre [K]
- **P_c** – critical pressure [Pa]

Returns Equation (4.3)

d_B0_d_Tr_expr(*T_r*)

Parameters **T_r** – reduced temperature

Returns Equation (4.7)

d_B1_d_Tr_expr(*T_r*)

Parameters **T_r** – reduced temperature

Returns Equation (4.8)

hat_phi_i_expr(**args*)

expression for fugacity coefficient :returns: $\exp\left(\ln \hat{\phi}_i\right)$

class realgas.eos.virial.**SecondVirial**(*dippr_no: str = None, compound_name: str = None, cas_number: str = None, pow: callable = <ufunc 'power'>, **kwargs*)

Virial equation of state for one component. See [GP07][SVanNessA05]

G_R_RT_expr(*P*, *T*)

Parameters

- **P** – pressure in Pa

- **T** – Temperautre in K

Returns Equation (4.9)

H_R_RT_expr(*P*, *T*)

Parameters

- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.10)

S_R_R_expr(*P*, *T*)

Parameters

- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.11)

calc_Z_from_dimensionless(*P_r*, *T_r*)

Parameters

- **P_r** – reduced pressure, dimensionless
- **T_r** – reduced temperature, dimensionless

Returns Equation (4.4)

calc_Z_from_units(*P*, *T*)

Parameters

- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.1)

ln_hat_phi_k_expr(*P*, *T*)
logarithm of fugacity coefficient

Note: single-component version

In this case, Equation (4.14) simplifies to

$$\ln \hat{\phi}_i = \frac{PB}{RT}$$

Parameters

- **P** (*float*) – pressure in Pa
- **T** (*float*) – temperature in K

plot_Z_vs_P(*T*, *P_min*, *P_max*, *symbol='o'*, *ax=None*, ***kwargs*)
Plot compressibility as a function of pressure

Parameters

- **T** (*float*) – temperature [K]
- **P_min** (*float*) – minimum pressure for plotting [Pa]

- **P_max** (*float*) – maximum pressure for plotting [Pa]
- **phase** (*str*) – phase type (liquid or vapor), defaults to vapor
- **symbol** (*str*) – marker symbol, defaults to 'o'
- **ax** (*plt.axis*) – matplotlib axes for plotting, defaults to None
- **kwargs** – keyword arguments for plotting

4.1.2 Cubic

Theory

The generic cubic equation of state is [GP07]

$$P = \frac{RT}{V-b} - \frac{a(T)}{(V+\epsilon b)(V+\sigma b)}$$

where ϵ and σ are pure numbers (the same for all substances), and $a(T)$ and b are given by the following equations

$$a(T) = \Psi \frac{\alpha(T_r) R^2 T_c^2}{P_c} \tag{4.28}$$

$$b = \Omega \frac{RT_c}{P_c}$$

The compressibility factor can be calculated by solving the following equation

$$Z - \left(1 + \beta - \frac{q\beta(Z-\beta)}{(Z+\epsilon\beta)(Z+\sigma\beta)}\right) = 0 \tag{4.29}$$

where

$$\beta = \Omega \frac{P_r}{T_r} \tag{4.30}$$

and

$$q = \frac{\Psi \alpha(T_r)}{\Omega T_r} \tag{4.31}$$

An iterative routine to calculate Z using Equation (4.29) is implemented, following [GP07], where

$$Z^{new} = 1 + \beta - \frac{q\beta(Z^{old}-\beta)}{(Z^{old}+\epsilon\beta)(Z^{old}+\sigma\beta)} \tag{4.32}$$

that is continued until the following is true

$$\left\| \frac{Z^{new} - Z^{old}}{Z^{new} + Z^{old}} \times 200 \right\| < \text{tol} \tag{4.33}$$

Residual Molar Properties

$$\frac{H^R}{RT} = Z - 1 + \left[\frac{\mathrm{d}\ln\alpha(T_r)}{\mathrm{d}\ln T_r} - 1\right] qI \tag{4.34}$$

$$\frac{G^R}{RT} = Z - 1 + \ln(Z-\beta) - qI \tag{4.35}$$

$$\frac{S^{\mathrm{R}}}{R} = \ln(Z-\beta) + \frac{\mathrm{d}\ln\alpha(T_{\mathrm{r}})}{\mathrm{d}\ln T_{\mathrm{r}}} qI \tag{4.36}$$

where the following functions have been defined

$$I = \frac{1}{\sigma-\epsilon}\ln\left(\frac{Z+\sigma\beta}{Z+\epsilon\beta}\right) \tag{4.37}$$

$$1+\beta-\frac{q\beta(Z-\beta)}{(Z+\epsilon\beta)(Z+\sigma\beta)} \tag{4.38}$$

Fugacity Coefficients

The pure-component fugacity coefficient is *defined* as

$$\frac{G_i^{\mathrm{R}}}{RT} = \ln\hat{\phi}_i$$

So that, from Equation (4.35),

$$\ln\hat{\phi}_i = Z_i - 1 + \ln(Z_i-\beta_i) - q_i I_i \tag{4.39}$$

Mixtures

Warning: Not implemented yet!

