gasthermo

Release 0.0.6

Robert F. De Jaco

May 23, 2020
CONTENTS:

1 Getting Started 3
  1.1 Installation .................................................. 3
  1.2 Single-Component Examples .................................. 3
  1.3 Mixture Examples .............................................. 8
  1.4 Gotchas ......................................................... 11

2 Definitions 13
  2.1 Nomenclature .................................................. 13

3 Partial Molar Properties 15
  3.1 Ideal Gas ....................................................... 15
  3.2 Residual ......................................................... 16

4 Equations of State 19
  4.1 Single Component .............................................. 19
  4.2 Multicomponent ............................................... 30

5 Input 37

6 Thermodynamic Property Data 39
  6.1 Heat Capacity .................................................. 39
  6.2 Critical Properties ............................................ 41
  6.3 Thermal Conductivity .......................................... 42
  6.4 Viscosity ....................................................... 43

7 References 45

8 Indices and tables 47

Bibliography 49

Python Module Index 51

Index 53
A package for obtaining gas-phase thermodynamic properties and computing integrals. Designed for integration with computational simulation environments.
1.1 Installation

The package can be installed via pip as follows

```bash
pip install GasThermo
```

1.2 Single-Component Examples

1.2.1 Heat Capacity

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

```python
>>> 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:

```python
>>> 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

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

And then we can save the results to a file

```python
>>> 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

```python
>>> 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)

>>> # 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

0.0
```
1.2.3 Virial Equation of State

```python
>>> 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.

```python
>>> 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

```python
>>> 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.

```python
>>> 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
```

(continues on next page)
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

```python
>>> from gasthermo.eos.virial import SecondVirialMixture
>>> P, T = 1e5, 300.
>>> mixture = SecondVirialMixture(compound_names=['Water', 'Tetrahydrofuran'], k_ij=0.1)
>>> 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

```python
>>> 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

```python
>>> I.enthalpy(ys=[0.5, 0.5], P=1e5, T=300.)
9307.3883
>>> I.enthalpy(ys=[0.5, 0.5], P=1e5, T=300., T_ref=0.)
```

(continues on next page)
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,

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

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

```python
>>> 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 \( \bar{H}_i^{\text{pure}} - H^{\text{pure}} = 0 \).

```python
>>> 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

```python
>>> 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

```python
>>> 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.enhalpy(ys=[0.4, 0.3, 0.17, 0.13], P=5e5, T=300.) - N.enhalpy(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

```python
>>> 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.enhalpy(ys=[0.4, 0.3, 0.3, 0.], P=5e5, T=300.) - Nm1.enhalpy(ys=[0.4, 0.3, 0.
˓→3], P=5e5, T=300.) 0.0
```

1.4 Gotchas

- All units are SI units
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

<table>
<thead>
<tr>
<th>Code</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>$P$</td>
<td>Pressure in Pa</td>
</tr>
<tr>
<td>V</td>
<td>$V$</td>
<td>Molar Volume in $m^3/mol$</td>
</tr>
<tr>
<td>R</td>
<td>$R$</td>
<td>gas constant SI units ($m^3 \times Pa/mol/K$)</td>
</tr>
<tr>
<td>T</td>
<td>$T$</td>
<td>temperature in K</td>
</tr>
<tr>
<td>T_c</td>
<td>$T_c$</td>
<td>critical temperature in K</td>
</tr>
<tr>
<td>P_c</td>
<td>$P_c$</td>
<td>critical pressure in Pa</td>
</tr>
<tr>
<td>T_r</td>
<td>$T_r$</td>
<td>reduced temperature (dimensionless)</td>
</tr>
<tr>
<td>V_c</td>
<td>$V_c$</td>
<td>Critical volume $m^3/mol$</td>
</tr>
<tr>
<td>w</td>
<td>$\omega$</td>
<td>Accentric factor</td>
</tr>
<tr>
<td>y_i</td>
<td>$y_i$</td>
<td>mole fraction of component $i$</td>
</tr>
<tr>
<td>- n_i</td>
<td>$n_i$</td>
<td>number of moles of component $i$</td>
</tr>
<tr>
<td>S</td>
<td>$S$</td>
<td>Molar entropy</td>
</tr>
<tr>
<td>H</td>
<td>$H$</td>
<td>Molar enthalpy</td>
</tr>
<tr>
<td>G</td>
<td>$G$</td>
<td>Molar Gibbs free energy</td>
</tr>
</tbody>
</table>
We define partial molar property $\bar{M}_i$ of species $i$ in a mixture as

