phasepy Documentation

Release 0.0.1

Gustavo Chaparro M.

Sep 20, 2023

Contents:

CHAPTER 1

Introduction

Phasepy is an open-source scientific Python package for calculation of physical properties of phases at thermodynamic equilibrium. Main application areas include computation of fluid phase equilibria and interfacial properties.

Phasepy includes routines for calculation of vapor-liquid equilibrium (VLE), liquid-liquid equilibrium (LLE) and vapor-liquid-liquid equilibrium (VLLE). Phase equilibrium can be modelled either with *the continous approach*, using a combination of a cubic equation of state (EoS, e.g. Van der Waals, Peng-Robinson, Redlich-Kwong, or their derivatives) model and a mixing rule (Quadratic, Modified Huron-Vidal or Wong-Sandler) for all phases, or *the discontinuous approach* using a virial equation for the vapor phase and an activity coefficient model (NRTL, Wilson, Redlich-Kister or Dortmund Modified UNIFAC) for the liquid phase(s).

Interfacial property estimation using the continuous phase equilibrium approach allows calculation of density profiles and interfacial tension using the Square Gradient Theory (SGT).

Phasepy supports fitting of model parameter values from experimental data.

CHAPTER 2

Installation Prerequisites

- numpy
- scipy
- pandas
- openpyxl
- C/C++ Compiler for Cython extension modules

CHAPTER 3

Installation

Get the latest version of phasepy from https://pypi.python.org/pypi/phasepy/

An easy installation option is to use Python pip:

```
$ pip install phasepy
```

Alternatively, you can build phasepy yourself using latest source files:

```
$ git clone https://github.com/gustavochm/phasepy
```

CHAPTER 4

Documentation

Phasepy's documentation is available on the web:

https://phasepy.readthedocs.io/en/latest/

CHAPTER 5

Getting Started

Base input variables include temperature [K], pressure [bar] and molar volume [cm^3/mol]. Specification of a mixture starts with specification of pure components:

```
>>> from phasepy import component, mixture
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948,
                      w=0.344861, GC={'H2O':1})
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0,
                        w=0.643558, GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(ethanol, water)
```

Here is an example how to calculate the bubble point vapor composition and pressure of saturated 50 mol-% ethanol - 50 mol-% water liquid mixture at temperature 320 K using Peng Robinson EoS. In this example the Modified Huron Vidal mixing rule utilizes the Dortmund Modified UNIFAC activity coefficient model for the solution of the mixture EoS.

```
>>> mix.unifac()
>>> from phasepy import preos
>>> eos = preos(mix, 'mhv_unifac')
>>> from phasepy.equilibrium import bubblePy
>>> y_guess, P_guess = [0.2, 0.8], 1.0
>>> bubblePy(y_guess, P_guess, X=[0.5, 0.5], T=320.0, model=eos)
(array([0.70761727, 0.29238273]), 0.23248584919691206)
```

For more examples, please have a look at the Jupyter Notebook files located in the *examples* folder of the sources or view examples in github.

CHAPTER 6

Bug reports

To report bugs, please use the phasepy's Bug Tracker at:

https://github.com/gustavochm/phasepy/issues

CHAPTER 7

License information

See `LICENSE.txt` for information on the terms & conditions for usage of this software, and a DISCLAIMER OF ALL WARRANTIES.

Although not required by the phasepy license, if it is convenient for you, please cite phasepy if used in your work. Please also consider contributing any changes you make back, and benefit the community.

Chaparro, G., Mejía, A. Phasepy: A Python based framework for fluid phase equilibria and interfacial properties computation. J Comput Chem. 2020; 1– 23. https://doi.org/10.1002/jcc.26405.

7.1 phasepy

Phasepy aims to require minimum quantity of parameters needed to do phase equilibrium calculations. First it is required to create components and mixtures, and then combine them with a phase equilibrium model to create a final model object, which can be used to carry out fluid phase equilibrium calculations.

7.1.1 phasepy.component

`phasepy.component` object stores pure component information needed for equilibria and interfacial properties computation. A component can be created as follows:

```
>>> from phasepy import component
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
```

Besides storing pure component data, the class incorporates basics methods for e.g. saturation pressure evaluation using Antoine equation, and liquid volume estimation with Rackett equation.

```
>>> water.psat(T=373.0) # vapor saturation pressure [bar]
1.0072796747419537
>>> water.vlrackett(T=310.0) # liquid molar volume [cm3/mol]
16.46025809309672
```

Warning: User is required to supply the necessary parameters for methods

class component (*name='None', Tc=0, Pc=0, Zc=0, Vc=0, w=0, c=0, cii=0, ksv=[0, 0], Ant=[0, 0, 0], GC=None, Mw=1.0, ri=0.0, qi=0.0, alpha_params=0.0, dHf=0.0, Tf=0.0, ms=1, sigma=0, eps=0, lambda_r=12.0, lambda_a=6.0, eAB=0.0, rcAB=1.0, rdAB=0.4, sites=[0, 0, 0]*)

Object class for storing pure component information.

Parameters

- **name** (*str*) – Name of the component
- **Tc** (*float*) – Critical temperature [K]
- **Pc** (*float*) – Critical pressure [bar]
- **Zc** (*float*) – Critical compressibility factor
- **Vc** (*float*) – Critical molar volume [$\mathrm{cm}^3/\mathrm{mol}$]
- **w** (*float*) – Acentric factor
- **c** (*float*) – Volume translation parameter used in cubic EoS [$\mathrm{cm}^3/\mathrm{mol}$]
- **cii** (*List[float]*) – Polynomial coefficients for influence parameter used in SGT model
- **ksv** (*List[float]*) – Parameter for alpha for PRSV EoS
- **Ant** (*List[float]*) – Antoine correlation parameters
- **GC** (*dict*) – Group contribution information used in Modified-UNIFAC activity coefficient model. Group definitions can be found here.
- **Mw** (*float*) – molar weight of the fluid [g/mol]
- **ri** (*float*) – Component molecular volume for UNIQUAC model
- **qi** (*float*) – Component molecular surface for UNIQUAC model
- **alpha_params** (*Any*) – Parameters for alpha function used in cubic EoS
- **dHf** (*float*) – Enthalpy of fusion [J/mol]
- **Tf** (*float*) – Temperature of fusion [K]

ci (*T*)

Returns value of SGT model influence parameter [$\mathrm{Jm}^5/\mathrm{mol}$] at a given temperature.

Parameters **T** (*float*) – absolute temperature [K]

psat (*T*)

Returns vapor saturation pressure [bar] at a given temperature using Antoine equation. Exponential base is e.

The following Antoine's equation is used:

:math:'ln (P /bar) = A -

rac{B}{T/K + C}‘

T [float] Absolute temperature [K]

tsat (P)

Returns vapor saturation temperature [K] at a given pressure using Antoine equation. Exponential base is e.

The following Antoine's equation is used:

:math:‘ln (P /bar) = A -

rac{B}{T/K + C}‘

P [float] Saturation pressure [bar]

vlrackett (T)

Returns liquid molar volume [$\mathrm{cm}^3/\mathrm{mol}$] at a given temperature using the Rackett equation.

$v = v_c Z_c^{(1-T_r)^{2/7}}$

Parameters **T** (*float*) – Absolute temperature [K]

7.1.2 phasepy.mixture

phasepy.mixture object stores both pure component and mixture related information and interaction parameters needed for equilibria and interfacial properties computation. Two pure components are required to create a base mixture:

```
>>> import numpy as np
>>> from phasepy import component, mixture
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(ethanol, water)
```

Additional components can be added to the mixture with *phasepy.mixture.add_component()*.

```
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
                     Ant=[9.16238246, 2541.97883529, -50.40534341],
                     GC={'CH3':3, 'CH3O':1, 'C':1})
>>> mix.add_component(mtbe)
```

Once all components have been added to the mixture, the interaction parameters must be supplied using a function depending on which model will be used:

For quadratic mixing rule (QMR) used in cubic EoS:

```
>>> kij = np.array([[0, k12, k13],
                    [k21, 0, k23],
                    [k31, k32, 0]])
>>> mix.kij_cubic(kij)
```

For NRTL model:

```
>>> alpha = np.array([[0, alpha12, alpha13],
                      [alpha21, 0, alpha23],
                      [alpha31, alpha32, 0]])
>>> g = np.array([[0, g12, g13],
                  [g21, 0, g23],
                  [g31, g32, 0]])
>>> g1 = np.array([[0, gT12, gT13],
                   [gT21, 0, gT23],
                   [gT31, gT32, 0]])
>>> mix.NRTL(alpha, g, g1)
```

For Wilson model:

```
>>> A = np.array([[0, A12, A13],
                  [A21, 0, A23],
                  [A31, A32, 0]])
>>> mix.wilson(A)
```

For Redlich Kister parameters are set by polynomial by pairs, the order of the pairs must be the following: 1-2, 1-3, . . . , 1-n, 2-3, . . . , 2-n, etc.

```
>>> C0 = np.array([poly12], [poly13], [poly23]]
>>> C1 = np.array([polyT12], [polyT13], [polyT23]]
>>> mix.rk(C0, C1)
```

For Modified-UNIFAC model, Dortmund public database must be read in:

```
>>> mix.unifac()
```

Warning: User is required to supply the necessary parameters for methods

class mixture(*component1*, *component2*)

Object class for info about a mixture.

Parameters

- **component1** (component) – First mixture component object
- **component2** (component) – Second mixture component object

name

Names of the components

Type List[str]

Tc

Critical temperatures [K]

Type List[float]

Pc

Critical pressures [bar]

Type List[float]

Zc

critical compressibility factors

Type List[float]

Vc
Critical molar volumes [$\mathrm{cm}^3/\mathrm{mol}$]

Type List[float]

w
Acentric factors

Type List[float]

c
Volume translation parameter used in cubic EoS [$\mathrm{cm}^3/\mathrm{mol}$]

Type List[float]

cii
Polynomial coefficients for influence parameter used in SGT model

Type List[list]

ksv
Parameters for alpha for PRSV EoS, if fitted

Type List[list]

Ant
Antoine correlation parameters

Type List[list]

GC
Group contribution information used in Modified-UNIFAC activity coefficient model. Group definitions can be found here.

Type List[dict]

Mw
molar weights of the fluids in the mixture [g/mol]

Type list[dict]

qi
Component molecular surface used in UNIQUAC model

Type list[dict]

ri
Component molecular volume used in UNIQUAC model

Type list[dict]

alpha_params
parameters for custom cubic alpha functions

Type list[Any]

dHf
Enthalpy of fusion [J/mol]

Type list[float]

Tf
Temperature of fusion [K]

Type list[float]

NRTL (*alpha*, *g*, *g1=None*)
Adds NRTL parameters to the mixture.