Todo: Implement mixtures with cubic equations of state

class realgas.eos.cubic.**Cubic**(*sigma: float*, *epsilon: float*, *Omega: float*, *Psi: float*, *dippr_no: str = None*, *compound_name: str = None*, *cas_number: str = None*, *log: callable = <ufunc 'log'>*, *exp: callable = <ufunc 'exp'>*, *name: str = 'cubic'*, ***kwargs*)

Generic Cubic Equation of State

Parameters

- **sigma** – Parameter defined by specific equation of state, σ
- **epsilon** – Parameter defined by specific equation of state, ϵ
- **Omega** – Parameter defined by specific equation of state, Ω
- **Psi** – Parameter defined by specific equation of state, Ψ
- **tol** – tolerance for iteration (see Equation (4.33)), set to 0.01
- **log** – function for computing natural log, defaults to `numpy.log`
- **exp** – function for computing exponential, defaults to `numpy.exp`

G_R_RT_expr(*P*, *V*, *T*)

Dimensionless residual gibbs

Parameters

- **P** – pressure in Pa
- **V** – molar volume in m**3/mol

- **T** – temperature in K

Returns $\frac{G^R}{RT}$, see Equation (4.35)

H_R_RT_expr(*P*, *V*, *T*)
Dimensionless residual enthalpy

Parameters

- **P** – pressure in Pa
- **V** – molar volume in m**3/mol
- **T** – temperature in K

Returns $\frac{H^R}{RT}$, see Equation (4.34)

I_expr(*P*, *V*, *T*)

Parameters

- **P** – pressure in Pa
- **V** – molar volume in m**3/mol
- **T** – temperature in K

Returns I (see Equation (4.37))

S_R_R_expr(*P*, *V*, *T*)
Dimensionless residual entropy

Parameters

- **P** – pressure in Pa
- **V** – molar volume in m**3/mol
- **T** – temperature in K

Returns $\frac{S^R}{R}$, see Equation (4.36)

Z_right_hand_side(*Z*, *beta*, *q*)
Estimate of compressibility of vapor [GP07], used for iterative methods

Returns RHS of residual, see Equation (4.38)

a_expr(*T*)

Parameters **T** – temperautre in K

Returns $a(T)$ (see Equation (4.28))

alpha_expr(*T_r*)
An empirical expression, specific to a particular form of the equation of state

Parameters **T_r** – reduced temperature (T/Tc), dimensionless

Returns $\alpha(T_r)$ see Table ??

beta_expr(*T*, *P*)

Parameters

- **T** – temperature in K
- **P** – pressure in Pa

Returns β (see Equation (4.30))

cardano_constants(*T*, *P*)

Parameters

- **T** – temperature [T]
- **P** – pressure [Pa]

Returns cardano constants p, q

Return type tuple

coefficients(*T*, *P*)

Polynomial oefficients for cubic equation of state

$$Z^3 c_0 + Z^2 c_1 + Z * c_2 + c_3 = 0$$

Returns (c_0, c_1, c_2, c_3)

d_ln_alpha_d_ln_Tr(*T_r*)

Parameters **T_r** – reduced temperature [dimensionless]

Returns Expression for $\frac{\mathrm{d}\ln\alpha(T_\mathrm{r})}{\mathrm{d}\ln T_\mathrm{r}}$

hat_phi_i_expr(**args*)

expression for fugacity coefficient :returns: $\exp\left(\ln\hat{\phi}_i\right)$

iterate_to_solve_Z(*T*, *P*) → float

Iterate to compute Z using Equation (4.32) util termination condition (Equation (4.33) is met)

Parameters

- **T** – temperature in K
- **P** – pressure in Pa

Returns compressibility factor

ln_hat_phi_k_expr(*P*, *V*, *T*)

Parameters

- **P** – pressure in Pa
- **T** – temperature in K
- **V** – molar volume in m**3/mol

Returns $\ln\hat{\phi}_k$, see Equation $ln_hat_phi_i$

num_roots(*T*, *P*)

Find number of roots

See [ML12][Dei02]

Parameters

- **T** – temperature in K
- **P** – pressure in Pa

Returns number of roots

plot_Z_vs_P(*T: float*, *P_min: float*, *P_max: float*, *symbol='o'*, *ax: matplotlib.pyplot.axis = None*, *fig: matplotlib.pyplot.figure = None*, ***kwargs*)