$$\bar{M}_i = \left( \frac{\partial (nM)}{\partial n_i} \right)_{P,T,n_j}$$  \hspace{1cm} (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

$$\bar{M}_i = \bar{M}^{IG}_i + \bar{M}^R_i$$  \hspace{1cm} (3.2)

$$M^R = \sum_i y_i \bar{M}^R_i$$  \hspace{1cm} (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, $\bar{V}^{IG}_i$, is

$$\bar{V}^{IG}_i = \frac{RT}{P}$$  \hspace{1cm} (3.4)

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

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

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

$$\bar{H}^{IG}_i = H^{IG}_i = \int_{T_{in}}^{T} C^{IG}_{i,p}dT$$  \hspace{1cm} (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,

$$
\bar{H}_i^{\text{IG,*}} = \frac{\bar{H}_i^{\text{IG}}}{RT_{\text{ref}}} = \int_1^{T^*} \frac{C_p^{\text{IG,*}}}{R} d(T')
$$

or

$$
\bar{H}_i^{\text{IG,*}} = \int_1^{T^*} C_p^{\text{IG,*}} dT'
$$

where

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

is a dimensionless variable and

$$
C_p^{\text{IG,*}} = \frac{C_p^{\text{IG}}}{R}
$$

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 $i$, $\bar{V}_i^R$, can be calculated as

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

(3.10)

Defining the dimensionless quantity

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

The expression in dimensionless units can be simplified to

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

(3.11)

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

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

(3.12)
Defining the dimensionless quantity

\[ \bar{H}_i^{R,*} = \frac{\bar{H}_i^R}{RT_{\text{ref}}} \]

(3.13)

\[ = \frac{RT^2}{RT_{\text{ref}}} \left( \frac{\partial \ln \hat{\phi}_i}{\partial T} \right) \]

(3.14)

\[ = -\frac{RT^2}{RT_{\text{ref}}^2} \left( \frac{\partial \ln \hat{\phi}_i}{\partial T^*} \right) \]

(3.15)

(3.16)

The expression in dimensionless units is computed as

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

(3.17)

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

\[ \bar{G}_i^R = RT \ln \hat{\phi}_i \]

(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 gasthemo.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 gasthemo.eos.virial.SecondVirialMixture

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

Parameters

- **T** – temperature in K
- **T_ref** – reference temperature in K

Returns \( \bar{H}_i^{xTIG} \), see Equation (3.5)

bar_Vi_IG(T, P)

Parameters

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

Returns \( \bar{V}_i^{xTIG} \), 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.
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 $\bar{H}_{\text{extIG}}$, 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
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]

\[ Z = 1 + B \rho = 1 + \frac{BP}{RT} \]  

(4.1)

Where the composition dependency of \( B \) is given by the exact mixing rule

\[ B = \sum_i \sum_j y_i y_j B_{ij} \]  

(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

\[ B = \frac{RT_c}{P_c} (B^0 + \omega B^1) \]  

(4.3)

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

\[ Z = 1 + (B^0 + \omega B^1) \frac{P_r}{T_r} \]  

(4.4)

where [SVanNessA05]

\[ B^0 = 0.083 - \frac{0.422}{T_r^{1.6}} \]  

(4.5)

\[ B^1 = 0.139 - \frac{0.172}{T_r^{1.2}} \]  

(4.6)
so that

so that the following derivatives can be computed as

\[
\frac{d B^0}{dT} = \frac{0.675}{T_r^{2.75}} \tag{4.7}
\]

\[
\frac{d B^1}{dT} = \frac{0.722}{T_r^{5.2}} \tag{4.8}
\]

Which allow the \( H^R \), \( S^R \), and \( G^R \) to be readily computed as follows \[GP07]\n
\[
\frac{G^R}{RT} = (B^0 + \omega B^1) \frac{P_r}{T_r} \tag{4.9}
\]