Parameters

- **alpha** (*array*) – Aleatory factor
- **g** (*array*) – Matrix of energy interactions [K]
- **g1** (*array, optional*) – Matrix of energy interactions [1/K]

Note: Parameters are evaluated as a function of temperature: $\tau = g/T + g_1$

add_component (*component*)
Adds a component to the mixture

ci (*T*)
Returns the matrix of cij interaction parameters for SGT model at a given temperature.

Parameters **T** (*float*) – Absolute temperature [K]

copy ()
Returns a copy of the mixture object

kij_cubic (*kij*, *balanced=True*)
Adds kij matrix coefficients for QMR mixing rule to the mixture. Matrix must be symmetrical and the main diagonal must be zero.

Parameters

- **kij** (*array_like*) – Matrix of interaction parameters
- **balanced** (*bool, optional*) – If True, the kij matrix must be symmetrical

kij_saft (*kij*)

Adds kij binary interaction matrix for SAFT-VR-Mie to the mixture. Matrix must be symmetrical and the main diagonal must be zero.

$$\epsilon_{ij} = (1 - k_{ij})$$

rac{sqrt{sigma_i^3 sigma_j^3}}{sigma_{ij}^3} sqrt{epsilon_i epsilon_j}

kij: array_like Matrix of interaction parameters

kij_ws (*kij*)
Adds kij matrix coefficients for WS mixing rule to the mixture. Matrix must be symmetrical and the main diagonal must be zero.

Parameters **kij** (*array_like*) – Matrix of interaction parameters

original_unifac ()
Reads database for Original-UNIFAC model to the mixture for calculation of activity coefficients.

Group definitions can be found here.

psat (*T*)

Returns array of vapour saturation pressures [bar] at a given temperature using Antoine equation. Exponential base is e.

The following Antoine's equation is used:

:math:‘ln (P /bar) = A -

rac{B}{T/K + C}‘

T [float] Absolute temperature [K]

Psat [array_like] saturation pressure of each component [bar]

rk(*c*, *c1=None*)

Adds Redlich Kister polynomial coefficients for excess Gibbs energy to the mixture.

Parameters

- **c** (*array*) – Polynomial values [Adim]
- **c1** (*array, optional*) – Polynomial values [K]

Note: Parameters are evaluated as a function of temperature: $G = c + c_1/T$

rkb(*c*, *c1=None*)

Adds binary Redlich Kister polynomial coefficients for excess Gibbs energy to the mixture.

Parameters

- **c** (*array*) – Polynomial values [Adim]
- **c1** (*array, optional*) – Polynomial values [K]

Note: Parameters are evaluated as a function of temperature: $G = c + c_1/T$

rkt(*D*)

Adds a ternary polynomial modification for NRTL model to the mixture.

Parameters **D** (*array*) – Ternary interaction parameter values

tsat(*P*)

Returns array of vapour saturation temperatures [K] at a given pressure using Antoine equation. Exponential base is e.

The following Antoine's equation is used:

:math:‘ln (P /bar) = A -

rac{B}{T/K + C}‘

Psat [float] Saturation pressure [bar]

Tsat [array_like] saturation temperature of each component [K]

unifac()

Reads the Dortmund database for Modified-UNIFAC model to the mixture for calculation of activity coefficients.

Group definitions can be found ‘here

<http://www.ddbst.com/PublishedParametersUNIFACDO.html#ListOfMainGroups>‘_.

uniquac(*a0*, *a1=None*)

Adds UNIQUAC interaction energies to the mixture.

Parameters

- **a0** (*array*) – Matrix of energy interactions [K]
- **a1** (*array, optional*) – Matrix of energy interactions [Adim.]

Note: Parameters are evaluated as a function of temperature: $a_{ij} = a_0 + a_1 T$

vlrackett(*T*)

Returns array of liquid molar volumes [$\mathrm{cm}^3/\mathrm{mol}$] at a given temperature using the Rackett equation.

$v = v_c Z_c^{(1-T_r)^{2/7}}$

Parameters **T** (*float*) – Absolute temperature [K]

Returns **vl** – liquid volume of each component [cm3 mol-1]

Return type array_like

wilson(*A*)

Adds Wilson model coefficients to the mixture. Argument matrix main diagonal must be zero.

Parameters **A** (*array*) – Interaction parameter values [K]

With the class component *phasepy.component*, only pure component info is saved. Info includes critical temperature, pressure, volume, acentric factor, Antoine coefficients and group contribution info. The class *phasepy.mixture* saves pure component data, but also interactions parameters for the activity coeffient models.

```
>>> from phasepy import component
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> water.psat(T=373.0) # vapor saturation pressure [bar]
1.0072796747419537
>>> ethanol.vlrackett(T=310.0) # liquid molar volume [cm3/mol]
56.32856995891473
```

A mixture can be created from two components:

```
>>> from phasepy import mixture
>>> mix = mixture(ethanol, water)
>>> mix.names
['ethanol', 'water']
>>> mix.nc # number of components
2
>>> mix.psat(T=373.0) # vapor saturation pressures [bar]
array([2.23333531, 1.00727967])
>>> mix.vlrackett(T=310.0) # liquid molar volumes [cm3/mol]
array([56.32856996, 16.46025809])
```

Phasepy includes two phase equilibrium models:

1. A discontinous ($\gamma - \phi$) Virial - Activity Coefficient Method model where the vapor and liquid deviations are modeled with an virial expansion and an activity coefficient model, respectively, or
2. A continuous ($\phi - \phi$) Cubic Equation of State model, using the same equation of state for all the phases.

7.1.3 Virial - Activity coefficient Model

This phase equilibrium model class uses a virial correlation for the vapor phase and an activity coefficient model for the liquid phase. For the gas phase, the virial coefficient can be estimated using ideal gas law, Abbott or Tsonopoulos correlations. For liquid phase, NRTL, Wilson, Redlich-Kister and Modified-UNIFAC activity coefficient models are available.

Virial EoS

All virial models use the same argument list:

T [float] absolute temperature [K]

Tij: array square matrix of critical temperatures [K]

Pij: array square matrix of critical pressures [bar]

wij: array square matrix of acentric factors

`Tsonopoulos` (*T*, *Tij*, *Pij*, *wij*)

Returns array of virial coefficient for a mixture at given temperature with Tsonopoulos correlation for the first virial coefficient, *B*:

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

Where $B^{(0)}$ and $B^{(1)}$ are obtained from:

$$B^{(0)} = 0.1445 - \frac{0.33}{T_r} - \frac{0.1385}{T_r^2} - \frac{0.0121}{T_r^3} - \frac{0.000607}{T_r^8}$$
$$B^{(1)} = 0.0637 + \frac{0.331}{T_r^2} - \frac{0.423}{T_r^3} - \frac{0.008}{T_r^8}$$

`Abbott` (*T*, *Tij*, *Pij*, *wij*)

Returns array of virial coefficients for a mixture at given temperature with Abbott-Van Ness correlation for the first virial coefficient, *B*:

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

Where $B^{(0)}$ and $B^{(1)}$ are obtained from:

$$B^{(0)} = 0.083 - \frac{0.422}{T_r^{1.6}}$$
$$B^{(1)} = 0.139 + \frac{0.179}{T_r^{4.2}}$$

`ideal_gas` (*T*, *Tij*, *Pij*, *wij*)

Returns array of ideal virial coefficients (zeros). The model equation is

$$Z = \frac{Pv}{RT} = 1$$

Note: Ideal gas model uses only the shape of Tij to produce zeros.

Activity coefficient models

nrtl (*X*, *T*, *alpha*, *g*, *g1*)

The non-random two-liquid (NRTL) activity coefficient model is a a local composition model, widely used to describe vapor-liquid, liquid-liquid and vapor-liquid-liquid equilibria. This function returns array of natural logarithm of the activity coefficients.

$$g^e = \sum_{i=1}^{c} x_i \frac{\sum_{j=1}^{c} \tau_{ji} G_{ji} x_j}{\sum_{l=1}^{c} G_{li} x_l}$$

$$\tau = g/T + g_1$$

Parameters

- **X** (*array*) – Molar fractions
- **T** (*float*) – Absolute temperature [K]
- **g** (*array*) – Matrix of energy interactions [K]
- **g1** (*array*) – Matrix of energy interactions [1/K]
- **alpha** (*array*) – Matrix of aleatory factors

wilson (*X*, *T*, *A*, *vl*)

Wilson activity coefficient model is a local composition model recommended for vapor-liquid equilibria calculation. It can't predict liquid liquid equilibrium. Function returns array of natural logarithm of activity coefficients.

$$g^e = \sum_{i=1}^{c} x_i \ln(\sum_{j=1}^{c} x_j \Lambda_{ij})$$

Parameters

- **X** (*array*) – Molar fractions
- **T** (*float*) – Absolute temperature [K]
- **A** (*array like*) – Matrix of energy interactions [K]
- **vl** (*function*) – Returns liquid volume of species $[\mathrm{cm}^3/\mathrm{mol}]$ given temperature [K] as argument.

rk (*x*, *T*, *C*, *C1*, *combinatory*)

Redlich-Kister activity coefficient model for multicomponent mixtures. This method uses a polynomial fit of Gibbs excess energy. It is not recommended to use more than 5 terms of the polynomial expansion. Function returns array of natural logarithm of activity coefficients.

$$g^e_{ij} = x_i x_j \sum_{k=0}^{m} C_k (x_i - x_j)^k$$

$$G = C + C_1/T$$

Parameters

- **X** (*array*) – Molar fractions
- **T** (*float*) – Absolute temperature [K]
- **C** (*array*) – Polynomial coefficient values adim
- **C1** (*array*) – Polynomial coefficient values [K]

unifac(*x*, *T*, *qi*, *ri*, *ri34*, *Vk*, *Qk*, *tethai*, *a0*, *a1*, *a2*)

Dortmund Modified-UNIFAC activity coefficient model for multicomponent mixtures is a group contribution method, which uses group definitions and parameter values from Dortmund public database. Function returns array of natural logarithm of activity coefficients.

$$\ln \gamma_i = \ln \gamma_i^{comb} + \ln \gamma_i^{res}$$

Energy interaction equation is

$$a_{mn} = a_0 + a_1 T + a_2 T^2$$

Parameters

- **X** (*array*) – Molar fractions
- **T** (*float*) – Absolute temperature [K]
- **qi** (*array*) – Component surface array
- **ri** (*array*) – Component volumes array
- **ri34** (*array*) – Component volume array, exponent 3/4
- **Vk** (*array*) – Group volumes
- **Qk** (*array*) – Group surface array
- **tethai** (*array*) – Surface fractions
- **a0** (*array*) – Energy interactions polynomial coefficients
- **a1** (*array*) – Energy interactions polynomial coefficients
- **a2** (*array*) – Energy interactions polynomial coefficients