Plot compressibility as a function of pressure

Parameters

- **T** – temperature [K]
- **P_min** – minimum pressure for plotting [Pa]
- **P_max** – maximum pressure for plotting [Pa]
- **symbol** – marker symbol, defaults to 'o'
- **ax** – matplotlib axes for plotting, defaults to None
- **kwargs** – keyword arguments for plotting

print_roots(*T*, *P*)
Check to see if all conditions have one root

q_expr(*T*)

Parameters **T** – temperautre in K

Returns q (see Equation (4.31))

residual(*P*, *V*, *T*)

Parameters

- **P** – pressure in Pa
- **V** – volume in [mol/m**3]
- **T** – temperature in K

Returns residual for cubic equation of state (Equation (4.29))

class realgas.eos.cubic.**RedlichKwong**(***kwargs*)
Redlich-Kwong Equation of State [RK49]

This Equation of state has the following parameters [SVanNessA05]

Symbol	Value
$\alpha(T_\mathrm{r})$	$1/\sqrt{T_\mathrm{r}}$
σ	1
ϵ	0
Ω	0.08664
Ψ	0.42748

```
>>> from realgas.eos.cubic import RedlichKwong
>>> model = RedlichKwong(compound_name='Propane')
>>> model.sigma
1
>>> model.epsilon
0
>>> model.Omega
0.08664
>>> model.Psi
0.42748
```

alpha_expr(*T_r*)
An empirical expression, specific to a particular form of the equation of state

Parameters **T_r** – reduced temperature (T/Tc), dimensionless

Returns $\alpha(T_r)$ see Table ??

d_ln_alpha_d_ln_Tr(*T_r*)

Parameters **T_r** – reduced temperature [dimensionless]

Returns Expression for $\frac{\mathrm{d}\ln\alpha(T_\mathrm{r})}{\mathrm{d}\ln T_\mathrm{r}}$

class realgas.eos.cubic.**SoaveRedlichKwong**(***kwargs*)

Soave Redlich-Kwong Equation of State [Soa72]

This equation of state has the following parameters

Symbol	Value
$\alpha(T_\mathrm{r})$	$\left[1+f_w(1-\sqrt{T_\mathrm{r}})\right]^2$
σ	1
ϵ	0
Ω	0.08664
Ψ	0.42748

where

$$f_w = 0.480 + 1.574\omega - 0.176\omega^2 \tag{4.40}$$

```
>>> from realgas.eos.cubic import SoaveRedlichKwong
>>> model = SoaveRedlichKwong(compound_name='Water')
>>> model.Omega
0.08664
>>> model.sigma
1
>>> model.epsilon
0
>>> model.Psi
0.42748
>>> model.f_w_rule(0.)
0.48
>>> model.f_w_rule(1.)
1.878
```

Parameters **f_w** (*float, derived*) – empirical expression used in *alpha* [dimensionless], see Equation (4.40)

alpha_expr(*T_r*)

An empirical expression, specific to a particular form of the equation of state

Parameters **T_r** – reduced temperature (T/Tc), dimensionless

Returns $\alpha(T_r)$ see Table ??

d_ln_alpha_d_ln_Tr(*T_r*)

Parameters **T_r** – reduced temperature [dimensionless]

Returns Expression for $\frac{\mathrm{d}\ln\alpha(T_\mathrm{r})}{\mathrm{d}\ln T_\mathrm{r}}$

f_w_rule(*w*)

Parameters **w** – accentric factor

Returns f_w, see Equation (4.40)

class realgas.eos.cubic.**PengRobinson**(***kwargs*)

Peng-Robinson Equation of State [PR76]

This equation of state has the following parameters

Symbol	Value
$\alpha(T_r)$	$\left[1+f_w(1-\sqrt{T_r})\right]^2$
σ	$1+\sqrt{2}$
ϵ	$1-\sqrt{2}$
Ω	0.07780
Ψ	0.45724

where

$$f_w = 0.480 + 1.574\omega - 0.176\omega^2 \tag{4.41}$$

```
>>> from realgas.eos.cubic import PengRobinson
>>> model = PengRobinson(compound_name='Water')
>>> model.Omega
0.0778
>>> model.sigma
2.4142135
>>> model.epsilon
-0.414213
>>> model.Psi
0.45724
>>> model.f_w_rule(0.)
0.37464
>>> model.f_w_rule(1.)
1.64698
```

Parameters **f_w** (*float, derived*) – empirical expression used in *alpha* [dimensionless?]

f_w_rule(*w*)

Parameters **w** – accentric factor

Returns f_w, see Equation (4.41)

4.2 Multicomponent

4.2.1 Virial

class realgas.eos.virial.**MixingRule**(*pow: callable = <ufunc 'power'>, exp: callable = <ufunc 'exp'>*)