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

\[
\frac{S^R}{R} = -P_r \left( \frac{d B^0}{dT_r} - \omega \frac{d B^1}{dT_r} \right) \tag{4.11}
\]

The cross coefficients are calculated as

\[
B_{ij} = \frac{RT_{cij}}{P_{cij}} \left( B^0 + \omega_{ij} B^1 \right) \tag{4.12}
\]

so that the cross derivatives can be computed as

\[
\frac{d B_{ij}}{dT} = \frac{RT_{cij}}{P_{cij}} \left( \frac{d B^0}{dT} + \omega_{ij} \frac{d B^1}{dT} \right) \tag{4.13}
\]

\[
\frac{d B_{ij}}{dT_{rij}} = \frac{RT_{cij}}{P_{cij}} \left( \frac{d B^0}{dT_{rij}} + \omega_{ij} \frac{d B^1}{dT_{rij}} \right) \tag{4.13}
\]

Fugacity Coefficients

The Fugacity coefficients are calculated as

\[
\ln \hat{\phi}_i = \left( \frac{\partial (nG^R/R/T)}{\partial n_i} \right)_{P,T,n_j} \tag{4.14}
\]

For the virial equation of state, this becomes \[VanNessA82]\n
\[
\ln \hat{\phi}_k = \frac{P}{RT} \left[ B_{kk} + \frac{1}{2} \sum_i \sum_j y_i y_j (2 \delta_{ik} - \delta_{ij}) \right] \tag{4.14}
\]

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

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

The partial molar residual enthalpy of component $k$ is

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

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

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

Since

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

In terms of partial molar properties, then

$$
\bar{S}^R_i = \frac{\bar{H}^R_i - G^R_i}{T} \tag{4.21}
$$

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

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

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

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

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

$$
\frac{\bar{V}^R_k}{RT} = \left( \frac{\partial \ln \hat{\phi}_i}{\partial P} \right)_{T,y}
= \frac{\partial}{\partial P} \left\{ \frac{P}{RT} \left[ B_{kk} + \frac{1}{2} \sum_i \sum_j y_i y_j \left( 2\delta_{ik} - \delta_{ij} \right) \right] \right\} \tag{4.25}
$$

which simplifies to

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

4.1. Single Component
from which we obtain the intuitive result of

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

**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 temperature [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)

$H_{\text{R-RT}} (P, T)$

**Parameters**

• $P$ – pressure in Pa  
• $T$ – temperature in K

**Returns**  
Equation (4.10)

$S_{\text{R-R}} (P, T)$

**Parameters**

• $P$ – pressure in Pa  
• $T$ – temperature in K

**Returns**  
Equation (4.11)

$\text{calc}_Z_{\text{from}}_{\text{dimensionless}} (P_r, T_r)$

**Parameters**

• $P_r$ – reduced pressure, dimensionless  
• $T_r$ – reduced temperature, dimensionless

**Returns**  
Equation (4.4)

$\text{calc}_Z_{\text{from}}_{\text{units}} (P, T)$

**Parameters**

• $P$ – pressure in Pa  
• $T$ – temperature in K

**Returns**  
Equation (4.1)

$\ln \hat{\phi}_k (P, T)$

**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

$\text{plot}_Z_{\text{vs}}_{\text{P}} (T, P_{\text{min}}, P_{\text{max}}, symbol='o', ax=None, **kwargs)$

Plot compressibility as a function of pressure

**Parameters**

• $T$ (*float*) – temperature [K]  
• $P_{\text{min}}$ (*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

\[ a(T) = \Psi \frac{\alpha(T_r) R^2 T^2_c}{P_c} \]  \hspace{1cm} (4.28)

\[ b = \Omega \frac{RT_c}{P_c} \]  \hspace{1cm} (4.29)

The compressibility factor can be calculated by solving the following equation

\[ Z - \left(1 + \beta - \frac{q \beta(Z - \beta)}{(Z + \epsilon \beta)(Z + \sigma \beta)} \right) = 0 \]  \hspace{1cm} (4.30)

where

\[ \beta = \Omega \frac{P_r}{T_r} \]  \hspace{1cm} (4.31)

and

\[ q = \frac{\Psi \alpha(T_r)}{\Omega T_r} \]  \hspace{1cm} (4.32)

