

---

# **gasthermo**

***Release 0.0.6***

**Robert F. De Jaco**

**May 23, 2020**

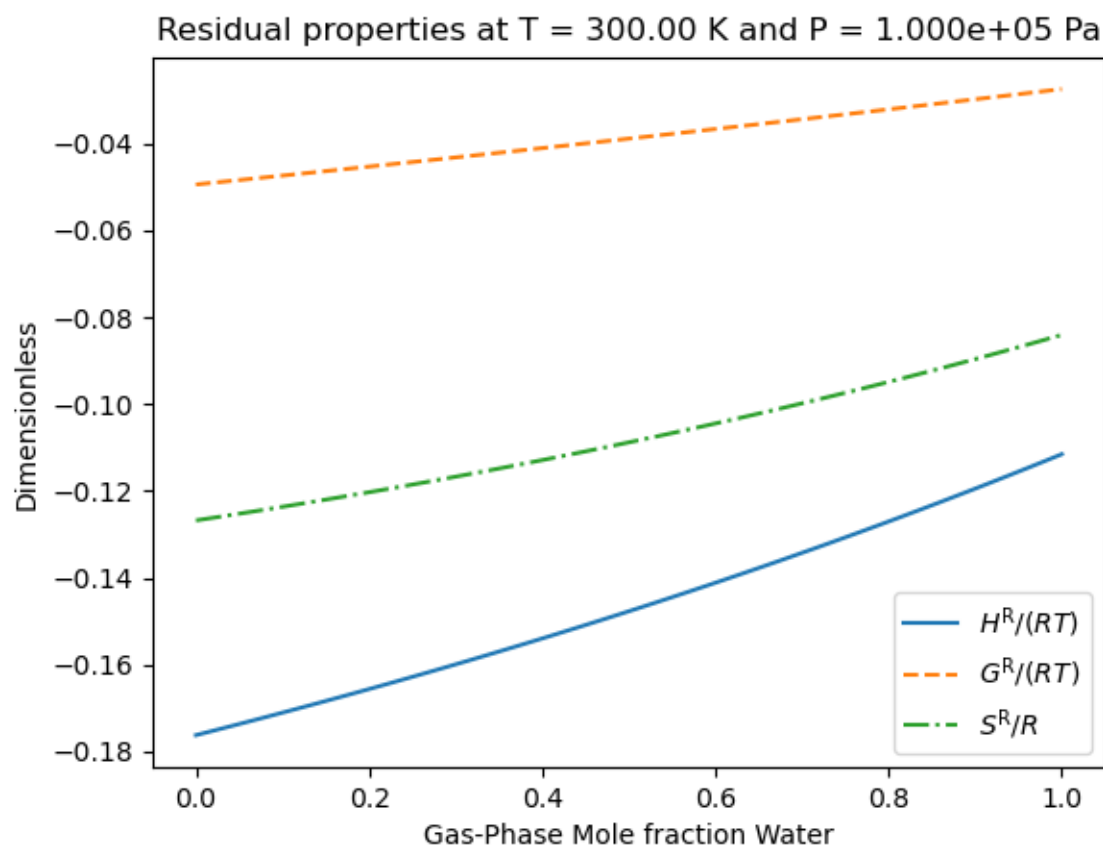


## CONTENTS:

<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Single-Component Examples . . . . .	3
1.3	Mixture Examples . . . . .	8
1.4	Gotchas . . . . .	11
<b>2</b>	<b>Definitions</b>	<b>13</b>
2.1	Nomenclature . . . . .	13
<b>3</b>	<b>Partial Molar Properties</b>	<b>15</b>
3.1	Ideal Gas . . . . .	15
3.2	Residual . . . . .	16
<b>4</b>	<b>Equations of State</b>	<b>19</b>
4.1	Single Component . . . . .	19
4.2	Multicomponent . . . . .	30
<b>5</b>	<b>Input</b>	<b>37</b>
<b>6</b>	<b>Thermodynamic Property Data</b>	<b>39</b>
6.1	Heat Capacity . . . . .	39
6.2	Critical Properties . . . . .	41
6.3	Thermal Conductivity . . . . .	42
6.4	Viscosity . . . . .	43
<b>7</b>	<b>References</b>	<b>45</b>
<b>8</b>	<b>Indices and tables</b>	<b>47</b>
	<b>Bibliography</b>	<b>49</b>
	<b>Python Module Index</b>	<b>51</b>
	<b>Index</b>	<b>53</b>



A package for obtaining gas-phase thermodynamic properties and computing integrals. Designed for integration with computational simulation environments.