Before creating a phase equilibrium model object, it is necessary to supply the interactions parameters of the activity coefficient model to the mixture object, for example using the NRTL model:

```
>>> import numpy as np
>>> from phasepy import component, mixture, virialgamma
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(ethanol, water)
>>> alpha = np.array([[0.0, 0.5597628],
                      [0.5597628, 0.0]])
>>> g = np.array([[0.0, -57.6880881],
                  [668.682368, 0.0]])
>>> g1 = np.array([[0.0, 0.46909821],
                   [-0.37982045, 0.0]])
>>> mix.NRTL(alpha, g, g1)
>>> model = virialgamma(mix, virialmodel='Abbott', actmodel='nrtl')
```

Parameters for Redlich-Kister are specified as follows:

```
>>> C0 = np.array([1.20945699, -0.62209997, 3.18919339])
>>> C1 = np.array([-13.271128, 101.837857, -1100.29221])
>>> mix.rk(C0, C1)
>>> model = virialgamma(mix, actmodel='rk')
```

Modified-UNIFAC with an ideal gas model is set up simply:

```
>>> mix.unifac()
>>> model = virialgamma(mix, virialmodel='ideal_gas', actmodel='unifac')
```

class virialgamma (*mix*, *virialmodel='Tsonopoulos'*, *actmodel='nrtl'*)

Bases: `object`

Returns a phase equilibrium model with mixture using a virial EOS to describe vapour phase, and an activity coefficient model for liquid phase.

Parameters

- **mix** (*object*) – mixture created with mixture class
- **virialmodel** (*string*) – function to compute virial coefficients, available options are 'Tsonopoulos', 'Abbott' or 'ideal_gas'
- **actmodel** (*string*) – function to compute activity coefficients, available optiones are 'nrtl', 'wilson', 'original_unifac', 'unifac', 'uniquac', 'rkb' or 'rk'

temperature_aux: computes temperature dependent parameters.

logfugef: computes effective fugacity coefficients.

dlogfugef: computes effective fugacity coefficients and its composition derivatives.

lngama: computes activity coefficients.

dlngama: computes activity coefficients and its composition derivatives.

dlngama (*X*, *T*)

Computes the natural logarithm of activy coefficients and its composition derivatives matrix.

Parameters

- **X** (*array*) – molar fractions
- **T** (*float*) – absolute temperature [K]

Returns

- **lngama** (*array_like*) – activity coefficient
- **dlngama** (*array_like*) – derivatives of the activity coefficients

dlogfugef (*X*, *T*, *P*, *state*, *v0=None*)

Returns array of effective fugacity coefficients at given composition, temperature and pressure as first return value, array of partial fugacity coefficients as second return value, and passes through argument v0 as third value.

Parameters

- **X** (*array*) – molar fractions
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]

- **state** (*string*) – 'L' for liquid phase, or 'V' for vapour phase
- **v0** (*float, optional*) – volume of phase

Returns

- **logfug** (*array_like*) – effective fugacity coefficients
- **dlogfug** (*array_like*) – derivatives of effective fugacity coefficients
- **v0** (*float*) – volume of phase, if calculated

lngama (*X*, *T*)

Computes the natural logarithm of activy coefficients.

Parameters

- **X** (*array*) – molar fractions
- **T** (*float*) – absolute temperature [K]

Returns **lngama** – activity coefficients

Return type array_like

logfugef (*X*, *T*, *P*, *state*, *v0=None*)

Returns array of effective fugacity coefficients at given composition, temperature and pressure as first return value, and passes through argument v0 as second value.

Parameters

- **X** (*array*) – molar fractions
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase, or 'V' for vapour phase
- **v0** (*float, optional*) – volume of phase

Returns

- **logfug** (*array_like*) – effective fugacity coefficients
- **v0** (*float*) – volume of phase, if calculated

7.1.4 Cubic Equation of State Model

This phase equilibrium model class applies equation of state (EoS) model for both vapor and liquid phases. EoS formulation is explicit:

$$P = \frac{RT}{v-b} - \frac{a}{(v+c_1 b)(v+c_2 b)}$$

Phasepy includes following cubic EoS:

- Van der Waals (VdW)
- Peng Robinson (PR)
- Redlich Kwong (RK)
- Redlich Kwong Soave (RKS), a.k.a Soave Redlich Kwong (SRK)
- Peng Robinson Stryjek Vera (PRSV)

Both pure component EoS and mixture EoS are supported.

Pure Component EoS

Pure component example using Peng-Robinson EoS:

```
>>> from phasepy import preos, component
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> eos = preos(ethanol)
>>> eos.psat(T=350.0) # saturation pressure, liquid volume and vapor volume
(array([0.98800647]), array([66.75754804]), array([28799.31921623]))
```

Density can be computed given the aggregation state (`L` for liquid, `V` for vapor):

```
>>> eos.density(T=350.0, P=1.0, state='L')
0.01497960198094922
>>> eos.density(T=350.0, P=1.0, state='V')
3.515440899573752e-05
```

Volume Translation

Volume translated (VT) versions of EoS are available for PR, RK, RKS and PRSV models. These models include an additional component specific volume translation parameter `c`, which can be used to improve liquid density predictions without changing phase equilibrium. EoS property `volume_translation` must be `True` to enable VT.

```
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1},
                        c=5.35490936)
>>> eos = preos(ethanol, volume_translation=True)
>>> eos.psat(T=350.0) # saturation pressure, liquid volume and vapor volume
(array([0.98800647]), array([61.40263868]), array([28793.96430687]))
>>> eos.density(T=350.0, P=1.0, state='L')
0.01628597159790686
>>> eos.density(T=350.0, P=1.0, state='V')
3.5161028012629526e-05
```

Mixture EoS

Mixture EoS utilize one-fluid mixing rules, using parameters for hypothetical pure fluids, to predict the mixture behavior. The mixing rules require interaction parameter values as input (zero values are assumed if no values are specified).

Classic Quadratic Mixing Rule (QMR)

$$a_m = \sum_{i=1}^{c}\sum_{j=1}^{c} x_i x_j a_{ij} \quad a_{ij} = \sqrt{a_i a_j}(1-k_{ij}) \quad b_m = \sum_{i=1}^{c} x_i b_i$$

Example of Peng-Robinson with QMR:

```
>>> from phasepy import preos
>>> mix = mixture(ethanol, water)
>>> Kij = np.array([[0, -0.11], [-0.11, 0]])
>>> mix.kij_cubic(Kij)
>>> eos = preos(mix, mixrule='qmr')
```

Modified Huron Vidal (MHV) Mixing Rule

MHV mixing rule is specified in combination with an activity coefficient model to solve EoS. In MHV model, the repulsive (covolume) parameter is calcuated same way as in QMR

$$b_m = \sum_{i=1}^{c} x_i b_i$$

while attractive term is an implicit function

$$g^e_{EOS} = g^e_{model}$$

Example of Peng-Robinson with MHV and NRTL:

```
>>> alpha = np.array([[0.0, 0.5597628],
                      [0.5597628, 0.0]])
>>> g = np.array([[0.0, -57.6880881],
                  [668.682368, 0.0]])
>>> g1 = np.array([[0.0, 0.46909821],
                   [-0.37982045, 0.0]])
>>> mix.NRTL(alpha, g, g1)
>>> eos = preos(mix, mixrule='mhv_nrtl')
```

Example of Peng-Robinson with MHV and Modified-UNIFAC:

```
>>> mix.unifac()
>>> eos = preos(mix, mixrule='mhv_unifac')
```

Wong Sandler (WS) Mixing Rule

WS mixing rule is specified in combination with an activity coefficient model to solve EoS.

Example of Peng-Robinson with WS and Redlich Kister:

```
>>> C0 = np.array([1.20945699, -0.62209997,  3.18919339])
>>> C1 = np.array([-13.271128, 101.837857, -1100.29221])
>>> mix.rk(C0, C1)
>>> eos = preos(mix, mixrule='ws_rk')
```

cubiceos (*mix_or_component*, *c1=-0.41421356237309515*, *c2=2.414213562373095*, *alpha_eos=<function alpha_soave>*, *mixrule='qmr'*, *volume_translation=False*)

Returns cubic EoS object. with custom c1, c2, alpha function and mixing rule.

Parameters

- **mix_or_component** (*object*) – *phasepy.mixture* or *phasepy.component* object
- **c2** (*c1,*) – Constants of the cubic EoS

- **alpha_eos** (*function*) – Function that returns the alpha parameter of the cubic EoS. alpha_eos(T, alpha_params, Tc)
- **mixrule** (*str*) – Mixing rule specification. Available opitions include 'qmr', 'mhv_nrtl', 'mhv_unifac', 'mhv_rk', 'mhv_wilson', 'mhv_uniquac', 'mhv_original_unifac', 'ws_nrtl', 'ws_wilson', 'ws_rk', 'ws_unifac', 'ws_uniquac', 'ws_original_unifac, mhv1_nrtl', 'mhv1_unifac', 'mhv1_rk', 'mhv1_wilson', 'mhv1_uniquac, 'mhv1_original_unifac'
- **volume_translation** (*bool*) – If True, the volume translated version of this EoS will be used.

preos (*mix_or_component*, *mixrule='qmr'*, *volume_translation=False*)

Returns Peng Robinson EoS object.

Parameters

- **mix_or_component** (*object*) – *phasepy.mixture* or *phasepy.component* object
- **mixrule** (*str*) – Mixing rule specification. Available opitions include 'qmr', 'mhv_nrtl', 'mhv_unifac', 'mhv_rk', 'mhv_wilson', 'mhv_uniquac', 'mhv_original_unifac', 'ws_nrtl', 'ws_wilson', 'ws_rk', 'ws_unifac', 'ws_uniquac', 'ws_original_unifac, mhv1_nrtl', 'mhv1_unifac', 'mhv1_rk', 'mhv1_wilson', 'mhv1_uniquac, 'mhv1_original_unifac'
- **volume_translation** (*bool*) – If True, the volume translated version of this EoS will be used.

prsveos (*mix_or_component*, *mixrule='qmr'*, *volume_translation=False*)

Returns Peng Robinson Stryjek Vera EoS object.