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

\[ Z\text{new} = 1 + \beta - \frac{q \beta(Z\text{old} - \beta)}{(Z\text{old} + \epsilon \beta)(Z\text{old} + \sigma \beta)} \]  \hspace{1cm} (4.33)

that is continued until the following is true

\[ \frac{\|Z\text{new} - Z\text{old}\|}{Z\text{new} + Z\text{old}} \times 200 < \text{tol} \]  \hspace{1cm} (4.34)

Residual Molar Properties

\[ \frac{H^R}{RT} = Z - 1 + \left[ \frac{\dln \alpha(T_r)}{\dln T_r} - 1 \right] qI \]  \hspace{1cm} (4.35)

\[ \frac{G^R}{RT} = Z - 1 + \ln(Z - \beta) - qI \]  \hspace{1cm} (4.36)
\[
\frac{SR}{R} = \ln(Z - \beta) + \frac{d \ln \alpha(T_i)}{dT_r} q I
\]  
(4.36)

where the following functions have been defined

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

\[
1 + \beta - \frac{q \beta (Z - \beta)}{(Z + \epsilon \beta)(Z + \sigma \beta)}
\]  
(4.38)

**Fugacity Coefficients**

The pure-component fugacity coefficient is defined as

\[
\frac{G^R_{i}}{RT_i} = \ln \hat{\phi}_i
\]

So that, from Equation (4.35),

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

**Mixtures**

**Warning:** Not implemented yet!

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

```python
class gasthemo.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 m**3/mol

4.1. Single Component
• T – temperature in K

Returns $\frac{G^*}{RT}$, 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 $\frac{H^*}{RT}$, 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 $\frac{S^*}{R}$, 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/Tc), 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

\[
Z^3c_0 + Z^2c_1 + Z * c_2 + c_3 = 0
\]

**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 \(d \ln \alpha (T_r)\)

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

expression for fugacity coefficient \(\hat{\phi}_i\)

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 \(ln_at_{phi_i}\)

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))

```python
class gasthermo.eos.cubic.RedlichKwong(**kwargs)
    Redlich-Kwong Equation of State [RK49]
    This Equation of state has the following parameters [SVanNessA05]
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>α(T_r)</td>
<td>1/\sqrt{T_r}</td>
</tr>
<tr>
<td>σ</td>
<td>1</td>
</tr>
<tr>
<td>ϵ</td>
<td>0</td>
</tr>
<tr>
<td>Ω</td>
<td>0.08664</td>
</tr>
<tr>
<td>Ψ</td>
<td>0.42748</td>
</tr>
</tbody>
</table>

```python
>>> 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 α(T_r) see Table ??
\[ \frac{\mathrm{d}\ln\alpha}{\mathrm{d}\ln T_r} (T_r) \]

**Parameters** \( T_r \) – reduced temperature [dimensionless]

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

```python
class gasthemo.eos.cubic.SoaveRedlichKwong(**kwargs)
Soave Redlich-Kwong Equation of State [Soa72]
```

This equation of state has the following parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha(T_r) )</td>
<td>( [1 + f_w(1 - \sqrt{T_r})]^2 )</td>
</tr>
<tr>
<td>( \sigma )</td>
<td>1</td>
</tr>
<tr>
<td>( \epsilon )</td>
<td>0</td>
</tr>
<tr>
<td>( \Omega )</td>
<td>0.08664</td>
</tr>
<tr>
<td>( \Psi )</td>
<td>0.42748</td>
</tr>
</tbody>
</table>

where

\[ f_w = 0.480 + 1.574\omega - 0.176\omega^2 \]  (4.40)

```python
>>> from gasthemo.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 ??

\[ \frac{\mathrm{d}\ln\alpha}{\mathrm{d}\ln T_r} (T_r) \]

Parameters \( T_r \) – reduced temperature [dimensionless]

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

**f_w_rule** \( (w) \)

Parameters \( w \) – accentric factor

Returns \( f_w \), see Equation (4.40)