Van der Walls mixing rule

combining rules from [PLdeAzevedo86]

$$\omega_{ij} = \frac{\omega_i + \omega_j}{2} \tag{4.42}$$

$$T_{cij} = \sqrt{T_{ci}T_{cj}}(1-k_{ij}) \tag{4.43}$$

$$P_{cij} = \frac{Z_{cij} R T_{cij}}{V_{cij}} \tag{4.44}$$

$$Z_{cij} = \frac{Z_{ci} + Z_{cj}}{2} \tag{4.45}$$

$$V_{cij} = \left(\frac{V_{ci}^{1/3} + V_{cj}^{1/3}}{2} \right)^3 \tag{4.46}$$

P_cij_rule(*Z_ci*, *V_ci*, *T_ci*, *Z_cj*, *V_cj*, *T_cj*, *k_ij*)

Returns Equation (4.44)

T_cij_rule(*T_ci*, *T_cj*, *k_ij*)

Parameters

- **T_ci** – critical temperature of component i [K]
- **T_cj** – ** j [K]
- **k_ij** – k_ij parameter

Returns Equation (4.43)

V_cij_rule(*V_ci*, *V_cj*)

Parameters

- **V_ci** – critical molar volume of component *i* [m**3/mol]
- **V_cj** – critical molar volume of component *j* [m**3/mol]

Returns Equation eq:*Vc_combine*

Z_cij_rule(*Z_ci*, *Z_cj*)

Parameters

- **Z_ci** – critical compressibility factor of component *i*
- **Z_cj** – critical compressibility factor of component *j*

Returns Equation (4.45)

w_ij_rule(*w_i*, *w_j*)

Parameters

- **w_i** – accentric factor of component *i*
- **w_j** – accentric factor of component *j*

Returns Equation (4.42)

class realgas.eos.virial.**SecondVirialMixture**(*pow: callable = <ufunc 'power'>, exp: callable = <ufunc 'exp'>, **kwargs*)

Second virial with mixing rule from *MixingRule*

Note: can only input both custom critical properties or both from DIPPR–cant have mixed at the moment

Parameters

- **pow** – function to calculate power, defaults to numpy.power
- **exp** – function to calculate exponential,d efaults to numpy.exp

- **kwargs** – key-word arguments to pass to *RealMixture*

B_ij_expr(*i: int*, *j: int*, *T*)

Parameters

- **i** – index of first component
- **j** – index of second component
- **T** – temperature [K]

Returns Equation (4.12)

B_mix_expr(*y_k: List[Union[float, Any]]*, *T*)

Parameters

- **y_k** – mole fractions of each component k
- **T** – temperature in K

Returns Equation (4.2)

G_R_RT(**args*)
Residual free energy of mixture $G^{\mathrm{R}}/R/T$

H_R_RT(**args*)
Residual enthalpy of mixture $H^{\mathrm{R}}/R/T$

M_R_dimensionless(*method: callable*, *ys: List[Union[float, Any]]*, *P: float*, *T: float*)
Residual property of X for mixture.

Similar to Equation (3.3) but in dimensionless form

Parameters **method** (*callable*) – function to compute partial molar property of compound

S_R_R(**args*)
Residual entropy of mixture S^{R}/R

bar_GiR_RT(*cas_k: str*, *ys: List[Union[float, Any]]*, *P*, *T*)
Dimensionless residual partial molar free energy of component i

Parameters

- **cas_k** – cas number of component k
- **ys** – mole fractions
- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.16)

bar_HiR_RT(*cas_k: str*, *ys: List[Union[float, Any]]*, *P*, *T*)
Dimensionless residual partial molar enthalpy of component i

Parameters

- **cas_k** – cas number for component of interest
- **ys** – mole fractions
- **P** – pressure in Pa
- **T** – temperature in K

bar_HiR_star(*T_star*, *cas_k: str*, *ys: List[Union[float, Any]]*, *P*, *T*)
Returns the scaled entropy useful for computations $\bar{H}_i^{R,\star}$, as defined in Equation (3.17)

Parameters

- **cas_k** – cas number of component k
- **ys** – mole fractions
- **P** – pressure in Pa
- **T** – temperature in K

bar_SiR_R(*cas_k: str*, *ys: List[Union[float, Any]]*, *P*, *T*)
Dimensionless residual partial molar entropy of component i

Parameters

- **cas_k** – cas number for component of interest
- **ys** – mole fractions
- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.24)

bar_ViR_RT(*cas_k: str*, *ys: List[Union[float, Any]]*, *P*, *T*)
residual Partial molar volume for component *i*

Note: This expression does not depend on P

Parameters

- **cas_k** – cas number for component of interest
- **ys** – mole fractions
- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.27)

calc_Z_from_units(*y_k: List*, *P*, *T*)

Parameters

- **y_k** – mole fractions of each component k
- **P** – pressure in Pa
- **T** – temperature in K