## GETTING STARTED

### 1.1 Installation

The package can be installed via `pip` as follows

```
pip install GasThermo
```

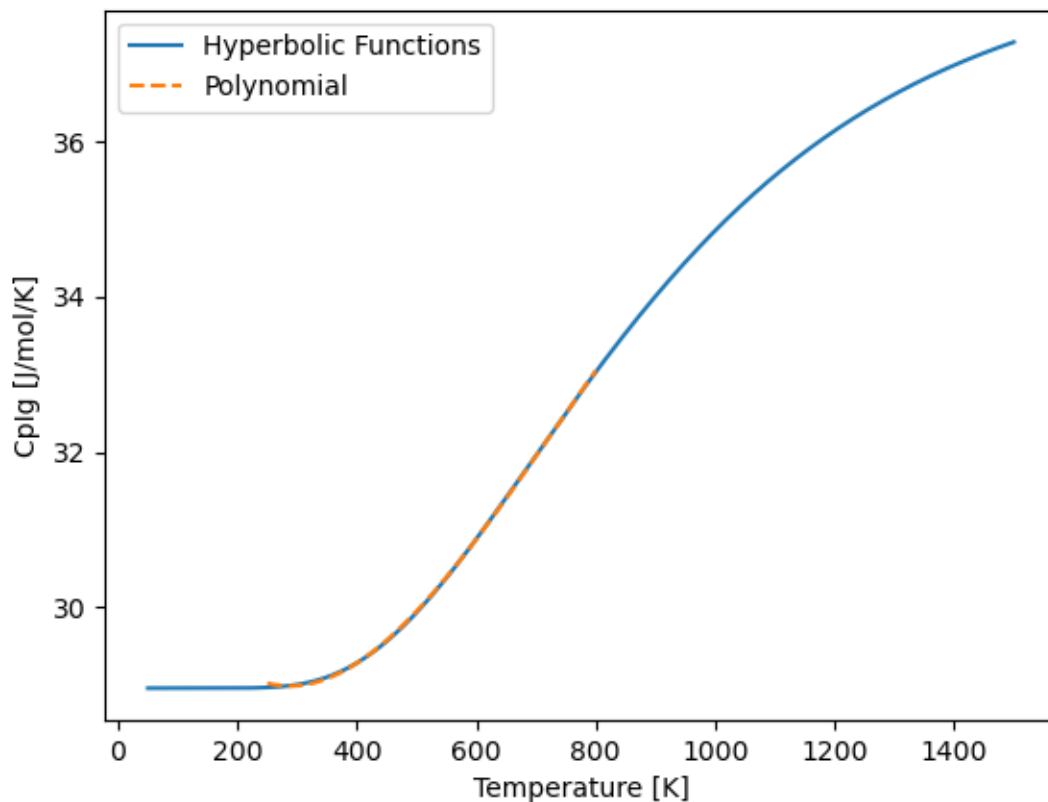
### 1.2 Single-Component Examples

#### 1.2.1 Heat Capacity

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

```
>>> import matplotlib.pyplot as plt
>>> from gasthermo.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 results
>>> fig, ax = I.plot()
>>> fig.savefig('docs/source/air.png')
>>> del I
```

And we will get something that looks like the following



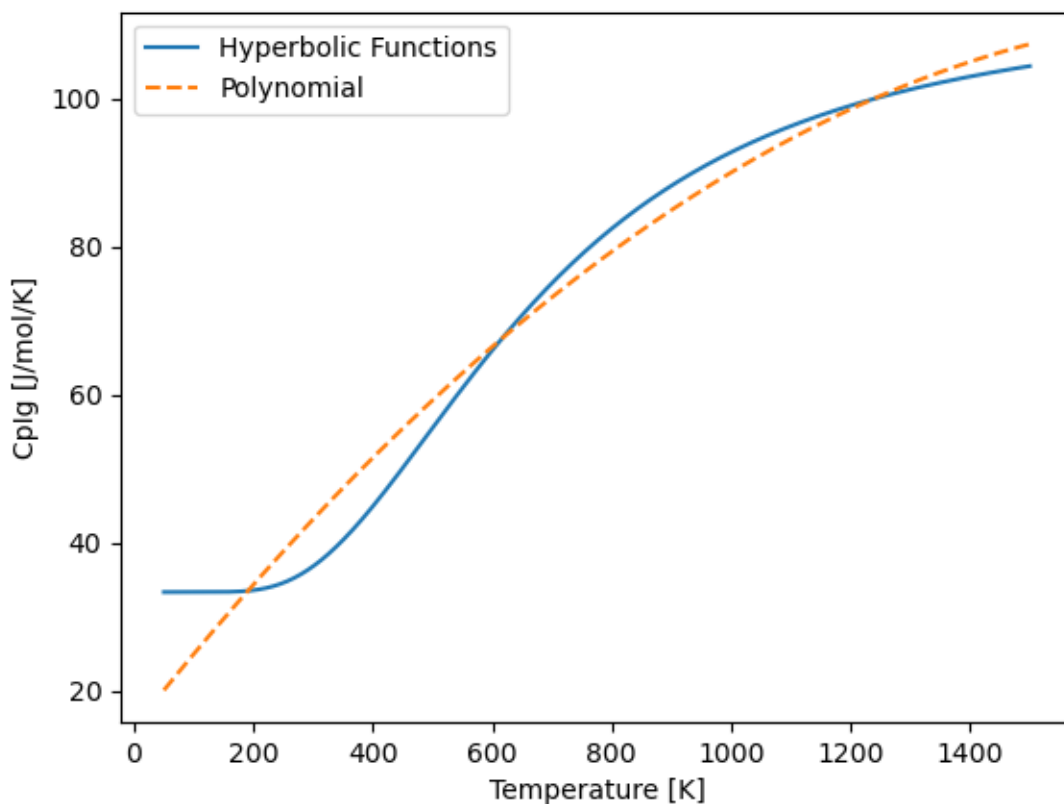
and we notice that the polynomial (orange dashed lines) fits the hyperbolic function well.