4.1. Single Component 29
class gasthermo.eos.cubic.PengRobinson(**kwargs)
Peng-Robinson Equation of State [PR76]

This equation of state has the following parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha(T_r)$</td>
<td>$[1 + f_w(1 - \sqrt{T_r})^2]$</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>$1 + \sqrt{2}$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>$1 - \sqrt{2}$</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>$0.07780$</td>
</tr>
<tr>
<td>$\Psi$</td>
<td>$0.45724$</td>
</tr>
</tbody>
</table>

where

$$f_w = 0.480 + 1.574\omega - 0.176\omega^2$$  \hspace{1cm} (4.41)

```python
>>> 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 $w$ – accentric 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]

$$\omega_{ij} = \frac{\omega_i + \omega_j}{2}$$  \hspace{1cm} (4.42)

$$T_{cij} = \sqrt{T_{c_i}T_{c_j}(1 - k_{ij})}$$  \hspace{1cm} (4.43)
\[ P_{cij} = \frac{Z_{cij}RT_{cij}}{V_{cij}} \]  
\[ Z_{cij} = \frac{Z_{ci} + Z_{cj}}{2} \]  
\[ V_{cij} = \left( \frac{V_{ci}^{1/3} + V_{cj}^{1/3}}{2} \right)^3 \]  

**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** – critical temperature of component 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** 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

### 4.2. Multicomponent
• `kwargs` – key-word arguments to pass to `RealMixture`

\[ B_{ij\text{\_expr}} (i: \text{ int}, j: \text{ int}, T) \]

**Parameters**
- \( i \) – index of first component
- \( j \) – index of second component
- \( T \) – temperature [K]

**Returns** Equation (4.12)

\[ B_{\text{mix\_expr}} (y_k: \text{ List[Union[float, Any]]}, T) \]

**Parameters**
- \( y_k \) – mole fractions of each component \( k \)
- \( T \) – temperature in K

**Returns** Equation (4.2)

\[ G_{R\text{\_RT}}(*args) \]
Residual free energy of mixture \( G^R / R/T \)

\[ H_{R\text{\_RT}}(*args) \]
Residual enthalpy of mixture \( H^R / R/T \)

\[ M_{R\text{\_dimensionless}} (\text{method: callable, ys: List[Union[float, Any]], P: float, T: float}) \]
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 \( S^R / R \)

\[ \text{bar\_GiR\_RT} (\text{cas\_k: str, ys: List[Union[float, Any]], P, T}) \]
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)

\[ \text{bar\_HiR\_RT} (\text{cas\_k: str, ys: List[Union[float, Any]], P, T}) \]
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 \((T_{\text{star}}, \text{cas}_k: \text{str}, \text{ys}: \text{List}[\text{Union}[\text{float}, \text{Any}]], P, T)\)

Returns the scaled entropy useful for computations \(\bar{H}_{i}^{R,*}\), as defined in Equation (3.17)

**Parameters**
- \text{cas}_k – cas number of component \(k\)
- \text{ys} – mole fractions
- \(P\) – pressure in Pa
- \(T\) – temperature in K

bar_SiR_R \((\text{cas}_k: \text{str}, \text{ys}: \text{List}[\text{Union}[\text{float}, \text{Any}]], P, T)\)

Dimensionless residual partial molar entropy of component \(i\)

**Parameters**
- \text{cas}_k – cas number for component of interest
- \text{ys} – mole fractions
- \(P\) – pressure in Pa
- \(T\) – temperature in K

**Returns** Equation (4.24)

bar_ViR_RT \((\text{cas}_k: \text{str}, \text{ys}: \text{List}[\text{Union}[\text{float}, \text{Any}]], P, T)\)

residual Partial molar volume for component \(i\)

**Note:** This expression does not depend on \(P\)

**Parameters**
- \text{cas}_k – cas number for component of interest
- \text{ys} – mole fractions
- \(P\) – pressure in Pa
- \(T\) – temperature in K

**Returns** Equation (4.27)

calc_Z_from_units \((y_k: \text{List}, P, T)\)

**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 \((i: \text{int}, j: \text{int}, T)\)

**Parameters**
- \(i\) – index for component \(i\)
- \(j\) – index for component \(j\)
- \(T\) – temperature in K

4.2. Multicomponent
**Returns** Equation (4.13)

\[ \frac{\partial \delta_{ij}}{\partial T_{rij}} = 2 \frac{\partial B_{ij}}{\partial T_{rij}} - \frac{\partial B_{ii}}{\partial T_{rij}} - \frac{\partial B_{jj}}{\partial T_{rij}} \]  

(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]

\( d_{ik} \) \text{expr}(i: \text{int}, k: \text{int}, T)

**Parameters**
- \( i \) – index of component \( i \)
- \( k \) – index of component \( k \)
- \( T \) – temperature [K]

**Returns** Equation (4.15)

\( \text{fugacity}_{i} \) \text{expr}(cas_i: \text{str}, ys: \text{List}[\text{Union[float, Any]}], P, T)

Fugacity of component \( i \) in mixture \( f_i = \hat{\phi}_i y_i P \)

**Parameters**
- \( \text{cas}_i \) – cas number for component of interest
- \( ys \) – mole fractions
- \( P \) – pressure in Pa
- \( T \) – temperature in K

**Returns**

**get_w_Tc_Pc** \((i: \text{int}, j=\text{None})\)

Returns critical constants for calculation based off of whether \( i = j \) or not

**Returns** \((w, T_c, P_c)\)

**Return type** tuple

\( \ln \hat{\phi}_k \) \text{expr}(k: \text{int}, ys: \text{List}[\text{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
CHAPTER FIVE

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 sin 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**3/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.