Returns Equation (4.1)

d_Bij_d_Trij(*i: int*, *j: int*, *T*)

Parameters

- **i** – index for component *i*
- **j** – index for componen t *j*
- **T** – temperature in K

Returns Equation (4.13)

d_dij_d_Tr(*i*, *j*, *T*)

$$\frac{\partial \delta_{ij}}{\partial T_{\mathrm{r}ij}} = 2\frac{\partial B_{ij}}{\partial T_{\mathrm{r}ij}} - \frac{\partial B_{ii}}{\partial T_{\mathrm{r}ij}} - \frac{\partial B_{jj}}{\partial T_{\mathrm{r}ij}} \tag{4.47}$$

Todo: test this with symbolic differentiation of d_ik expression

Parameters

- **i** – index for component *i*
- **j** – index for componen t *j*
- **T** – temperature [K]

d_ik_expr(*i: int*, *k: int*, *T*)

Parameters

- **i** – index of component *i*
- **k** – index of component *k*
- **T** – temperature [K]

Returns Equation (4.15)

fugacity_i_expr(*cas_i: str*, *ys: List[Union[float, Any]]*, *P*, *T*)
Fugacity of component i in mixture $f_i = \hat{\phi}_i y_i P$

Parameters

- **cas_i** – cas number for component of interest
- **ys** – mole fractions
- **P** – pressure in Pa
- **T** – temperature in K

get_w_Tc_Pc(*i: int*, *j=None*)
Returns critical constants for calculation based off of whetner i = j or not

Returns (w, T_c, P_c)

Return type tuple

ln_hat_phi_k_expr(*k: int*, *ys: List[Union[float, Any]]*, *P*, *T*)
logarithm of fugacity coefficient

Note: The order of `ys` corresponds to the order of components made during initialization

Parameters

- **k** – index of component *k*
- **ys** – mole fractions ordered as component order
- **P** – pressure in Pa

- **T** – temperature in K

Returns Equation (4.14)

plot_residual_HSG(*P*, *T*, *ax=None*, *fig=None*) → Tuple[matplotlib.pyplot.figure, matplotlib.pyplot.subplot]

Plot dimensionless residual properties as a function of mole fraction

Parameters

- **P** – pressure in Pa
- **T** – Temperature in K
- **ax** – matplotlib ax, defaults to None
- **fig** – matplotlib figure, defautls to None

CHAPTER

FIVE

INPUT

Helper classes for inputting parameters

class realgas.input.**IdealMixture**(***kwargs*)

Parameters

- **num_components** (*int*) – number of components
- **dippr_nos** (*typing.Optional[typing.Union[str, None]]*) – dippr numbers of components
- **compound_names** (*typing.Optional[typing.Union[str, None]]*) – names of components
- **cas_numbers** (*typing.Optional[typing.Union[str, None]]*) – cas registry numbers

get_point_input(*i: int*) → dict

Parameters **i** – index of point

Returns keyword arguments for point input

set_point_input(*i: int*, ***kwargs*)

Parameters **i** – index of point

setup()

setup input parameters

class realgas.input.**RealMixture**(***kwargs*)

Parameters

- **MWs** (*typing.Optional[typing.List[float]]*) – molecular weights of component sin g/mol
- **P_cs** (*typing.Optional[typing.List[float]]*) – critical pressures of componets in Pa
- **T_cs** (*typing.Optional[typing.List[float]]*) – critical temperatures of pure components in K
- **V_cs** (*typing.Optional[typing.List[float]]*) – critical molar volumes of pure components in m**3/mol
- **ws** (*typing.Optional[typing.List[float]]*) – accentric factors of components
- **k_ij** (*typing.Union[float, typing.List[float]], defaults to 0*) – equation of state mixing rule in calculation of critical temperautre, see Equation (4.43). When $i = j$ and for chemical similar species, $k_{ij} = 0$. Otherwise, it is a small (usually)

positive number evaluated from minimal PVT data or, in the absence of data, set equal to zero.

get_point_input(*i: int*) → dict

Parameters **i** – index of point

Returns keyword arguments for point input

set_point_input(*i: int*, ***kwargs*)

Parameters **i** – index of point

CHAPTER

SIX

THERMODYNAMIC PROPERTY DATA

Raw data obtained from [RWO+07] and [GP07]

6.1 Heat Capacity

class realgas.cp.**CpIdealGas**(*dippr_no: str = None*, *compound_name: str = None*, *cas_number: str = None*, *T_min_fit: float = None*, *T_max_fit: float = None*, *n_points_fit: int = 1000*, *poly_order: int = 2*, *T_units='K'*, *Cp_units='J/mol/K'*)

Heat Capacity $C_{\mathrm{p}}^{\mathrm{IG}}$ [J/mol/K] at Constant Pressure of Inorganic and Organic Compounds in the Ideal Gas State Fit to Hyperbolic Functions [RWO+07]

$$C_{\mathrm{p}}^{\mathrm{IG}} = C_1 + C_2 \left[\frac{C_3/T}{\sinh\left(C_3/T\right)}\right] + C_4 \left[\frac{C_5/T}{\cosh\left(C_5/T\right)}\right]^2 \tag{6.1}$$

where $C_{\mathrm{p}}^{\mathrm{IG}}$ is in J/mol/K and T is in K.