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

```
>>> from gasthermo.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:





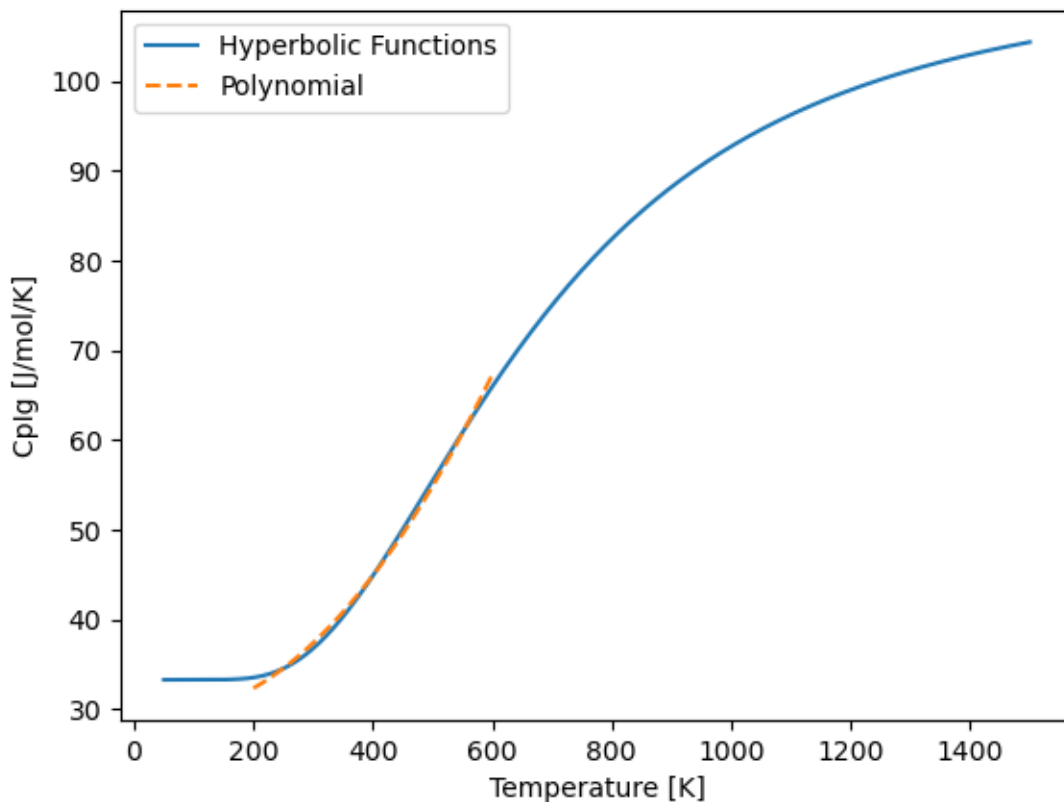
Usually, we won't need an accurate function over this entire temperature range. Let's imagine that we are interested instead in a temperature interval between 200 and 600 K. In this case

```
>>> 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 gasthermo.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
```

### 1.2.3 Virial Equation of State

```
>>> from gasthermo.eos.virial import SecondVirial
>>> 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 gasthermo.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 gasthermo.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
>>> custom.iterate_to_solve_Z(T=300., P=8e5)
0.9823877
```

If we accidentally input the wrong custom units, it is likely that *gasthermo.critical\_constants.CriticalConstants* will catch it.

```
>>> from gasthermo.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',
```

(continues on next page)

(continued from previous page)

```
... 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

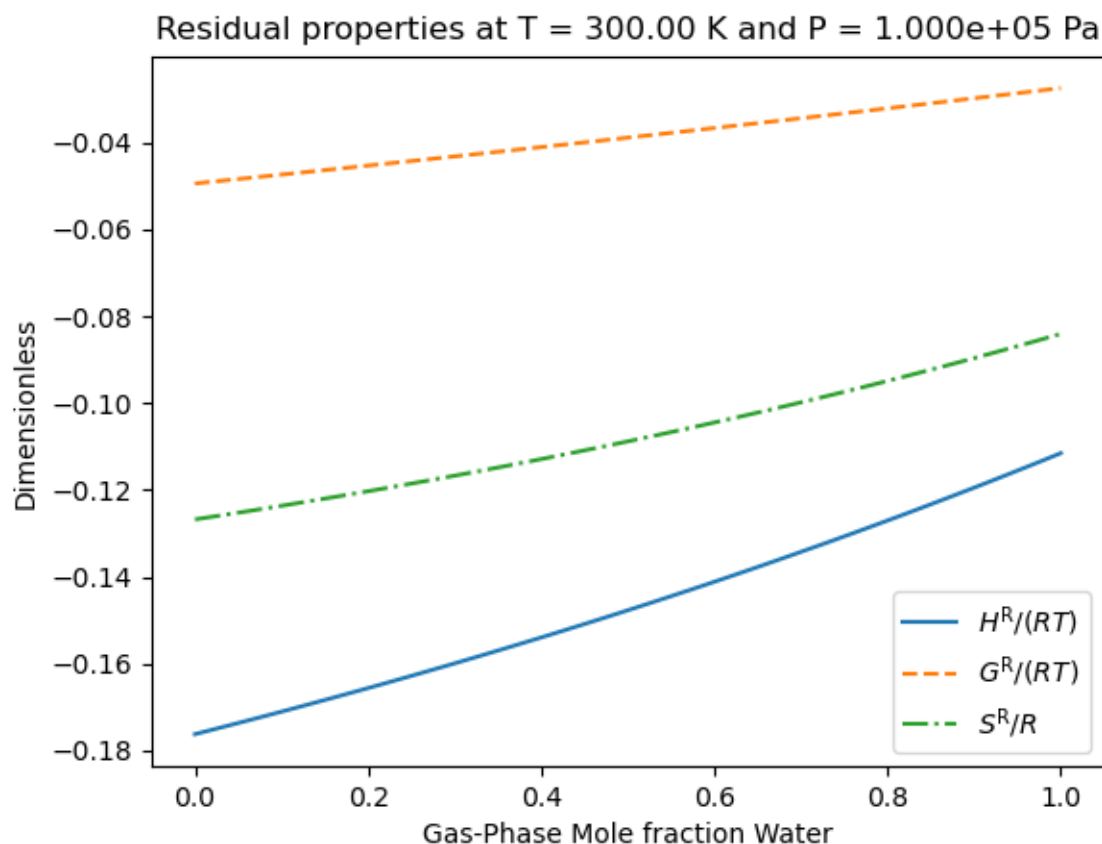
---

### 1.3.1 Residual Properties

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

```
>>> from gasthermo.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 gasthermo.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  $P = 0$  and  $T = T_{\text{ref}}$ . The reference temperature is  $T_{\text{ref}}$  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.)
```