```python
def get_point_input(i: int) -> dict:
    Parameters i – index of point
    Returns keyword arguments for point input

def set_point_input(i: int, **kwargs):
    Parameters i – index of point
```
6.1 Heat Capacity

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

$$C_p^{IG} = C_1 + C_2 \left[ \frac{C_3}{\sinh(C_3/T)} \right] + C_4 \left[ \frac{C_5}{\cosh(C_5/T)} \right]$$ \hspace{1cm} (6.1)

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

```python
cp_integral(T_a, T_b)
Evaluate integral
\[ \int_{T_a}^{T_b} C_p^{IG}(T')dT' \] (6.2)
```

**Parameters**

• **T_a** – start temperature in K

• **T_b** – finish temperature in K

**Returns** integral

```python
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))

```python
get_numerical_percent_difference()
Calculate the percent difference with numerical integration
```

```python
numerical_integration(T_a, T_b) \rightarrow \text{tuple}
Numerical integration using scipy
```

```python
class gasthermo.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

$$C_p^* = \frac{C_p^{IG}}{R}$$

(6.3)

$$T^* = \frac{T}{T_{ref}}$$

(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

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

(6.5)
where

\[
\begin{align*}
C_1^* &= \frac{C_1}{R} \\
C_2^* &= \frac{C_2}{R} \\
C_3^* &= \frac{C_3}{T_{\text{ref}}} \\
C_4^* &= \frac{C_4}{R} \\
C_5^* &= \frac{C_5}{T_{\text{ref}}}
\end{align*}
\] (6.6)

**Parameters**

\texttt{T\_ref} (float) – reference temperature [K] for dimensionless computations

\texttt{eval} (\texttt{T}, \texttt{f\_sinh=<ufunc 'sinh'>}, \texttt{f\_cosh=<ufunc 'cosh'>})

**Parameters**

- \texttt{T} – temperature in K
- \texttt{f\_sinh} (callable) – function for hyperbolic sine, defaults to \texttt{np.sinh}
- \texttt{f\_cosh} (callable) – function for hyperbolic cosine, defaults to \texttt{np.cosh}

**Returns**

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

class gasthermo.cp.CpRawData (\texttt{T\_raw: list}, \texttt{Cp\_raw: list}, \texttt{T\_min\_fit: float = None}, \texttt{T\_max\_fit: float = None}, \texttt{poly\_order: int = 2}, \texttt{T\_units='K'}, \texttt{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**

- \texttt{T\_min\_fit} (float, optional) – minimum temperature for fitting function [K]
- \texttt{T\_max\_fit} (float, optional) – maximum temperature for fitting function [K]
- \texttt{poly\_order} (int, optional) – order of polynomial for fitting, defaults to 2
- \texttt{T\_raw} (list) – raw temperatures in K
- \texttt{Cp\_raw} (list) – raw heat capacities in J/K/mol
- \texttt{Cp\_units} (str, optional) – units for \(C_p\), defaults to J/mol/K
- \texttt{T\_units} (str, optional) – units for \(T\), defaults to K

\texttt{get\_max\_percent\_difference}()

Get largest percent difference

### 6.2 Critical Properties

class gasthermo.critical\_constants.CriticalConstants (\texttt{dippr\_no: str = None}, \texttt{compound\_name: str = None}, \texttt{cas\_number: str = None}, \texttt{MW: float = None}, \texttt{P\_c: float = None}, \texttt{V\_c: float = None}, \texttt{Z\_c: float = None}, \texttt{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 [m³/mol]
- `Z_c` – critical compressibility factor [dimensionless]
- `w` – accentric factor [dimensionless]
- `tol (float, hard-coded)` – tolerance for percent difference in Zc calculated and tabulated, set to 0.5