Computing integrals of Equation (6.1) is challenging. Instead, the function is fit to a polynomial within a range of interest so that it can be integrated by using an antiderivative that is a polynomial.

Parameters

- **dippr_no** (*str, optional*) – dippr_no of compound by DIPPR table, defaults to None
- **compound_name** (*str, optional*) – name of chemical compound, defaults to None
- **cas_number** (*str, optional*) – CAS registry number for chemical compound, defaults to None
- **MW** (*float, derived from input*) – molecular weight in g/mol
- **T_min** (*float, derived from input*) – minimum temperature of validity for relationship [K]
- **T_max** (*float, derived from input*) – maximum temperature of validity [K]
- **T_min_fit** – minimum temperature for fitting, defaults to Tmin
- **T_max_fit** – maximum temperature for fitting, defaults to Tmax
- **C1** (*float, derived from input*) – parameter in Equation (6.1)
- **C2** (*float, derived from input*) – parameter in Equation (6.1)
- **C3** (*float, derived from input*) – parameter in Equation (6.1)
- **C4** (*float, derived from input*) – parameter in Equation (6.1)

- **C5** (*float, derived from input*) – parameter in Equation (6.1)
- **Cp_units** (*str, optional*) – units for C_p^{IG}, defaults to J/mol/K (SI units)
- **T_units** (*str, optional*) – units for T, defaults to K
- **n_points_fit** (*int, optional*) – number of points for fitting polynomial and plotting, defaults to 1000
- **poly_order** (*int, optional*) – order of polynomial for fitting, defaults to 2

cp_integral(*T_a*, *T_b*)
Evaluate integral

$$\int_{T_a}^{T_b} C_p^{IG}(T')dT' \tag{6.2}$$

Parameters

- **T_a** – start temperature in K
- **T_b** – finish temperature in K

Returns integral

eval(*T*, *f_sinh=<ufunc 'sinh'>*, *f_cosh=<ufunc 'cosh'>*)
Evaluate heat capacity

Parameters

- **T** – temperature in K
- **f_sinh** (*callable*) – function for hyperbolic sine, defaults to `np.sinh`
- **f_cosh** (*callable*) – function for hyperbolic cosine, defaults to `np.cosh`

Returns C_p^{IG} J/mol/K (see equation (6.1))

get_numerical_percent_difference()
Calculate the percent difference with numerical integration

numerical_integration(*T_a*, *T_b*) → tuple
Numerical integration using scipy

class realgas.cp.**CpStar**(*T_ref: float*, ***kwargs*)
Dimensionless Heat Capacity at Constant Pressure of Inorganic and Organic Compounds in the Ideal Gas State Fit to Hyperbolic Functions [RWO+07]

The dimensionless form is obtained by introducing the following variables

$$C_p^\star = \frac{C_p^{IG}}{R} \tag{6.3}$$

$$T^\star = \frac{T}{T_{ref}} \tag{6.4}$$

where R is the gas constant in units of J/mol/K, and T_{ref} is a reference temperature [K] input by the user (see `T_ref`)

The heat capacity in dimensionless form becomes

$$C_p^\star = C_1^\star + C_2^\star \left[\frac{C_3^\star/T^\star}{\sinh\left(C_3^\star/T^\star\right)}\right] + C_4^\star \left[\frac{C_5^\star/T^\star}{\cosh\left(C_5^\star/T^\star\right)}\right]^2 \tag{6.5}$$

where

$$C_1^\star = \frac{C_1}{\mathrm{R}} \tag{6.6}$$

$$C_2^\star = \frac{C_2}{\mathrm{R}} \tag{6.7}$$

$$C_3^\star = \frac{C_3}{T_{\mathrm{ref}}} \tag{6.8}$$

$$C_4^\star = \frac{C_4}{\mathrm{R}} \tag{6.9}$$

$$C_5^\star = \frac{C_5}{T_{\mathrm{ref}}} \tag{6.10}$$

Parameters **T_ref** (*float*) – reference temperature [K] for dimensionless computations

eval(*T*, *f_sinh=<ufunc 'sinh'>*, *f_cosh=<ufunc 'cosh'>*)

Parameters

- **T** – temperature in K
- **f_sinh** (*callable*) – function for hyperbolic sine, defaults to `np.sinh`
- **f_cosh** (*callable*) – function for hyperbolic cosine, defaults to `np.cosh`