(continues on next page)

(continued from previous page)

```

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)],

```

(continues on next page)

(continued from previous page)

```

...     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





## DEFINITIONS

Residual properties for a given thermodynamic property  $M$  are defined as

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

where  $M^{\text{IG}}$  is the value of the property in the ideal gas state and  $M^{\text{R}}$  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 $\text{m}^3/\text{mol}$
R	$R$	gas constant SI units ( $\text{m}^3 \times \text{Pa}/\text{mol}/\text{K}$ )
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 $\text{m}^3/\text{mol}$
w	$\omega$	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



## PARTIAL MOLAR PROPERTIES

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

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

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

$$\bar{M}_i = \bar{M}^{\text{IG}} + \bar{M}^{\text{R}} \quad (3.2)$$

$$M^{\text{R}} = \sum_i y_i \bar{M}^{\text{R}} \quad (3.3)$$

### 3.1 Ideal Gas

The *Gibbs theorem* is

A partial molar property (other than volume) of a constituent 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,  $\bar{V}_i^{\text{IG}}$ , is

$$\bar{V}_i^{\text{IG}} = \frac{RT}{P} \quad (3.4)$$

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

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

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

$$\bar{H}_i^{\text{IG}} = H_i^{\text{IG}} = \int_{T_{\text{ref}}}^T C_{p,i}^{\text{IG}} dT' \quad (3.5)$$

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

$$\begin{aligned}\bar{H}_i^{\text{IG},*} &= \frac{\bar{H}_i^{\text{IG}}}{RT_{\text{ref}}} \\ &= \int_1^{T^*} \frac{C_{p,i}^{\text{IG}}}{R} dT' \\ (3.8)\end{aligned}\tag{3.6}$$

or

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

where

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

is a dimensionless variable and

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

is a dimensionless parameter that is a function of  $T^*$ .

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

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

The expression in dimensionless units can be simplified to

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

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

$$\bar{H}_i^{R,*} = \frac{\bar{H}_i^R}{RT_{\text{ref}}} \quad (3.13)$$

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

$$= -\frac{RT_{\text{ref}}^2}{RT_{\text{ref}}^2} \left( \frac{\partial \ln \hat{\phi}_i}{\partial T^*} \right) \quad (3.15)$$

(3.16)

The expression in dimensionless units is computed as

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

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

$$\bar{G}_i^R = RT \ln \hat{\phi}_i \quad (3.18)$$

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** gasthermo.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 `gasthermo.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** – temperature in K

**Returns** *arH<sub>i</sub><sup>e</sup>xtIG*, see Equation (3.5)

**bar\_Vi\_IG** (*T, P*)

#### Parameters

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

**Returns** *arV<sub>i</sub><sup>e</sup>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 gasthermo.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 `gasthermo.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**  $arH_i^e at IG$ , 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

## EQUATIONS OF STATE

### 4.1 Single Component

#### 4.1.1 Virial

---

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

---

#### Theory

The second order virial equation of state is [GP07]

$$Z = 1 + B\rho = 1 + \frac{BP}{RT} \quad (4.1)$$

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

$$B = \sum_i \sum_j y_i y_j B_{ij} \quad (4.2)$$

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

In this package, the useful correlation

$$\frac{BP_c}{RT_c} = B^0 + \omega B^1$$

or

$$B = \frac{RT_c}{P_c} (B^0 + \omega B^1) \quad (4.3)$$

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

$$Z = 1 + (B^0 + \omega B^1) \frac{P_r}{T_r} \quad (4.4)$$

where [SVanNessA05]

$$B^0 = 0.083 - \frac{0.422}{T_r^{1.6}} \quad (4.5)$$

$$B^1 = 0.139 - \frac{0.172}{T_r^{4.2}} \quad (4.6)$$

so that

so that the following derivatives can be computed as

$$\frac{dB^0}{dT_r} = \frac{0.675}{T_r^{2.6}} \quad (4.7)$$

$$\frac{dB^1}{dT_r} = \frac{0.722}{T_r^{5.2}} \quad (4.8)$$

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

$$\frac{G^R}{RT} = (B^0 + \omega B^1) \frac{P_r}{T_r} \quad (4.9)$$

$$\frac{H^R}{RT} = P_r \left[ \frac{B^0}{T_r} - \frac{dB^0}{dT_r} + \omega \left( \frac{B^1}{T_r} - \frac{dB^1}{dT_r} \right) \right] \quad (4.10)$$

$$\frac{S^R}{R} = -P_r \left( \frac{dB^0}{dT_r} - \omega \frac{dB^1}{dT_r} \right) \quad (4.11)$$

The cross coefficients are calculated as

$$B_{ij} = \frac{RT_{cij}}{P_{cij}} (B^0 + \omega_{ij} B^1) \quad (4.12)$$

so that the cross derivatives can be computed as

$$\begin{aligned} \frac{dB_{ij}}{dT} &= \frac{RT_{cij}}{P_{cij}} \left( \frac{dB^0}{dT} + \omega_{ij} \frac{dB^1}{dT} \right) \\ \frac{dB_{ij}}{dT_{rij}} &= \frac{R}{P_{cij}} \left( \frac{dB^0}{dT_{rij}} + \omega_{ij} \frac{dB^1}{dT_{rij}} \right) \end{aligned}$$

or, equivalently,

$$\frac{dB_{ij}}{dT_{rij}} = \frac{RT_{cij}}{P_{cij}} \left( \frac{dB^0}{dT_{rij}} + \omega_{ij} \frac{dB^1}{dT_{rij}} \right) \quad (4.13)$$

## Fugacity Coefficients

The Fugacity coefficients are calculated as

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

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

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

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

$$\delta_{ik} = 2B_{ik} - B_{ii} - B_{kk} = \delta_{ki} \quad (4.15)$$

and

$$\delta_{ii} = 0$$



## Residual Partial Molar Properties

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

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

The partial molar residual enthalpy of component  $k$  is

$$\begin{aligned} \frac{\bar{H}_k^R}{RT} &= -T \left( \frac{\partial \ln \hat{\phi}_k}{\partial T} \right)_{P,y} \\ &= -\frac{T}{T_c} \left( \frac{\partial \ln \hat{\phi}_k}{\partial T_r} \right)_{P,y} \quad (4.17) \\ &= -\frac{T}{T_c} \left\{ \frac{P}{RT} \left[ \frac{\partial B_{kk}}{\partial T_r} + \frac{1}{2} \sum_i \sum_j y_i y_j \left( 2 \frac{\partial \delta_{ik}}{\partial T_r} - \frac{\partial \delta_{ij}}{\partial T_r} \right) \right] - \frac{T_c \ln \hat{\phi}_k}{T} \right\} \quad (4.18) \end{aligned}$$

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

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

Since

$$G^R = H^R - TS^R$$

In terms of partial molar properties, then

$$\begin{aligned} \bar{S}_i^R &= \frac{\bar{H}_i^R - \bar{G}_i^R}{T} \quad (4.21) \\ \frac{\bar{S}_i^R}{R} &= \frac{\bar{H}_i^R}{RT} - \frac{\bar{G}_i^R}{RT} \quad (4.22) \end{aligned}$$

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

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

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

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

$$\begin{aligned} \frac{\bar{V}_k^R}{RT} &= \left( \frac{\partial \ln \hat{\phi}_k}{\partial P} \right)_{T,y} \\ &= \frac{\partial}{\partial P} \left\{ \frac{P}{RT} \left[ B_{kk} + \frac{1}{2} \sum_i \sum_j y_i y_j (2\delta_{ik} - \delta_{ij}) \right] \right\} \quad (4.25) \end{aligned}$$

which simplifies to

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

from which we obtain the intuitive result of

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