```python
Z_c_percent_difference()
```
calculate percent difference between Z_c calculated and tabulated

```python
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]

\[
k = \frac{C_1 T^{C_2}}{1 + C_3/T + C_4/T^2}
\]

(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
• \( T_{\text{min}} \) (float, derived from input) – minimum temperature of validity for relationship [K]
• \( T_{\text{max}} \) (float, derived from input) – maximum temperature of validity [K]
• \( C_1 \) (float, derived from input) – parameter in Equation (6.11)
• \( C_2 \) (float, derived from input) – parameter in Equation (6.11)
• \( C_3 \) (float, derived from input) – parameter in Equation (6.11)
• \( C_4 \) (float, derived from input) – parameter in Equation (6.11)
• \( \text{units} \) (str) – units for \( k \), set to W/m/K
• \( T_{\text{min\_fit}} \) – minimum temperature for fit, defaults to \( T_{\text{min}} \)
• \( T_{\text{max\_fit}} \) – maximum temperature for fit, defaults to \( T_{\text{max}} \)

\[
eval(T)
\]

Parameters \( T \) – temperature in K
Returns \( k \) W/m/K (see equation (6.11))

class gasthermo.thermal_conductivity.ThermalConductivityMixture (name_to_cas:\ndict, mixing_rule='Simple')

Viscosity of vapor mixture using Wilke mixing rule

Parameters

• \( \text{name\_to\_cas} \) (dict[components, str]) – mapping of chemical name to cas registry number
• \( \text{mixing\_rule} \) (str, optional) – mixing rule for calculation of viscosity, defaults to Simple
• \( \text{pure} \) (dict[components, Viscosity]) – pure component viscosity info, obtained from gasthermo.vapor_viscosity.Viscosity

\[
eval_{\text{HR}}(y_i, T)
\]
Weights based off of sqrt of molecular weights

\[
eval_{\text{simple}}(y_i, T)
\]
Calculate thermal conductivity using simple relationship

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

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]

\[
\mu = \frac{C_1 T^{C_2}}{1 + C_3/T + C_4/T^2}
\]

(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 gasthemo.viscosity.ViscosityMixture**

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 $j$
REFERENCES
INDICES AND TABLES

- genindex
- modindex
- search
BIBLIOGRAPHY


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
### INDEX

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_expr()</code></td>
<td><code>calc_Z_c()</code></td>
</tr>
<tr>
<td><code>alpha_expr()</code></td>
<td><code>calc_Z_from_dimensionless()</code></td>
</tr>
<tr>
<td><code>alpha_expr()</code></td>
<td><code>calc_Z_from_units()</code></td>
</tr>
<tr>
<td><code>alpha_expr()</code></td>
<td><code>calc_Z_from_units()</code></td>
</tr>
</tbody>
</table>