Parameters

- **mix_or_component** (*object*) – *phasepy.mixture* or *phasepy.component* object
- **mixrule** (*str*) – Mixing rule specification. Available opitions include 'qmr', 'mhv_nrtl', 'mhv_unifac', 'mhv_rk', 'mhv_wilson', 'mhv_uniquac', 'mhv_original_unifac', 'ws_nrtl', 'ws_wilson', 'ws_rk', 'ws_unifac', 'ws_uniquac', 'ws_original_unifac, mhv1_nrtl', 'mhv1_unifac', 'mhv1_rk', 'mhv1_wilson', 'mhv1_uniquac, 'mhv1_original_unifac'
- **volume_translation** (*bool*) – If True, the volume translated version of this EoS will be used.

rkeos (*mix_or_component*, *mixrule='qmr'*, *volume_translation=False*)

Returns Redlich Kwong EoS object.

Parameters

- **mix_or_component** (*object*) – *phasepy.mixture* or *phasepy.component* object
- **mixrule** (*str*) – Mixing rule specification. Available opitions include 'qmr', 'mhv_nrtl', 'mhv_unifac', 'mhv_rk', 'mhv_wilson', 'mhv_uniquac', 'mhv_original_unifac', 'ws_nrtl', 'ws_wilson', 'ws_rk', 'ws_unifac', 'ws_uniquac', 'ws_original_unifac, mhv1_nrtl', 'mhv1_unifac', 'mhv1_rk', 'mhv1_wilson', 'mhv1_uniquac, 'mhv1_original_unifac'
- **volume_translation** (*bool*) – If True, the volume translated version of this EoS will be used.

rkseos (*mix_or_component*, *mixrule='qmr'*, *volume_translation=False*)

Returns Redlich Kwong Soave EoS object.

Parameters

- **mix_or_component** (*object*) – *phasepy.mixture* or *phasepy.component* object
- **mixrule** (*str*) – Mixing rule specification. Available opitions include 'qmr', 'mhv_nrtl', 'mhv_unifac', 'mhv_rk', 'mhv_wilson', 'mhv_uniquac', 'mhv_original_unifac', 'ws_nrtl', 'ws_wilson', 'ws_rk', 'ws_unifac', 'ws_uniquac', 'ws_original_unifac, mhv1_nrtl', 'mhv1_unifac', 'mhv1_rk', 'mhv1_wilson', 'mhv1_uniquac, 'mhv1_original_unifac'
- **volume_translation** (*bool*) – If True, the volume translated version of this EoS will be used.

vdweos (*mix_or_component*)

Returns Van der Waals EoS object.

Parameters **mix_or_component** (*object*) – *phasepy.mixture* or *phasepy.component* object

EoS classes

Pure Cubic Equation of State Class

class cpure (*pure*, *c1*, *c2*, *oma*, *omb*, *alpha_eos*)

Bases: `object`

Pure component Cubic EoS Object

This object have implemeted methods for phase equilibrium as for interfacial properties calculations.

Parameters

- **pure** (*object*) – pure component created with component class
- **c2** (*c1,*) – constants of cubic EoS
- **omb** (*oma,*) – constants of cubic EoS
- **alpha_eos** (*function*) – function that gives thermal funcionality to attractive term of EoS

Tc

critical temperture [K]

Type float

Pc

critical pressure [bar]

Type float

w

acentric factor

Type float

cii

influence factor for SGT polynomial [J m5 mol-2]

Type array_like

Mw

molar weight of the fluid [g mol-1]

Type float

```
a_eos : computes the attractive term of cubic eos.
psat : computes saturation pressure.
tsat : computes saturation temperature
density : computes density of mixture.
logfug : computes fugacity coefficient.
a0ad : computes dimensionless Helmholtz density energy
muad : computes dimensionless chemical potential.
dOm : computes dimensionless Thermodynamic Grand Potential.
ci :  computes influence parameters matrix for SGT.
sgt_adim : computes dimensionless factors for SGT.
EntropyR : computes residual Entropy.
EnthalpyR: computes residual Enthalpy.
CvR : computes residual isochoric heat capacity.
CpR : computes residual isobaric heat capacity.
speed_sound : computes the speed of sound.
```

CpR (*T*, *P*, *state*, *v0=None*, *T_Step=0.1*)

Cpr(T, P, state, v0, T_step)

Method that computes the residual heat capacity at given temperature and pressure.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Cp** – residual heat capacity [J/mol K]

Return type float

CvR (*T*, *P*, *state*, *v0=None*, *T_Step=0.1*)

Cpr(T, P, state, v0, T_step)

Method that computes the residual isochoric heat capacity at given temperature and pressure.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Cv** – residual isochoric heat capacity [J/mol K]

Return type float

EnthalpyR(*T*, *P*, *state*, *v0*, *T_step*)

Method that computes the residual enthalpy at given temperature and pressure.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Hr** – residual enthalpy [J/mol]

Return type float

EntropyR(*T*, *P*, *state*, *v0*, *T_step*)

Method that computes the residual entropy at given temperature and pressure.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Sr** – residual entropy [J/mol K]

Return type float

a0ad(*ro*, *T*)

Method that computes the dimensionless Helmholtz density energy at given density and temperature.

Parameters

- **ro** (*float*) – dimensionless density vector [rho = rho * b]
- **T** (*float*) – absolute dimensionless temperature [Adim]

Returns **a0ad** – dimensionless Helmholtz density energy [Adim]

Return type float

a_eos(*T*)

Method that computes atractive term of cubic eos at fixed T (in K)

Parameters **T** (*float*) – absolute temperature [K]

Returns **a** – atractive term array [bar cm6 mol-2]

Return type float

ci(*T*)

Method that evaluates the polynomial for the influence parameters used in the SGT theory for surface tension calculations.

Parameters **T** (*float*) – absolute temperature [K]

Returns **ci** – influence parameters [J m5 mol-2]

Return type float

dOm (*roa*, *T*, *mu*, *Psat*)

Method that computes the dimensionless Thermodynamic Grand potential at given density and temperature.

Parameters

- **roa** (*float*) – dimensionless density vector [rho = rho * b]
- **T** (*floar*) – absolute dimensionless temperature [Adim]
- **mu** (*float*) – dimensionless chemical potential at equilibrium [Adim]
- **Psat** (*float*) – dimensionless pressure at equilibrium [Adim]

Returns **Out** – Thermodynamic Grand potential [Adim]

Return type float

density (*T*, *P*, *state*)

Method that computes the density of the mixture at given temperature and pressure.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase

Returns **density** – molar density [mol/cm3]

Return type float

logfug (*T*, *P*, *state*)

Method that computes the fugacity coefficient at given temperature and pressure.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase

Returns

- **logfug** (*float*) – fugacity coefficient
- **v** (*float*) – volume of the fluid [cm3/mol]

muad (*ro*, *T*)

Method that computes the dimensionless chemical potential at given density and temperature.

Parameters

- **roa** (*float*) – dimensionless density vector [rho = rho * b]
- **T** (*float*) – absolute dimensionless temperature [adim]

Returns **muad** – chemical potential [Adim]

Return type float

pressure(*v*, *T*)

Method that computes the pressure at given volume (cm3/mol) and temperature T (in K)

Parameters

- **T** (*float*) – absolute temperature [K]
- **v** (*float*) – molar volume [cm3/mol]

Returns **P** – pressure [bar]

Return type float

psat(*T*, *P0*)

Method that computes saturation pressure at given temperature

Parameters

- **T** (*float*) – absolute temperature [K]
- **P0** (*float, optional*) – initial value to find saturation pressure [bar], None for automatic initiation

Returns

- **psat** (*float*) – saturation pressure [bar]
- **vl** (*float*) – saturation liquid volume [cm3/mol]
- **vv** (*float*) – saturation vapor volume [cm3/mol]

sgt_adim(*T*)

Method that evaluates dimensionless factors for temperature, pressure, density, tension and distance for interfacial properties computations with SGT.

Parameters

- **T** (*float*) –
- **temperature [K]** (*absolute*) –

Returns

- **Tfactor** (*float*) – factor to obtain dimentionless temperature (K -> adim)
- **Pfactor** (*float*) – factor to obtain dimentionless pressure (bar -> adim)
- **rofactor** (*float*) – factor to obtain dimentionless density (mol/cm3 -> adim)
- **tenfactor** (*float*) – factor to obtain dimentionless surface tension (mN/m -> adim)
- **zfactor** (*float*) – factor to obtain dimentionless distance (Amstrong -> adim)

speed_sound(*T*, *P*, *state*, *v0*, *T_step*, *CvId*, *CpId*)

Method that computes the speed of sound at given temperature and pressure.

This calculation requires that the molar weight [g/mol] of the fluid has been set in the component function.

By default the ideal gas Cv and Cp are set to 3R/2 and 5R/2, the user can supply better values if available.

Parameters

- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]

- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy
- **CvId** (*float, optional*) – Ideal gas isochoric heat capacity, set to 3R/2 by default [J/mol K]
- **CpId** (*float, optional*) – Ideal gas heat capacity, set to 3R/2 by default [J/mol K]

Returns **w** – speed of sound [m/s]

Return type float

tsat (*P*, *T0*, *Tbounds*)

Method that computes saturation temperature at given pressure

Parameters

- **P** (*float*) – pressure [bar]
- **T0** (*float, optional*) – Temperature to start iterations [K]
- **Tbounds** (*tuple, optional*) – (Tmin, Tmax) Temperature interval to start iterations [K]

Returns

- **tsat** (*float*) – saturation pressure [bar]
- **vl** (*float*) – saturation liquid volume [cm3/mol]
- **vv** (*float*) – saturation vapor volume [cm3/mol]

Mixture Cubic Equation of State Class

class cubicm (*mix*, *c1*, *c2*, *oma*, *omb*, *alpha_eos*, *mixrule*)

Bases: `object`

Mixture Cubic EoS Object

This object have implemeted methods for phase equilibrium as for interfacial properties calculations.

Parameters

- **mix** (*object*) – mixture created with mixture class
- **c2** (*c1,*) – constants of cubic EoS
- **omb** (*oma,*) – constants of cubic EoS
- **alpha_eos** (*function*) – function that gives thermal funcionality to attractive term of EoS
- **mixrule** (*function*) – computes mixture attactive and cohesive terms

Tc

critical temperture [K]

Type array_like

Pc

critical pressure [bar]

Type array_like

w

acentric factor

Type array_like

cii
influence factor for SGT polynomials [J m5 mol-2]

Type array_like

nc
number of components of mixture

Type int

Mw
molar weight of the fluids [g mol-1]

Type array_like

secondorder
True if dlogfugef and dlogfugef_aux methods are available

Type bool

secondordersgt
True if dmuad and dmuad_aux methods are available

Type bool

temperature_aux: computes temperature dependent parameters.

a_eos : computes the attractive term of cubic eos.