```
class gasthermo.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:  $\exp(\ln \hat{\phi}_i)$

```
class gasthermo.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** – Temperature 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  $\epsilon$  and  $\sigma$  are pure numbers (the same for all substances), and  $a(T)$  and  $b$  are given by the following equations

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

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

The compressibility factor can be calculated by solving the following equation

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

where

$$\beta = \Omega \frac{P_r}{T_r} \quad (4.30)$$

and

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

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

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

that is continued until the following is true

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

### Residual Molar Properties

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

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

$$\frac{S^R}{R} = \ln(Z - \beta) + \frac{d \ln \alpha(T_r)}{d \ln T_r} qI \quad (4.36)$$

where the following functions have been defined

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

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

## Fugacity Coefficients

The pure-component fugacity coefficient is *defined* as

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

So that, from Equation (4.35),

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

## Mixtures

**Warning:** Not implemented yet!

---

**Todo:** Implement mixtures with cubic equations of state

---

```
class gasthermo.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,  $\sigma$
- **epsilon** – Parameter defined by specific equation of state,  $\epsilon$
- **Omega** – Parameter defined by specific equation of state,  $\Omega$
- **Psi** – Parameter defined by specific equation of state,  $\Psi$
- **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  $\text{m}^3/\text{mol}$

- **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** – temperature 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*/*T<sub>c</sub>*), dimensionless

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

**beta\_expr** (*T*, *P*)

**Parameters**

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

**Returns**  $\beta$  (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 coefficients 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  $\frac{d \ln \alpha(T_r)}{d \ln T_r}$

**hat\_phi\_i\_expr** (\*args)

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

**iterate\_to\_solve\_Z** ( $T, P$ ) → float

Iterate to compute  $Z$  using Equation (4.32) until 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<sup>3</sup>/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** – temperature 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** gasthermo.eos.cubic.**RedlichKwong** (**\*\*kwargs**)

Redlich-Kwong Equation of State [RK49]

This Equation of state has the following parameters [SVanNessA05]

Symbol	Value
$\alpha(T_r)$	$1/\sqrt{T_r}$
$\sigma$	1
$\epsilon$	0
$\Omega$	0.08664
$\Psi$	0.42748

```
>>> from gasthermo.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 ( $T/T_c$ ), dimensionless

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



**d\_ln\_alpha\_d\_ln\_Tr** ( $T_r$ )

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

**Returns** Expression for  $\frac{d \ln \alpha(T_r)}{d \ln T_r}$

**class** gasthermo.eos.cubic.**SoaveRedlichKwong** (*\*\*kwargs*)  
Soave Redlich-Kwong Equation of State [Soa72]

This equation of state has the following parameters

Symbol	Value
$\alpha(T_r)$	$[1 + f_w(1 - \sqrt{T_r})]^2$
$\sigma$	1
$\epsilon$	0
$\Omega$	0.08664
$\Psi$	0.42748

where

$$f_w = 0.480 + 1.574\omega - 0.176\omega^2 \quad (4.40)$$

```
>>> from gasthermo.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 ( $T/T_c$ ), dimensionless

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

**d\_ln\_alpha\_d\_ln\_Tr** ( $T_r$ )

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

**Returns** Expression for  $\frac{d \ln \alpha(T_r)}{d \ln T_r}$

**f\_w\_rule** ( $w$ )

**Parameters**  $w$  – accentric factor

**Returns**  $f_w$ , see Equation (4.40)

**class** gasthermo.eos.cubic.**PengRobinson** (\*\*kwargs)

Peng-Robinson Equation of State [PR76]