### B

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>B0_expr()</code></td>
<td><code>B1_expr()</code></td>
</tr>
<tr>
<td><code>B_expr()</code></td>
<td><code>B_mix_expr()</code></td>
</tr>
<tr>
<td><code>B_iij_expr()</code></td>
<td><code>d_B0_d_Tr_expr()</code></td>
</tr>
<tr>
<td><code>d_B1_d_Tr_expr()</code></td>
<td><code>d_Pij_d_Trij()</code></td>
</tr>
<tr>
<td><code>d_Bij_d_Trij()</code></td>
<td><code>d_dij_d_Tr()</code></td>
</tr>
</tbody>
</table>

### C

<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bar_GiR_RT()</code></td>
<td><code>bar_Hi_R()</code></td>
</tr>
<tr>
<td><code>bar_GiR_RT()</code></td>
<td><code>CpIntegral()</code></td>
</tr>
<tr>
<td><code>bar_Hi_IG()</code></td>
<td><code>CpIdealGas</code></td>
</tr>
<tr>
<td><code>bar_Hi_IG_star()</code></td>
<td><code>CpStar</code></td>
</tr>
<tr>
<td><code>bar_Hi_IG_star()</code></td>
<td><code>CriticalConstants</code></td>
</tr>
<tr>
<td><code>bar_Hi_R()</code></td>
<td><code>CriticalConstants</code></td>
</tr>
</tbody>
</table>

### D

<table>
<thead>
<tr>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d_dij_d_Tr()</code></td>
<td><code>d_ik_expr()</code></td>
</tr>
<tr>
<td><code>d_ik_expr()</code></td>
<td><code>d_ln_alpha_d_ln_Tr()</code></td>
</tr>
</tbody>
</table>

### E

<table>
<thead>
<tr>
<th>E</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calc_Z_c()</code></td>
<td><code>calc_Z_from_units()</code></td>
</tr>
<tr>
<td><code>calc_Z_from_dimensionless()</code></td>
<td><code>CpIdealGas</code></td>
</tr>
<tr>
<td><code>calc_Z_from_units()</code></td>
<td><code>CpStar</code></td>
</tr>
</tbody>
</table>

### F

<table>
<thead>
<tr>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cardano_constants()</code></td>
<td><code>cardano_constants()</code></td>
</tr>
<tr>
<td><code>coefficients()</code></td>
<td><code>coefficients()</code></td>
</tr>
</tbody>
</table>

### G

<table>
<thead>
<tr>
<th>G</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>calc_Z_c()</code></td>
<td><code>calc_Z_from_units()</code></td>
</tr>
<tr>
<td><code>calc_Z_from_dimensionless()</code></td>
<td><code>Cubic</code></td>
</tr>
<tr>
<td><code>calc_Z_from_units()</code></td>
<td><code>Cubic</code></td>
</tr>
</tbody>
</table>
gasthermo, Release 0.0.6

d\_ln\_alpha\_d\_ln\_Tr() (gasthermo.eos.cubic.SoaveRedlichKwong method), 29
d\_ln\_alpha\_d\_ln\_Tr() (gasthermo.eos.cubic.SoaveRedlichKwong method), 29

gasthermo.viscosity module, 43
get_max_percent_difference() (gasthermo.cp.CpRawData method), 41
get_numerical_percent_difference() (gasthermo.cp.CpIdealGas method), 40

enthalpy() (gasthermo.partial\_molar\_properties.Mixture method), 17
enthalpy\_star() (gasthermo.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\_HR() (gasthermo.thermal\_conductivity.ThermalConductivity method), 43
eval\_simple() (gasthermo.thermal\_conductivity.ThermalConductivity method), 43
eval\_Wilke() (gasthermo.viscosity.ViscosityMixture method), 44

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

G

G\_R\_RT() (gasthermo.eos.virial.SecondVirialMixture method), 32
G\_R\_RT\_expr() (gasthermo.eos.cubic.Cubic method), 26
G\_R\_RT\_expr() (gasthermo.eos.virial.SecondVirial method), 23
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

H

H\_R\_RT() (gasthermo.eos.virial.SecondVirialMixture method), 32
H\_R\_RT\_expr() (gasthermo.eos.cubic.Cubic method), 26
H\_R\_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() (gasthermo.eos.virial.SecondVirial method), 27

L

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

M

M\_R\_dimensionless() (gasthermo.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
Index