Zmix : computes the roots of compressibility factor polynomial.

density : computes density of mixture.

logfugef : computes effective fugacity coefficients.

logfugmix : computes mixture fugacity coeficcient.

dlogfugef: computes effective fugacity coefficients and its
composition derivatives.

a0ad : computes dimensionless Helmholtz density energy.

muad : computes dimensionless chemical potential.

dmuad : computes dimensionless chemical potential and its
composition derivatives.

dOm : computes dimensionless Thermodynamic Grand Potential.

ci : computes influence parameters matrix for SGT.

sgt_adim : computes dimensionless factors for SGT.

beta_sgt : method that incorporates the beta correction for SGT.

EntropyR : computes residual Entropy.

EnthalpyR: computes residual Enthalpy.

CvR : computes residual isochoric heat capacity.

CpR : computes residual isobaric heat capacity.

speed_sound : computes the speed of sound.

CpR (*X*, *T*, *P*, *state*, *v0=None*, *T_Step=0.1*)
Cpr(X, T, P, state, v0, T_step)

Method that computes the residual heat capacity at given composition, temperature and pressure.

Parameters

- **X** (*array_like*) – molar fraction array
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Cp** – residual heat capacity [J/mol K]

Return type float

CvR (*X*, *T*, *P*, *state*, *v0=None*, *T_Step=0.1*)
Cpr(X, T, P, state, v0, T_step)

Method that computes the residual isochoric heat capacity at given composition, temperature and pressure.

Parameters

- **X** (*array_like*) – molar fraction array
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Cv** – residual isochoric heat capacity [J/mol K]

Return type float

EnthalpyR (*X*, *T*, *P*, *state*, *v0*, *T_step*)
Method that computes the residual enthalpy at given composition, temperature and pressure.

Parameters

- **X** (*array_like*) – molar fraction array
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Hr** – residual enthalpy [J/mol]

Return type float

EntropyR(*X*, *T*, *P*, *state*, *v0*, *T_step*)

Method that computes the residual entropy at given composition, temperature and pressure.

Parameters

- **X** (*array_like*) – molar fraction array
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

Returns **Sr** – residual entropy [J/mol K]

Return type float

Zmix(*X*, *T*, *P*)

Method that computes the roots of the compressibility factor polynomial at given mole fractions (X), Temperature (T) and Pressure (P)

Parameters

- **X** (*array_like*) – mole fraction vector
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]

Returns **Z** – roots of Z polynomial

Return type array_like

a0ad(*roa*, *T*)

Method that computes the dimensionless Helmholtz density energy at given density and temperature.

Parameters

- **rhoa** (*array_like*) – dimensionless density vector [rhoa = rho * b[0]]
- **T** (*float*) – absolute temperature [K]

Returns **a0ad** – dimensionless Helmholtz density energy

Return type float

a_eos(*T*)

Method that computes attractive term of cubic eos at fixed T (in K)

Parameters **T** (*float*) – absolute temperature [K]

Returns **a** – attractive term array [bar cm6 mol-2]

Return type array_like

beta_sgt(*beta*)

Method that allow asigning the beta correction for the influence parameter in Square Gradient Theory.

Parameters **beta** (*array_like*) – beta corrections for influence parameter

ci(*T*)

Method that evaluates the polynomials for the influence parameters used in the SGT theory for surface tension calculations.

Parameters **T** (*float*) – absolute temperature [K]

Returns **cij** – matrix of influence parameters with geometric mixing rule [J m5 mol-2]

Return type array_like

dOm(*roa*, *T*, *mu*, *Psat*)

Method that computes the dimensionless Thermodynamic Grand potential at given density and temperature.

Parameters

- **rhoa** (*array_like*) – dimensionless density vector [rhoa = rho * b[0]]
- **T** (*float*) – absolute temperature [K]
- **mu** (*array_like*) – dimensionless chemical potential at equilibrium [adim]
- **Psat** (*float*) – dimensionless pressure at equilibrium [adim]

Returns **dom** – Thermodynamic Grand potential [Adim]

Return type float

density(*X*, *T*, *P*, *state*, *rho0=None*)

Method that computes the molar concentration (molar density) of the mixture at given composition (X), temperature (T) and pressure (P)

Parameters

- **X** (*array_like*) – mole fraction vector
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapor phase

Returns **density** – Molar concentration of the mixture [mol/cm3]

Return type float

dlogfugef(*X*, *T*, *P*, *state*)

Method that computes the effective fugacity coefficients and its composition derivatives at given composition, temperature and pressure.

Parameters

- **X** (*array_like*) – mole fraction vector
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase, 'V' for vapour phase
- **v0** (*float, optional*) – initial volume to iterate [cm3/mol]

Returns

- **logfug** (*array_like*) – effective fugacity coefficients
- **dlogfug** (*array_like*) – composition derivatives of effective fugacity coefficients
- **v** (*float*) – volume of phase [cm3/mol]

dmuad(*rhoa*, *T*)

muad(roa, T)

Method that computes the dimensionless chemical potential and its derivatives at given density and temperature.

Parameters

- **rhoa** (*array_like*) – dimensionless density vector [rhoa = rho * b[0]]
- **T** (*float*) – absolute temperature [K]

Returns

- **muad** (*array_like*) – dimensionless chemical potential vector
- **dmuad** (*array_like*) – dimensionless derivatives of chemical potential vector

logfugef(*X*, *T*, *P*, *state*)

Method that computes the effective fugacity coefficients at given composition, temperature and pressure.

Parameters

- **X** (*array_like,*) – mole fraction vector
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase, 'V' for vapour phase
- **v0** (*float, optional*) – initial volume to iterate [cm3/mol]

Returns

- **logfug** (*array_like*) – effective fugacity coefficients
- **v** (*float*) – volume of the mixture [cm3/mol]

logfugmix(*X*, *T*, *P*, *state*)

Method that computes the mixture fugacity coefficient at given composition, temperature and pressure.

Parameters

- **X** (*array_like*) – mole fraction vector
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase

Returns

- **lofgfug** (*array_like*) – effective fugacity coefficients
- **v** (*float*) – volume of phase [cm3/mol]

muad(*roa*, *T*)

Method that computes the dimensionless chemical potential at given density and temperature.

Parameters

- **rhoa** (*array_like*) – dimensionless density vector [rhoa = rho * b[0]]
- **T** (*float*) – absolute temperature [K]

Returns **muad** – dimensionless chemical potential vector

Return type array_like

muad_aux(*rhoa*, *temp_aux*)

muad(roa, T)

Method that computes the dimensionless chemical potential at given density and temperature.

Parameters

- **rhoa** (*array_like*) – dimensionless density vector [rhoa = rho * b[0]]
- **T** (*float*) – absolute temperature [K]

Returns **muad** – dimensionless chemical potential vector

Return type array_like

pressure(*X*, *v*, *T*)

Method that computes the pressure at given composition X, volume (cm3/mol) and temperature T (in K)

Parameters

- **X** (*array_like*) – mole fraction vector
- **v** (*float*) – molar volume in [cm3/mol]
- **T** (*float*) – absolute temperature [K]

Returns **P** – pressure [bar]

Return type float

sgt_adim(*T*)

Method that evaluates dimensionless factors for temperature, pressure, density, tension and distance for interfacial properties computations with SGT.

Parameters **T** (*absolute temperature [K]*) –

Returns

- **Tfactor** (*float*) – factor to obtain dimentionless temperature (K -> adim)
- **Pfactor** (*float*) – factor to obtain dimentionless pressure (bar -> adim)
- **rofactor** (*float*) – factor to obtain dimentionless density (mol/cm3 -> adim)
- **tenfactor** (*float*) – factor to obtain dimentionless surface tension (mN/m -> adim)
- **zfactor** (*float*) – factor to obtain dimentionless distance (Amstrong -> adim)

speed_sound(*X*, *T*, *P*, *state*, *v0*, *T_step*, *CvId*, *CpId*)

Method that computes the speed of sound at given temperature and pressure.

This calculation requires that the molar weight [g/mol] of the fluid has been set in the component function.

By default the ideal gas Cv and Cp are set to 3R/2 and 5R/2, the user can supply better values if available.

Parameters

- **X** (*array_like*) – molar fraction array
- **T** (*float*) – absolute temperature [K]
- **P** (*float*) – pressure [bar]
- **state** (*string*) – 'L' for liquid phase and 'V' for vapour phase
- **v0** (*float, optional*) – initial guess for volume root [cm3/mol]
- **T_step** (*float, optional*) – Step to compute the numerical temperature derivates of Helmholtz free energy

- **CvId** (*float, optional*) – Ideal gas isochoric heat capacity, set to 3R/2 by default [J/mol K]
- **CpId** (*float, optional*) – Ideal gas heat capacity, set to 3R/2 by default [J/mol K]

Returns **w** – speed of sound [m/s]

Return type float

The coded models were tested to pass molar partial property test and Gibbs-Duhem consistency, in the case of activity coefficient model the following equations were tested:

$$\frac{G^E}{RT} - \sum_{i=1}^{c} x_i \ln \gamma_i = 0$$
$$\sum_{i=1}^{c} x_i d \ln \gamma_i = 0$$

where, G^E refers to the Gibbs excess energy, R is the ideal gas constant, T is the absolute temperature, and x_i and γ_i are the mole fraction and activity coefficient of component i. And in the case of cubic EoS:

$$\ln \phi - \sum_{i=1}^{c} x_i \ln \hat{\phi}_i = 0$$
$$\frac{d \ln \phi}{dP} - \frac{Z-1}{P} = 0$$
$$\sum_{i=1}^{c} x_i d \ln \hat{\phi}_i = 0$$

Here, ϕ is the fugacity coefficient of the mixture, x_i and $\hat{\phi}_i$ are the mole fraction and fugacity coefficient of component i, P refers to pressure and Z to the compressibility factor.

7.2 phasepy.equilibrium

Phase equilibrium conditions are obtained from a differential entropy balance of a system. The following equationes must be solved:

$$T^\alpha = T^\beta = ... = T^\pi$$
$$P^\alpha = P^\beta = ... = P^\pi$$
$$\mu_i^\alpha = \mu_i^\beta = ... = \mu_i^\pi \quad i = 1, ..., c$$

Where T, P and μ are the temperature, pressure and chemical potencial, α, β and π are the phases and i is component index.