This equation of state has the following parameters

Symbol	Value
$\alpha(T_r)$	$[1 + f_w(1 - \sqrt{T_r})]^2$
$\sigma$	$1 + \sqrt{2}$
$\epsilon$	$1 - \sqrt{2}$
$\Omega$	0.07780
$\Psi$	0.45724

where

$$f_w = 0.480 + 1.574\omega - 0.176\omega^2 \quad (4.41)$$

```
>>> from gasthermo.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** – eccentric factor

**Returns**  $f_w$ , see Equation (4.41)

## 4.2 Multicomponent

### 4.2.1 Virial

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

Van der Walls mixing rule

combining rules from [PLdeAzevedo86]

$$\omega_{ij} = \frac{\omega_i + \omega_j}{2} \quad (4.42)$$

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

$$P_{cij} = \frac{Z_{cij}RT_{cij}}{V_{cij}} \quad (4.44)$$

$$Z_{cij} = \frac{Z_{ci} + Z_{cj}}{2} \quad (4.45)$$

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

**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** – acentric factor of component *i*
- **w\_j** – acentric factor of component *j*

**Returns** Equation (4.42)

**class** gasthermo.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, defaults 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  $G^R/R/T$

**H\_R\_RT** (\*args)

Residual enthalpy of mixture  $H^R/R/T$

**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  $\bar{H}_i^{R,*}$ , 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 component  $j$
- **T** – temperature in K

**Returns** Equation (4.13)

**d\_dij\_d\_Tr** (*i*, *j*, *T*)

$$\frac{\partial \delta_{ij}}{\partial T_{rij}} = 2 \frac{\partial B_{ij}}{\partial T_{rij}} - \frac{\partial B_{ii}}{\partial T_{rij}} - \frac{\partial B_{jj}}{\partial T_{rij}} \quad (4.47)$$

---

**Todo:** test this with symbolic differentiation of d\_ik expression

---

**Parameters**

- **i** – index for component *i*
- **j** – index for component *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 whether *i* = *j* or not

**Returns** (*w*, *T<sub>c</sub>*, *P<sub>c</sub>*)

**Return type** tuple

**ln\_hat\_phi\_k\_expr** (*k*: int, *ys*: List[Union[float, Any]], *P*, *T*)

logarithm of fugacity coefficient

**Parameters**

- **k** – index of component *k*
- **ys** – mole fractions
- **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, defaults to None





## INPUT

Helper classes for inputting parameters

```
class gasthermo.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 gasthermo.input.RealMixture(**kwargs)
```

**Parameters**

- **MWs** (*typing.Optional[typing.List[float]]*) – molecular weights of component in g/mol
- **P\_cs** (*typing.Optional[typing.List[float]]*) – critical pressures of components 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<sup>3</sup>/mol
- **ws** (*typing.Optional[typing.List[float]]*) – acentric factors of components
- **k\_ij** (*typing.Union[float, typing.List[float]]*, defaults to 0) – equation of state mixing rule in calculation of critical temperature, 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

## THERMODYNAMIC PROPERTY DATA

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

## 6.1 Heat Capacity

```
class gasthermo.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  $C_p^{\text{IG}}$  [J/mol/K] at Constant Pressure of Inorganic and Organic Compounds in the Ideal Gas State  
Fit to Hyperbolic Functions [RWO+07]

$$C_p^{\text{IG}} = C_1 + C_2 \left[ \frac{C_3/T}{\sinh(C_3/T)} \right] + C_4 \left[ \frac{C_5/T}{\cosh(C_5/T)} \right]^2 \quad (6.1)$$

where  $C_p^{\text{IG}}$  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  $C_p^{\text{IG}}$ , 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

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

#### Parameters

- **T\_a** – start temperature in K
- **T\_b** – finish temperature in K

**Returns** integral

**eval** ( $T, f_{\text{sinh}}=<\text{ufunc 'sinh'}>, f_{\text{cosh}}=<\text{ufunc 'cosh'}>$ )

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**  $C_p^{\text{IG}}$  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** `gasthermo.cp.CpStar` ( $T_{\text{ref}}$ : 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

$$C_p^* = \frac{C_p^{\text{IG}}}{R} \quad (6.3)$$

$$T^* = \frac{T}{T_{\text{ref}}} \quad (6.4)$$

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

The heat capacity in dimensionless form becomes

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

where

$$\begin{aligned} C_1^* &= \frac{C_1}{R} \\ C_2^* &= \frac{C_2}{R} \\ C_3^* &= \frac{C_3}{T_{\text{ref}}} \\ C_4^* &= \frac{C_4}{R} \\ C_5^* &= \frac{C_5}{T_{\text{ref}}} \end{aligned} \quad (6.6)$$

**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**  $C_p^*$  [dimensionless] (see equation (6.5))

**class** `gasthermo.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** `gasthermo.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<sup>3</sup>/mol]
- **Z\_c** – critical compressibility factor [dimensionless]
- **w** – accentric factor [dimensionless]
- **tol** (*float, hard-coded*) – tolerance for percent difference in Z<sub>c</sub> calculated and tabulated, set to 0.5

**Z\_c\_percent\_difference** ()

calculate percent difference between Z<sub>c</sub> calculated and tabulated

**calc\_Z\_c** ()

Calculate critical compressibility, for comparison to tabulated value

## 6.3 Thermal Conductivity

```
class gasthermo.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]

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

where  $k$  is the thermal conductivity in W/m/K and  $T$  is in K. Thermal conductivities 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 gasthermo.thermal_conductivity.ThermalConductivityMixture (name_to_cas:
                                                                    dict,      mix-
                                                                    ing_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 from gasthermo.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 gasthermo.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]

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

where  $\mu$  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  $\mu$ , 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**  $\mu$  Pa\*s (see equation (6.12))