Returns $C_{\mathrm{p}}^\star$ [dimensionless] (see equation (6.5))

class realgas.cp.**CpRawData**(*T_raw: list*, *Cp_raw: list*, *T_min_fit: float = None*, *T_max_fit: float = None*, *poly_order: int = 2*, *T_units='K'*, *Cp_units='J/mol/K'*)

Obtain heat capacity relationships from raw data

Using input raw data # fit to polynomial of temperature # fit polynomial to antiderivative

Parameters

- **T_min_fit** (*float, optional*) – minimum temperature for fitting function [K]
- **T_max_fit** (*float, optional*) – maximum temperature for fitting function [K]
- **poly_order** (*int, optional*) – order of polynomial for fitting, defaults to 2
- **T_raw** (*list*) – raw temperatures in K
- **Cp_raw** (*list*) – raw heat capacities in J/K/mol
- **Cp_units** (*str, optional*) – units for C_{p}, defaults to J/mol/K
- **T_units** (*str, optional*) – units for T, defaults to K

get_max_percent_difference()

Get largest percent difference

6.2 Critical Properties

class realgas.critical_constants.**CriticalConstants**(*dippr_no: str = None*, *compound_name: str = None*, *cas_number: str = None*, *MW: float = None*, *P_c: float = None*, *V_c: float = None*, *Z_c: float = None*, *T_c: float = None*, *w: float = None*)

Get critical constants of a compound

If critical constants are not passed in, reads from DIPPR table

Parameters

- **dippr_no** (*str, optional*) – dippr_no of compound by DIPPR table, defaults to None
- **compound_name** (*str, optional*) – name of chemical compound, defaults to None
- **cas_number** (*str, optional*) – CAS registry number for chemical compound, defaults to None
- **MW** – molecular weight in g/mol
- **T_c** – critical temperature [K]
- **P_c** – critical pressure [Pa]
- **V_c** – critical molar volume [m^3/mol]
- **Z_c** – critical compressibility factor [dimensionless]
- **w** – accentric factor [dimensionless]
- **tol** (*float, hard-coded*) – tolerance for percent difference in Zc calulcated and tabulated, set to 0.5

Z_c_percent_difference()
calculate percent difference between Z_c calculated and tabulated

calc_Z_c()
Calculate critical compressibility, for comparison to tabulated value

6.3 Thermal Conductivity

class realgas.thermal_conductivity.**ThermalConductivity**(*dippr_no: str = None*, *compound_name: str = None*, *cas_number: str = None*, *T_min_fit: float = None*, *T_max_fit: float = None*, *n_points_fit: int = 1000*, *poly_order: int = 2*)

Thermal Conductivity of Inorganic and Organic Substances [W/m/K] [RWO+07]

$$k = \frac{C_1 T^{C_2}}{1 + C_3/T + C_4/T^2} \tag{6.11}$$

where k is the thermal conductivity in W/m/K and T is in K. Thermal conductivites are either at 1 atm or the vapor pressure, whichever is lower.

Parameters

- **dippr_no** (*str, optional*) – dippr_no of compound by DIPPR table, defaults to None
- **compound_name** (*str, optional*) – name of chemical compound, defaults to None
- **cas_number** (*str, optional*) – CAS registry number for chemical compound, defaults to None
- **MW** (*float, derived from input*) – molecular weight in g/mol

- **T_min** (*float, derived from input*) – minimum temperature of validity for relationship [K]
- **T_max** (*float, derived from input*) – maximum temperature of validity [K]
- **C1** (*float, derived from input*) – parameter in Equation (6.11)
- **C2** (*float, derived from input*) – parameter in Equation (6.11)
- **C3** (*float, derived from input*) – parameter in Equation (6.11)
- **C4** (*float, derived from input*) – parameter in Equation (6.11)
- **units** (*str*) – units for k, set to W/m/K
- **T_min_fit** – minimum temperature for fit, defaults to T_min
- **T_max_fit** – maximum temperature for fit, defaults to T_max

eval(*T*)

Parameters **T** – temperature in K

Returns k W/m/K (see equation (6.11))

class realgas.thermal_conductivity.**ThermalConductivityMixture**(*name_to_cas: dict, mixing_rule='Simple'*)

Viscosity of vapor mixture using Wilke mixing rule

Parameters

- **name_to_cas** (dict[components, str]) – mapping of chemical name to cas registry number
- **mixing_rule** (*str, optional*) – mixing rule for calculation of viscosity, defaults to Simple
- **pure** (dict[components, Viscosity]) – pure component viscosity info, obtained rom realgas.vapor_viscosity.Viscosity

eval_HR(*y_i*, *T*)

Weights based off of sqrt of molecular weights

eval_simple(*y_i*, *T*)

Calculate thermal conductivity using simple relationship

Parameters **y_i** (dict[component, float]) – mole fraction of each component i

6.4 Viscosity

class realgas.viscosity.**Viscosity**(*dippr_no: str = None, compound_name: str = None, cas_number: str = None, T_min_fit: float = None, T_max_fit: float = None, n_points_fit: int = 1000, poly_order: int = 2*)

Vapor Viscosity of Inorganic and Organic Substances [W/m/K] [RWO+07]

$$\mu = \frac{C_1 T^{C_2}}{1 + C_3/T + C_4/T^2} \tag{6.12}$$

where μ is the thermal conductivity in W/m/K and T is in K. Viscosities are either at 1 atm or the vapor pressure, whichever is lower.