For the continuous ($\phi - \phi$) phase equilibrium approach, equilibrium is defined using fugacity coefficients ϕ:

$$x_i^\alpha \hat{\phi}_i^\alpha = x_i^\beta \hat{\phi}_i^\beta = ... = x_i^\pi \hat{\phi}_i^\pi \quad i = 1, ..., c$$

For the discontinuous ($\gamma - \phi$) phase equilibrium approach, the equilibrium is defined with vapor fugacity coefficient and liquid phase activity coefficients γ:

$$x_i^\alpha \hat{\gamma}_i^\alpha f_i^0 = x_i^\beta \hat{\gamma}_i^\beta f_i^0 = ... = x_i^\pi \hat{\phi}_i^\pi P \quad i = 1, ..., c$$

where f_i^0 is standard state fugacity.

7.2.1 Phase Stability Analysis

Stability tests give a relative estimate of how stable a phase (given temperature, pressure and composition) is thermodynamically. A stable phase decreases the Gibbs free energy of the system compared to a reference phase (e.g. the overall composition of a mixture). To evaluate the relative stability, phasepy applies the Tangent Plane Distance (TPD) function introduced by Michelsen.

$$F_{TPD}(w) = \sum_{i=1}^{c} w_i \left[\ln w_i + \ln \hat{\phi}_i(w) - \ln z_i - \ln \hat{\phi}_i(z)\right]$$

First, the TPD function is minimized locally w.r.t. phase composition w, starting from an initial composition. If the TPD value at the minimum is negative, it implies that the phase is less stable than the reference. A positive value means more stable than the reference.

The function `tpd_min()` can be applied to find a phase composition from given initial values, and the TPD value of the minimized result. Negative TPD value for the first example (trial phase is liquid) means that the resulting liquid phase composition is unstable. Positive TPD value for the second example (trial phase is vapor) means that the resulting vapor phase is more stable than the reference phase. Therefore the second solution could be used e.g. as an initial estimate for two-phase flash calculation.

```
>>> import numpy as np
>>> from phasepy import component, mixture, preos
>>> from phasepy.equilibrium import tpd_min
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
                     Ant=[9.16238246, 2541.97883529, -50.40534341],
                     GC={'CH3':3, 'CH3O':1, 'C':1})
>>> mix = mixture(water, mtbe)
>>> mix.unifac()
>>> eos = preos(mix, 'mhv_unifac')
>>> w = np.array([0.01, 0.99])
>>> z = np.array([0.5, 0.5])
>>> T = 320.0
>>> P = 1.01
>>> tpd_min(w, z, T, P, eos, stateW='L', stateZ='L') # molar fractions and TPD value
(array([0.30683438, 0.69316562]), -0.005923915138229763)
>>> tpd_min(w, z, T, P, eos, stateW='V', stateZ='L') # molar fractions and TPD value
(array([0.16434188, 0.83565812]), 0.24576563932356765)
```

tpd_min(*W*, *Z*, *T*, *P*, *model*, *stateW*, *stateZ*, *vw=None*, *vz=None*)

Minimizes the Tangent Plane Distance (TPD) function for trial phase and calculates TPD value for the minimum.

Parameters

- **W** (*array*) – Initial molar fractions of the trial phase
- **Z** (*array*) – Molar fractions of the reference phase
- **T** (*float*) – Absolute temperature [K]
- **P** (*float*) – Absolute pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **stateW** (*string*) – Trial phase type. 'L' for liquid phase, 'V' for vapor phase
- **stateZ** (*string*) – Reference phase type. 'L' for liquid phase, 'V' for vapor phase

- **vz** (*vw,*) – Phase molar volume used as initial value to compute fugacities

Returns

- **W** (*array*) – Minimized phase molar fractions
- **f** (*float*) – Minimized TPD distance value

Phasepy function `tpd_minimas()` can be used to try to find several TPD minima. The function uses random initial compositions to search for minima, so it can find the same minimum multiple times.

```
>>> from phasepy.equilibrium import tpd_minimas
>>> nmin = 3
>>> tpd_minimas(nmin, z, T, P, eos, 'L', 'L') # minima and TPD values (two unstable␣
↪minima)
(array([0.99538258, 0.00461742]), array([0.30683414, 0.69316586]), array([0.99538258,␣
↪0.00461742])),
array([-0.33722905, -0.00592392, -0.33722905])
>>> tpd_minimas(nmin, z, T, P, eos, 'V', 'L') # minima and TPD values (one stable␣
↪minimum)
(array([0.1643422, 0.8356578]), array([0.1643422, 0.8356578]), array([0.1643422, 0.
↪8356578])),
array([0.24576564, 0.24576564, 0.24576564])
```

tpd_minimas (*nmin, Z, T, P, model, stateW, stateZ, vw=None, vz=None*)

Repetition of Tangent Plane Distance (TPD) function minimization with random initial values to try to find several minima.

Parameters

- **nmin** (*int*) – Number of randomized minimizations to carry out
- **Z** (*array*) – Molar fractions of the reference phase
- **T** (*float*) – Absolute temperature [K]
- **P** (*float*) – Absolute pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **stateW** (*string*) – Trial phase type. 'L' for liquid phase, 'V' for vapor phase
- **stateZ** (*string*) – Reference phase type. 'L' for liquid phase, 'V' for vapor phase
- **vz** (*vw,*) – Phase molar volume used as initial value to compute fugacities

Returns

- **W_minima** (*tuple(array)*) – Minimized phase molar fractions
- **f_minima** (*array*) – Minimized TPD distance values

Function `lle_init()` can be used to generate two phase compositions e.g. for subsequent liquid liquid equilibrium calculations. Note that the results are not guaranteed to be stable or even different.

```
>>> from phasepy.equilibrium import lle_init
>>> lle_init(z, T, P, eos)
array([0.99538258, 0.00461742]), array([0.30683414, 0.69316586])
```

lle_init (*Z, T, P, model, vw=None, vz=None*)

Carry out two repetitions of Tangent Plane Distance (TPD) function minimization with random initial values to find two liquid phase compositions.

Parameters

- **Z** (*array*) – Molar fractions of the reference phase
- **T** (*float*) – Absolute temperature [K]
- **P** (*float*) – Absolute pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **vz** (*vw,*) – if supplied volume used as initial value to compute fugacities

Returns **W_minima** – Two minimized phase molar fractions

Return type tuple(array)

7.2.2 Bubble Point and Dew Point

Calculation of bubble point and dew point for vapor-liquid systems apply simplifications of the Rachford-Rice mass balance, as the liquid phase fraction (0% or 100%) is already known. Default solution strategy applies first Accelerated Successive Substitutions method to update phase compositions in an inner loop, and a Quasi-Newton method to update pressure or temperature in an outer loop. If convergence is not reached in 10 iterations, or if user defines initial estimates to be good, the algorithm switches to a Phase Envelope method solving the following system of equations:

$$f_i = \ln K_i + \ln \hat{\phi}_i^v(\underline{y}, T, P) - \ln \hat{\phi}_i^l(\underline{x}, T, P) \quad i = 1, ..., c$$

$$f_{c+1} = \sum_{i=1}^{c} (y_i - x_i)$$

Bubble Point

Bubble point is a fluid state where saturated liquid of known composition and liquid fraction 100% is forming a differential size bubble. The algorithm finds vapor phase composition and either temperature or pressure of the bubble point.

The algorithm solves composition using a simplified Rachford-Rice equation:

$$FO = \sum_{i=1}^{c} x_i(K_i - 1) = \sum_{i=1}^{c} y_i - 1 = 0$$

```
>>> import numpy as np
>>> from phasepy import component, mixture, rkseos
>>> from phasepy.equilibrium import bubbleTy, bubblePy
>>> butanol = component(name='butanol', Tc=563.0, Pc=44.14, Zc=0.258, Vc=274.0, w=0.
↪589462,
                        Ant=[10.20170373, 2939.88668723, -102.28265042])
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
                     Ant=[9.16238246, 2541.97883529, -50.40534341])
>>> Kij = np.zeros([2,2])
>>> mix = mixture(mtbe, butanol)
>>> mix.kij_cubic(Kij)
>>> eos = rkseos(mix, 'qmr')
>>> x = np.array([0.5, 0.5])
>>> P0, T0 = 1.0, 340.0
>>> y0 = np.array([0.8, 0.2])
>>> bubbleTy(y0, T0, x, 1.0, eos) # vapor fractions, temperature
(array([0.90411878, 0.09588122]), 343.5331023048577)
>>> bubblePy(y0, P0, x, 343.533, eos) # vapor fractions, pressure
(array([0.90411894, 0.09588106]), 0.9999969450754181)
```

bubbleTy (*y_guess, T_guess, X, P, model, good_initial=False, v0=[None, None], full_output=False*)
Bubble point (X, P) -> (Y, T).

Solves bubble point (vapor phase composition and temperature) at given pressure and liquid phase composition.

Parameters

- **y_guess** (*array*) – Initial guess of vapor phase molar fractions
- **T_guess** (*float*) – Initial guess of equilibrium temperature [K]
- **X** (*array*) – Liquid phase molar fractions
- **P** (*float*) – Pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **good_initial** (*bool, optional*) – If True uses only phase envelope method in solution
- **v0** (*list, optional*) – Liquid and vapor phase molar volume used as initial values to compute fugacities
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **Y** (*array*) – Vapor molar fractions
- **T** (*float*) – Equilibrium temperature [K]

bubblePy (*y_guess, P_guess, X, T, model, good_initial=False, v0=[None, None], full_output=False*)
Bubble point (X, T) -> (Y, P).

Solves bubble point (vapor phase composition and pressure) at given temperature and liquid composition.

Parameters

- **y_guess** (*array*) – Initial guess of vapor phase molar fractions
- **P_guess** (*float*) – Initial guess for equilibrium pressure [bar]
- **X** (*array*) – Liquid phase molar fractions
- **T** (*float*) – Absolute temperature [K]
- **model** (*object*) – Phase equilibrium model object
- **good_initial** (*bool, optional*) – If True uses only phase envelope method in solution
- **v0** (*list, optional*) – Liquid and vapor phase molar volume used as initial values to compute fugacities
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **Y** (*array*) – Vapor molar fractions
- **P** (*float*) – Equilibrium pressure [bar]

Dew Point

Dew point is a fluid state where saturated vapor of known composition and liquid fraction 0% is forming a differential size liquid droplet. The algorithm finds liquid phase composition and either temperature or pressure of the dew point.