```
class gasthermo.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 from *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*



**REFERENCES**



## 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<sup>®</sup> 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

### g

- `gasthermo.cp`, [39](#)
- `gasthermo.critical_constants`, [41](#)
- `gasthermo.eos.cubic`, [24](#)
- `gasthermo.eos.virial`, [19](#)
- `gasthermo.input`, [35](#)
- `gasthermo.partial_molar_properties`, [13](#)
- `gasthermo.thermal_conductivity`, [42](#)
- `gasthermo.viscosity`, [43](#)





## A

`a_expr()` (*gasthermo.eos.cubic.Cubic* method), 26  
`alpha_expr()` (*gasthermo.eos.cubic.Cubic* method), 26  
`alpha_expr()` (*gasthermo.eos.cubic.RedlichKwong* method), 28  
`alpha_expr()` (*gasthermo.eos.cubic.SoaveRedlichKwong* method), 29

## B

`B0_expr()` (*gasthermo.eos.virial.Virial* method), 22  
`B1_expr()` (*gasthermo.eos.virial.Virial* method), 22  
`B_expr()` (*gasthermo.eos.virial.Virial* method), 22  
`B_ij_expr()` (*gasthermo.eos.virial.SecondVirialMixture* method), 32  
`B_mix_expr()` (*gasthermo.eos.virial.SecondVirialMixture* method), 32  
`bar_GiR_RT()` (*gasthermo.eos.virial.SecondVirialMixture* method), 32  
`bar_Hi_IG()` (*gasthermo.partial\_molar\_properties.Mixture* method), 17  
`bar_Hi_IG_star()` (*gasthermo.partial\_molar\_properties.MixtureDimensionless* method), 18  
`bar_HiR_RT()` (*gasthermo.eos.virial.SecondVirialMixture* method), 32  
`bar_HiR_star()` (*gasthermo.eos.virial.SecondVirialMixture* method), 32  
`bar_SiR_R()` (*gasthermo.eos.virial.SecondVirialMixture* method), 33  
`bar_Vi_IG()` (*gasthermo.partial\_molar\_properties.Mixture* method), 17  
`bar_ViR_RT()` (*gasthermo.eos.virial.SecondVirialMixture* method), 33  
`beta_expr()` (*gasthermo.eos.cubic.Cubic* method), 26

## C

`calc_Z_c()` (*gasthermo.critical\_constants.CriticalConstants* method), 42  
`calc_Z_from_dimensionless()` (*gasthermo.eos.virial.SecondVirial* method), 23  
`calc_Z_from_units()` (*gasthermo.eos.virial.SecondVirial* method), 23  
`calc_Z_from_units()` (*gasthermo.eos.virial.SecondVirialMixture* method), 33  
`cardano_constants()` (*gasthermo.eos.cubic.Cubic* method), 26  
`coefficients()` (*gasthermo.eos.cubic.Cubic* method), 27  
`cp_integral()` (*gasthermo.cp.CpIdealGas* method), 40  
`CpIdealGas` (class in *gasthermo.cp*), 39  
`CpRawData` (class in *gasthermo.cp*), 41  
`CpStar` (class in *gasthermo.cp*), 40  
`CriticalConstants` (class in *gasthermo.critical\_constants*), 41  
`Cubic` (class in *gasthermo.eos.cubic*), 25

## D

`d_B0_d_Tr_expr()` (*gasthermo.eos.virial.Virial* method), 22  
`d_B1_d_Tr_expr()` (*gasthermo.eos.virial.Virial* method), 22  
`d_Bij_d_Trij()` (*gasthermo.eos.virial.SecondVirialMixture* method), 33  
`d_dij_d_Tr()` (*gasthermo.eos.virial.SecondVirialMixture* method), 34  
`d_ik_expr()` (*gasthermo.eos.virial.SecondVirialMixture* method), 34  
`d_ln_alpha_d_ln_Tr()` (*gasthermo.eos.cubic.Cubic* method), 27  
`d_ln_alpha_d_ln_Tr()` (*gasthermo.eos.cubic.RedlichKwong* method), 27

28  
d\_ln\_alpha\_d\_ln\_Tr() (gas-thermo.eos.cubic.SoaveRedlichKwong method), 29

## E

enthalpy() (gasthermo.partial\_molar\_properties.Mixture method), 17  
enthalpy\_star() (gas-thermo.partial\_molar\_properties.MixtureDimensionless method), 18  
eval() (gasthermo.cp.CpIdealGas method), 40  
eval() (gasthermo.cp.CpStar method), 41  
eval() (gasthermo.thermal\_conductivity.ThermalConductivity method), 43  
eval() (gasthermo.viscosity.Viscosity method), 44  
eval\_HR() (gasthermo.thermal\_conductivity.ThermalConductivity method), 43  
eval\_simple() (gas-thermo.thermal\_conductivity.ThermalConductivityMixture method), 43  
eval\_Wilke() (gasthermo.viscosity.ViscosityMixture method), 44

## F

f\_w\_rule() (gasthermo.eos.cubic.PengRobinson method), 30  
f\_w\_rule() (gasthermo.eos.cubic.SoaveRedlichKwong method), 29  
fugacity\_i\_expr() (gas-thermo.eos.virial.SecondVirialMixture method), 34