Parameters

- **dippr_no** (*str, optional*) – dippr_no of compound by DIPPR table, defaults to None
- **compound_name** (*str, optional*) – name of chemical compound, defaults to None
- **cas_number** (*str, optional*) – CAS registry number for chemical compound, defaults to None
- **MW** (*float, derived from input*) – molecular weight in g/mol
- **T_min** (*float, derived from input*) – minimum temperature of validity for relationship [K]
- **T_max** (*float, derived from input*) – maximum temperature of validity [K]
- **C1** (*float, derived from input*) – parameter in Equation (6.12)
- **C2** (*float, derived from input*) – parameter in Equation (6.12)
- **C3** (*float, derived from input*) – parameter in Equation (6.12)
- **C4** (*float, derived from input*) – parameter in Equation (6.12)
- **units** (*str*) – units for μ, set to Pa*s
- **T_min_fit** – minimum temperature for fit, defaults to T_min
- **T_max_fit** – maximum temperature for fit, defaults to T_max

eval(*T*)

Parameters **T** – temperature in K

Returns μ Pa*s (see equation (6.12))

class realgas.viscosity.**ViscosityMixture**(*name_to_cas: dict = None, mixing_rule='Herning Zipperer', **kwargs*)

Viscosity of vapor mixture using Wilke or HR mixing rule

Parameters

- **name_to_cas** (dict[components, str]) – mapping of chemical name to cas registry number
- **mixing_rule** (*str, optional*) – mixing rule for calculation of viscosity, defaults to Herning Zipperer
- **pure** (dict[components, Viscosity]) – pure component viscosity info, obtained rom *Viscosity*

eval_Wilke(*y_i*, *T*)

Calculate mixture viscosity in Pa*s using Wilke mixing rule

Parameters **y_i** (dict[component, float]) – mole fraction of each component i

phi_ij(*i: str*, *j: str*, *T: float*)

Coefficient for each pair of components in a mixtures

Parameters

- **i** – name of component i
- **j** – name of component j

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Dei02] U K Deiters. Calculation of Densities from Cubic Equations of State. *AIChE J.*, 48:882–886, 2002.

[GP07] D W Green and R H Perry. *Perry's Chemical Engineers' Handbook*. McGraw-Hill Professional Publishing, 8th edition, 2007.

[ML12] R Monroy-Loperena. A Note on the Analytical Solution of Cubic Equations of State in Process Simulation. *Ind. Eng. Chem. Res.*, 51:6972–6976, 2012. doi:10.1021/ie2023004.

[PR76] D-Y Peng and D B Robinson. A New Two-Constant Equation of State. *Ind. Eng. Chem., Fundam.*, 15:59–64, 1976.

[PLdeAzevedo86] J Prausnitz, R N Lichtenthaler, and E G de Azevedo. *Molecular Thermodynamics of Fluid-Phase Equilibria*, pages 132,162. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1986.

[RK49] O Redlich and J N S Kwong. On the Thermodynamics of Solutions. V: An Equation of State. Fugacities of Gaseous Solutions. *Chem. Rev.*, 44:233–244, 1949.

[RWO+07] R L Rowley, W V Wilding, J L Oscarson, Y Yang, N A Zundel, T E Daubert, and R P Danner. DIPPR$^\mathrm ®$ Data Compilation of Pure Chemical Properties, Design Institute for Physical Properties. In *Design Institute for Physical Properties of the American Institute of Chemical Engineers*. AIChE, New York, 2007.

[SVanNessA05] J M Smith, H C Van Ness, and M M Abbott. *Introduction to Chemical Engineering Thermodynamics*. McGraw-Hill, 7th edition, 2005.

[Soa72] G Soave. Equilibrium Constants from a Modified Redlich-Kwong Equation of State. *Chem. Eng. Sci.*, 27:1197–1203, 1972.

[VanNessA82] H C Van Ness and M M Abbott. *Classical Thermodynamics of Nonelectrolyte Solutions: With Applications to Phase Equilibria*. McGraw-Hill, New York, 1982.

PYTHON MODULE INDEX

r

INDEX

E

F

G

H

I

L

M

N

P