The algorithm solves composition using a simplified Rachford-Rice equation:

$$FO = 1 - \sum_{i=1}^{c} \frac{y_i}{K_i} = 1 - \sum_{i=1}^{c} x_i = 0$$

```
>>> import numpy as np
>>> from phasepy import component, mixture, prsveos
>>> from phasepy.equilibrium import dewPx, dewTx
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
...                 ksv=[1.27092923, 0.0440421],
...                 GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
...                 ksv=[0.76429651, 0.04242646],
...                 GC={'CH3':3, 'CH3O':1, 'C':1})
>>> mix = mixture(mtbe, ethanol)
>>> C0 = np.array([0.02635196, -0.02855964, 0.01592515])
>>> C1 = np.array([312.575789, 50.1476555, 5.13981131])
>>> mix.rk(C0, C1)
>>> eos = prsveos(mix, mixrule = 'mhv_rk')
>>> y = np.array([0.5, 0.5])
>>> P0, T0 = 1.0, 340.0
>>> x0 = np.array([0.2, 0.8])
>>> dewPx(x0, P0, y, 340.0, eos) # liquid fractions, pressure
(array([0.20223477, 0.79776523]), 1.0478247383242278)
>>> dewTx(x0, T0, y, 1.047825, eos) # liquid fractions, temperature
(array([0.20223478, 0.79776522]), 340.0000061757033)
```

dewPx (*x_guess, P_guess, y, T, model, good_initial=False, v0=[None, None], full_output=False*)

Dew point (y, T) -> (x, P)

Solves dew point (liquid phase composition and pressure) at given temperature and vapor composition.

Parameters

- **x_guess** (*array*) – Initial guess of liquid phase molar fractions
- **P_guess** (*float*) – Initial guess of equilibrium pressure [bar]
- **y** (*array*) – Vapor phase molar fractions
- **T** (*float*) – Temperature [K]
- **model** (*object*) – Phase equilibrium model object
- **good_initial** (*bool, optional*) – If True uses only phase envelope method in solution
- **v0** (*list, optional*) – Liquid and vapor phase molar volume used as initial values to compute fugacities
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **x** (*array*) – Liquid molar fractions
- **P** (*float*) – Equilibrium pressure [bar]

dewTx (*x_guess, T_guess, y, P, model, good_initial=False, v0=[None, None], full_output=False*)

Dew point (y, P) -> (x, T)

Solves dew point (liquid phase composition and temperature) at given pressure and vapor phase composition.

Parameters

- **x_guess** (*array*) – Initial guess of liquid phase molar fractions
- **T_guess** (*float*) – Initial guess of equilibrium temperature [K]
- **y** (*array*) – Vapor phase molar fractions
- **P** (*float*) – Pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **good_initial** (*bool, optional*) – If True uses only phase envelope method in solution
- **v0** (*list, optional*) – Liquid and vapor phase molar volume used as initial values to compute fugacities
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **x** (*array*) – Liquid molar fractions
- **T** (*float*) – Equilibrium temperature [K]

7.2.3 Two-Phase Flash

Determination of phase composition in flash evaporation is a classical phase equilibrium problem. In the simplest case covered here, temperature, pressure and overall composition of a system are known (PT flash). If the system is thermodynamically unstable it will form two (or more) distinct phases. The flash algorithm first solves the Rachford-Rice mass balance and then updates composition by the Accelerated Successive Substitution method.

$$FO = \sum_{i=1}^{c} \left(x_i^{\beta} - x_i^{\alpha} \right) = \sum_{i=1}^{c} \frac{z_i(K_i - 1)}{1 + \psi(K_i - 1)}$$

x is molar fraction of a component in a phase, z is the overall molar fraction of component in the system, $K = x^{\beta}/x^{\alpha}$ is the equilibrium constant and ψ is the phase fraction of phase β in the system. Subscript refers to component index and superscript refers to phase index.

If convergence is not reached in three iterations, the algorithm switches to a second order minimization of the Gibbs free energy of the system:

$$min\, G(\underline{F}^{\alpha}, \underline{F}^{\beta}) = \sum_{i=1}^{c} (F_i^{\alpha} \ln \hat{f}_i^{\alpha} + F_i^{\beta} \ln \hat{f}_i^{\beta})$$

F is the molar amount of component in a phase and $\hat{f}$ is the effective fugacity.

Warning: `flash()` routine does not check for the stability of the numerical solution (see *Phase Stability Analysis*).

Vapor-Liquid Equilibrium

This example shows solution of vapor-liquid equilibrium (VLE) using the `flash()` function.

```
>>> import numpy as np
>>> from phasepy import component, mixture, preos
>>> from phasepy.equilibrium import flash
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(water, ethanol)
>>> mix.unifac()
>>> eos = preos(mix, 'mhv_unifac')
>>> T = 360.0
>>> P = 1.01
>>> Z = np.array([0.8, 0.2])
>>> x0 = np.array([0.1, 0.9])
>>> y0 = np.array([0.2, 0.8])
>>> flash(x0, y0, 'LV', Z, T, P, eos) # phase compositions, vapor phase fraction
(array([0.8979481, 0.1020519]), array([0.53414948, 0.46585052]), 0.26923713078124695)
```

flash(*x_guess, y_guess, equilibrium, Z, T, P, model, v0=[None, None], K_tol=1e-08, nacc=5, full_output=False*)

Isobaric isothermic (PT) flash: (Z, T, P) -> (x, y, beta)

Parameters

- **x_guess** (*array*) – Initial guess for molar fractions of phase 1 (liquid)
- **y_guess** (*array*) – Initial guess for molar fractions of phase 2 (gas or liquid)
- **equilibrium** (*string*) – Two-phase system definition: 'LL' (liquid-liquid) or 'LV' (liquid-vapor)
- **Z** (*array*) – Overall molar fractions of components
- **T** (*float*) – Absolute temperature [K]
- **P** (*float*) – Pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **v0** (*list, optional*) – Liquid and vapor phase molar volume used as initial values to compute fugacities
- **K_tol** (*float, optional*) – Tolerance for equilibrium constant values
- **nacc** (*int, optional*) – number of accelerated successive substitution cycles to perform
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **x** (*array*) – Phase 1 molar fractions of components
- **y** (*array*) – Phase 2 molar fractions of components
- **beta** (*float*) – Phase 2 phase fraction

Liquid-Liquid Equilibrium

For liquid-liquid equilibrium (LLE), it is important to consider stability of the phases. `lle()` function takes into account both stability and equilibrium simultaneously for the PT flash.

```
>>> import numpy as np
>>> from phasepy import component, mixture, virialgamma
>>> from phasepy.equilibrium import lle
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
                     Ant=[9.16238246, 2541.97883529, -50.40534341],
                     GC={'CH3':3, 'CH3O':1, 'C':1})
>>> mix = mixture(water, mtbe)
>>> mix.unifac()
>>> eos = virialgamma(mix, actmodel = 'unifac')
>>> T = 320.0
>>> P = 1.01
>>> Z = np.array([0.5, 0.5])
>>> x0 = np.array([0.01, 0.99])
>>> w0 = np.array([0.99, 0.01])
>>> lle(x0, w0, Z, T, P, eos) # phase compositions, phase 2 fraction
(array([0.1560131, 0.8439869]), array([0.99289324, 0.00710676]), 0.4110348438873743)
```

Liquid-liquid flash can be also solved without considering stability by using the `flash()` function, but this is not recommended.

```
>>> from phasepy.equilibrium import flash
>>> flash(x0, w0, 'LL', Z, T, P, eos) # phase compositions, phase 2 fraction
(array([0.1560003, 0.8439997]), array([0.99289323, 0.00710677]), 0.41104385845638447)
```

7.2.4 Vapor-Liquid-Liquid Equilibrium

To avoid meta-stable solutions in vapor-liquid-liquid equilibrium (VLLE) calculations, phasepy applies a modified Rachford-Rice mass balance system of equations by Gupta et al, which allows to verify the stability and equilibria of the phases simultaneously.

$$\sum_{i=1}^{c} \frac{z_i(K_{ik}\exp\theta_k - 1)}{1 + \sum\limits_{\substack{j=1 \\ j\neq r}}^{\pi} \psi_j(K_{ij}\exp\theta_j - 1)} = 0 \qquad k = 1, ..., \pi, k \neq r$$

θ is a stability variable. Positive value indicates unstable phase and zero value a stable phase. If default solution using Accelerated Successive Substitution and Newton's method does not converge fast, the algorith will switch to minimization of the Gibbs free energy of the system:

$$min\, G = \sum_{k=1}^{\pi}\sum_{i=1}^{c} F_{ik} \ln \hat{f}_{ik}$$

Multicomponent Flash

Function `vlle()` solves VLLE for mixtures with two or more components given overall molar fractions Z, temperature and pressure.

```
>>> import numpy as np
>>> from phasepy import component, mixture, virialgamma
>>> from phasepy.equilibrium import vlle
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
                     Ant=[9.16238246, 2541.97883529, -50.40534341],
                     GC={'CH3':3, 'CH3O':1, 'C':1})
>>> ethanol = component(name='ethanol', Tc=514.0, Pc=61.37, Zc=0.241, Vc=168.0, w=0.
↪643558,
                        Ant=[11.61809279, 3423.0259436, -56.48094263],
                        GC={'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(water, mtbe)
>>> mix.add_component(ethanol)
>>> mix.unifac()
>>> eos = virialgamma(mix, actmodel='unifac')
>>> x0 = np.array([0.95, 0.025, 0.025])
>>> w0 = np.array([0.4, 0.5, 0.1])
>>> y0 = np.array([0.15, 0.8, 0.05])
>>> Z = np.array([0.5, 0.44, 0.06])
>>> T = 328.5
>>> P = 1.01
>>> vlle(x0, w0, y0, Z, T, P, eos, full_output=True)
           T: 328.5
           P: 1.01
 error_outer: 3.985492841236682e-08
 error_inner: 3.4482008487377304e-10
        iter: 14
        beta: array([0.41457868, 0.22479531, 0.36062601])
       tetha: array([0., 0., 0.])
           X: array([[0.946738  , 0.01222701, 0.04103499],
       [0.23284911, 0.67121402, 0.09593687],
       [0.15295408, 0.78764474, 0.05940118]])
           v: [None, None, None]
      states: ['L', 'L', 'V']
```

vlle (*X0, W0, Y0, Z, T, P, model, v0=[None, None, None], K_tol=1e-10, nacc=5, full_output=False*)

Solves liquid-liquid-vapor equilibrium (VLLE) multicomponent flash: (Z, T, P) -> (X, W, Y)

Parameters

- **X0** (*array*) – Initial guess molar fractions of liquid phase 1
- **W0** (*array*) – Initial guess molar fractions of liquid phase 2
- **Y0** (*array*) – Initial guess molar fractions of vapor phase
- **T** (*float*) – Absolute temperature [K]
- **P** (*float*) – Pressure [bar]
- **model** (*object*) – Phase equilibrium model object
- **good_initial** (*bool, optional*) – if True skip Gupta's method and solve full system of equations
- **v0** (*list, optional*) – Liquid phase 1 and 2 and vapor phase molar volume used as initial values to compute fugacities