## G

G\_RT() (gasthermo.eos.virial.SecondVirialMixture method), 32  
G\_RT\_expr() (gasthermo.eos.cubic.Cubic method), 25  
G\_RT\_expr() (gasthermo.eos.virial.SecondVirial method), 22  
gasthermo.cp module, 39  
gasthermo.critical\_constants module, 41  
gasthermo.eos.cubic module, 24  
gasthermo.eos.virial module, 19  
gasthermo.input module, 35  
gasthermo.partial\_molar\_properties module, 13  
gasthermo.thermal\_conductivity module, 42  
gasthermo.viscosity module, 43  
get\_max\_percent\_difference() (gas-thermo.cp.CpRawData method), 41  
get\_numerical\_percent\_difference() (gas-thermo.cp.CpIdealGas method), 40  
get\_point\_input() (gasthermo.input.IdealMixture method), 37  
get\_point\_input() (gasthermo.input.RealMixture method), 38  
get\_w\_Tc\_Pc() (gas-thermo.eos.virial.SecondVirialMixture method), 34

## H

H\_RT() (gasthermo.eos.virial.SecondVirialMixture method), 32  
H\_RT\_expr() (gasthermo.eos.cubic.Cubic method), 26  
H\_RT\_expr() (gasthermo.eos.virial.SecondVirial method), 23  
hat\_phi\_i\_expr() (gasthermo.eos.cubic.Cubic method), 27  
hat\_phi\_i\_expr() (gasthermo.eos.virial.Virial method), 22

## I

I\_expr() (gasthermo.eos.cubic.Cubic method), 26  
IdealMixture (class in gasthermo.input), 37  
iterate\_to\_solve\_Z() (gas-thermo.eos.cubic.Cubic method), 27

## L

ln\_hat\_phi\_k\_expr() (gasthermo.eos.cubic.Cubic method), 27  
ln\_hat\_phi\_k\_expr() (gas-thermo.eos.virial.SecondVirial method), 23  
ln\_hat\_phi\_k\_expr() (gas-thermo.eos.virial.SecondVirialMixture method), 34

## M

M\_R\_dimensionless() (gas-thermo.eos.virial.SecondVirialMixture method), 32  
MixingRule (class in gasthermo.eos.virial), 30  
Mixture (class in gas-thermo.partial\_molar\_properties), 17  
MixtureDimensionless (class in gas-thermo.partial\_molar\_properties), 17  
module  
gasthermo.cp, 39  
gasthermo.critical\_constants, 41

[gasthermo.eos.cubic](#), 24  
[gasthermo.eos.virial](#), 19  
[gasthermo.input](#), 35  
[gasthermo.partial\\_molar\\_properties](#), 13  
[gasthermo.thermal\\_conductivity](#), 42  
[gasthermo.viscosity](#), 43

## N

[num\\_roots\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 27  
[numerical\\_integration\(\)](#) ([gasthermo.cp.CpIdealGas method](#)), 40

## P

[P\\_cij\\_rule\(\)](#) ([gasthermo.eos.virial.MixingRule method](#)), 31  
[PengRobinson](#) ([class in gasthermo.eos.cubic](#)), 29  
[phi\\_ij\(\)](#) ([gasthermo.viscosity.ViscosityMixture method](#)), 44  
[plot\\_residual\\_HSG\(\)](#) ([gasthermo.eos.virial.SecondVirialMixture method](#)), 34  
[plot\\_Z\\_vs\\_P\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 27  
[plot\\_Z\\_vs\\_P\(\)](#) ([gasthermo.eos.virial.SecondVirial method](#)), 23  
[print\\_roots\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 28

## Q

[q\\_expr\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 28

## R

[RealMixture](#) ([class in gasthermo.input](#)), 37  
[RedlichKwong](#) ([class in gasthermo.eos.cubic](#)), 28  
[residual\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 28

## S

[S\\_R\\_R\(\)](#) ([gasthermo.eos.virial.SecondVirialMixture method](#)), 32  
[S\\_R\\_R\\_expr\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 26  
[S\\_R\\_R\\_expr\(\)](#) ([gasthermo.eos.virial.SecondVirial method](#)), 23  
[SecondVirial](#) ([class in gasthermo.eos.virial](#)), 22  
[SecondVirialMixture](#) ([class in gasthermo.eos.virial](#)), 31  
[set\\_point\\_input\(\)](#) ([gasthermo.input.IdealMixture method](#)), 37  
[set\\_point\\_input\(\)](#) ([gasthermo.input.RealMixture method](#)), 38  
[setup\(\)](#) ([gasthermo.input.IdealMixture method](#)), 37  
[SoaveRedlichKwong](#) ([class in gasthermo.eos.cubic](#)), 29

## T

[T\\_cij\\_rule\(\)](#) ([gasthermo.eos.virial.MixingRule method](#)), 31  
[ThermalConductivity](#) ([class in gasthermo.thermal\\_conductivity](#)), 42  
[ThermalConductivityMixture](#) ([class in gasthermo.thermal\\_conductivity](#)), 43

## V

[V\\_cij\\_rule\(\)](#) ([gasthermo.eos.virial.MixingRule method](#)), 31  
[Virial](#) ([class in gasthermo.eos.virial](#)), 22  
[Viscosity](#) ([class in gasthermo.viscosity](#)), 43  
[ViscosityMixture](#) ([class in gasthermo.viscosity](#)), 44

## W

[w\\_ij\\_rule\(\)](#) ([gasthermo.eos.virial.MixingRule method](#)), 31

## Z

[Z\\_c\\_percent\\_difference\(\)](#) ([gasthermo.critical\\_constants.CriticalConstants method](#)), 42  
[Z\\_cij\\_rule\(\)](#) ([gasthermo.eos.virial.MixingRule method](#)), 31  
[Z\\_right\\_hand\\_side\(\)](#) ([gasthermo.eos.cubic.Cubic method](#)), 26