- **K_tol** (*float, optional*) – Tolerance for equilibrium constant values
- **nacc** (*int, optional*) – number of accelerated successive substitution cycles to perform
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **X** (*array*) – Liquid phase 1 molar fractions
- **W** (*array*) – Liquid phase 2 molar fractions
- **Y** (*array*) – Vapor phase molar fractions

Binary Mixture Composition

The function `vlleb()` can solve a special case to find component molar fractions in each of the three phases in a VLLE system containing only two components. Given either temperature or pressure, `vlleb()` solves the other, along with the component molar fractions. This system is fully defined (zero degrees of freedom) according to the Gibbs phase rule.

```
>>> import numpy as np
>>> from phasepy import component, mixture, virialgamma
>>> from phasepy.equilibrium import vlleb
>>> water = component(name='water', Tc=647.13, Pc=220.55, Zc=0.229, Vc=55.948, w=0.
↪344861,
                      Ant=[11.64785144, 3797.41566067, -46.77830444],
                      GC={'H2O':1})
>>> mtbe = component(name='mtbe', Tc=497.1, Pc=34.3, Zc=0.273, Vc=329.0, w=0.266059,
                     Ant=[9.16238246, 2541.97883529, -50.40534341],
                     GC={'CH3':3, 'CH3O':1, 'C':1})
>>> mix = mixture(water, mtbe)
>>> mix.unifac()
>>> eos = virialgamma(mix, actmodel='unifac')
>>> x0 = np.array([0.01, 0.99])
>>> w0 = np.array([0.99, 0.01])
>>> y0 = (x0 + w0)/2
>>> T0 = 320.0
>>> P = 1.01
>>> vlleb(x0, w0, y0, T0, P, 'P', eos, full_output=True)
      T: array([327.60666698])
      P: 1.01
  error: 4.142157056965187e-12
   nfev: 17
      X: array([0.17165659, 0.82834341])
     vx: None
 statex: 'Liquid'
      W: array([0.99256232, 0.00743768])
     vw: None
 statew: 'Liquid'
      Y: array([0.15177615, 0.84822385])
     vy: None
 statey: 'Vapor'
```

vlleb (*X0, W0, Y0, P_T, T_P, spec, model, v0=[None, None, None], full_output=False*)

Solves component molar fractions in each phase and either temperature or pressure in vapor-liquid-liquid equilibrium (VLLE) of binary (two component) mixture: (T or P) -> (X, W, Y, and P or T)

Parameters

- **X0** (*array*) – Initial guess molar fractions of liquid phase 1
- **W0** (*array*) – Initial guess molar fractions of liquid phase 2
- **Y0** (*array*) – Initial guess molar fractions of vapor phase
- **P_T** (*float*) – Absolute temperature [K] or pressure [bar] (see *spec*)
- **T_P** (*float*) – Absolute temperature [K] or pressure [bar] (see *spec*)
- **spec** (*string*) – 'T' if T_P is temperature or 'P' if T_P is pressure.
- **model** (*object*) – Phase equilibrium model object
- **v0** (*list, optional*) – Liquid phase 1 and 2 and vapor phase molar volume used as initial values to compute fugacities
- **full_output** (*bool, optional*) – Flag to return a dictionary of all calculation info

Returns

- **X** (*array*) – Liquid phase 1 molar fractions
- **W** (*array*) – Liquid phase 2 molar fractions
- **Y** (*array*) – Vapor phase molar fractions
- **var** (*float*) – Temperature [K] or pressure [bar], opposite of *spec*

7.3 phasepy.sgt

The Square Gradient Theory (SGT) is the reference framework when studying interfacial properties between fluid phases in equilibrium. It was originally proposed by van der Waals and then reformulated by Cahn and Hilliard. SGT proposes that the Helmholtz free energy density at the interface can be described by a homogeneous and a gradient contribution.

$$a = a_0 + \frac{1}{2}\sum_i\sum_j c_{ij}\frac{d\rho_i}{dz}\frac{d\rho_j}{dz} + \cdots$$

Here a is the Helmholtz free energy density, a_0 is the Helmholtz free energy density bulk contribution, c_{ij} is the cross influence parameter between component i and j , ρ is denisty vector and z is the lenght coordinate. The cross influence parameter is usually computed using a geometric mean rule and a correction $c_{ij} = (1-\beta_{ij})\sqrt{c_i c_j}$.

The density profiles between the bulk phases are mean to minimize the energy of the system. This results in the following Euler-Lagrange system:

$$\sum_j c_{ij}\frac{d^2\rho_j}{dz^2} = \mu_i - \mu_i^0 \qquad i = 1, ..., c$$

$$\rho(z \to -\infty) = \rho^\alpha \qquad \rho(z \to \infty) = \rho^\beta$$

Here μ represent the chemical potential and the superscript indicates its value evaluated at the bulk phase. α and β are the bulk phases index.

Once the density profiles were solved the interfacial tension, σ between the phases can be computed as:

$$\sigma = \int_{-\infty}^{\infty}\sum_i\sum_j c_{ij}\frac{d\rho_i}{dz}\frac{d\rho_j}{dz}dz$$

Solution procedure for SGT strongly depends for if you are working with a pure component or a mixture. In the latter, the correction value of β_{ij} plays a huge role in the solution procedure. These cases will be covered.

7.3.1 SGT for pure components

When working with pure components, SGT implementation is direct as there is a continuous path from the vapor to the liquid phase. SGT can be reformulated with density as indepedent variable.

$$\sigma = \sqrt{2}\int_{\rho^\alpha}^{\rho_\beta} \sqrt{c_i \Delta\Omega} d\rho$$

Here, $\Delta\Omega$ represents the grand thermodynamic potential, obtained from:

$$\Delta\Omega = a_0 - \rho\mu^0 + P^0$$

Where P^0 is the equilibrium pressure.

In phasepy this integration is done using ortoghonal collocation, which reduces the number of nodes needed for a desired error. This calculation is done with the `sgt_pure` function and it requires the equilibrium densities, temperature and pressure as inputs.

```
>>> #component creation
>>> water =  component(name = 'Water', Tc = 647.13, Pc = 220.55, Zc = 0.229, Vc = 55.
↪948, w = 0.344861
... ksv = [ 0.87185176, -0.06621339], cii = [2.06553362e-26, 2.64204784e-23, 4.
↪10320513e-21])
>>> #EoS object creation
>>> eos = prsveos(water)
```

First vapor-liquid equilibria has to be computed. This is done with the `psat` method from the EoS, which returns the pressure and densities at equilibrium. Then the interfacial can be computed as it is shown.

```
>>> T = 350 #K
>>> Psat, vl, vv = eos.psat(T)
>>> rhol = 1/vl
>>> rhov = 1/vv
>>> sgt_pure(rhov, rhol, T, Psat, eos, full_output = False)
>>> #Tension in mN/m
... 63.25083234
```

Optionally, `full_output` allows to get all the computation information as the density profile, interfacial lenght and grand thermodynamic potential.

```
>>> solution = sgt_pure(rhol, rhov, T, Psat, eos, full_output = True)
>>> solution.z #interfacial lenght array
>>> solution.rho #density array
>>> solution.tension #IFT computed value
```

7.3.2 SGT for mixtures and $\beta_{ij} = 0$

When working with mixtures, SGT solution procedure depends wether the influece parameter matrix is singular or not. The geometric mean rule leads to a singular matrix when all $\beta_{ij} = 0$. In those cases the boundary value problem (BVP) can not be solved and alternative methods has to be used. Some of the options are the reference component method, which is the most popular. For this method the following system of equations has to be solved:

$$\sqrt{c_r}\left[\mu_j(\rho) - \mu_j^0\right] = \sqrt{c_j}\left[\mu_r(\rho) - \mu_r^0\right] \qquad j \neq r$$

Where the subscript r refers to the reference component and j to the other components present in the mixture. Alought implementation of this method is direct it may not be suitable for mixtures with several stationary points in the interface. In those cases a path function is recommended, Cornellise's doctoral thesis proposed the following path function:

$$(dh)^2 = \sum_i c_i d\rho_i^2$$

Which increased monotonically from one phase to another. One of the disadvantages of this path function is that its final lenght is not known beforehand and an iterative procedure with nested for loops is needed. For some of these reasons, Liang proposed the following path function:

$$h = \sum_i \sqrt{c_i}\rho_i$$

This path function has a known value when the equilibrium densities are available. Also the solution procedure allows to formulate a auxiliar variable $\alpha = (\mu_i - \mu_i^0)/\sqrt{c_i}$. This variable gives information about whether the geometric mean rule is suitable for the mixture.

The `sgt_mix_beta0` function allows to compute interfacial tension and density profiles using SGT and $\beta_{ij} = 0$, its use is showed in the following code block for the mixture of ethanol and water:

```
>>> #component creation
>>> water =  component(name = 'Water', Tc = 647.13, Pc = 220.55, Zc = 0.229, Vc = 55.
↪948, w = 0.344861,
               ksv = [ 0.87185176, -0.06621339],
               cii = [2.06553362e-26, 2.64204784e-23, 4.10320513e-21],
               GC = {'H2O':1})
>>> ethanol = component(name = 'Ethanol', Tc = 514.0, Pc = 61.37, Zc = 0.241, Vc =␣
↪168.0, w = 0.643558,
               ksv = [1.27092923, 0.0440421 ],
               cii = [ 2.35206942e-24, -1.32498074e-21,  2.31193555e-19],
               GC = {'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(ethanol, water)
>>> mix.unifac()
>>> eos1 = prsveos(mix, 'mhv_unifac')
>>>T = 320 #K
>>> X = np.array([0.3, 0.7])
>>> P0 = 0.3 #bar
>>> Y0 = np.array([0.7, 0.3])
>>>#The full_output option allows to obtain the compositions and volume of the phases
>>> sol = bubblePy(Y0, P0, X, T, eos1, full_output = True)
>>> Y = sol.Y
>>> P = sol.P
>>> vl = sol.v1
>>> vv = sol.v2
>>> #computing the density vector
>>> rhol = X / vl
>>> rhov = Y / vv
```

As the equilibrium is already computed, the interfacial tension of the mixture can be calculated.

```
>>> #if reference component is set to ethanol (index = 0) a lower value is obtained␣
↪as the
>>> #full density profile was not calculated because of a stationary point in the␣
↪interface
>>> solr1 = sgt_mix_beta0(rhov, rhol, T, P, eos1, s = 0, n = 100,
... method = 'reference', full_output = True)
>>> #water doesnt show surface activity across the interface
```

(continues on next page)

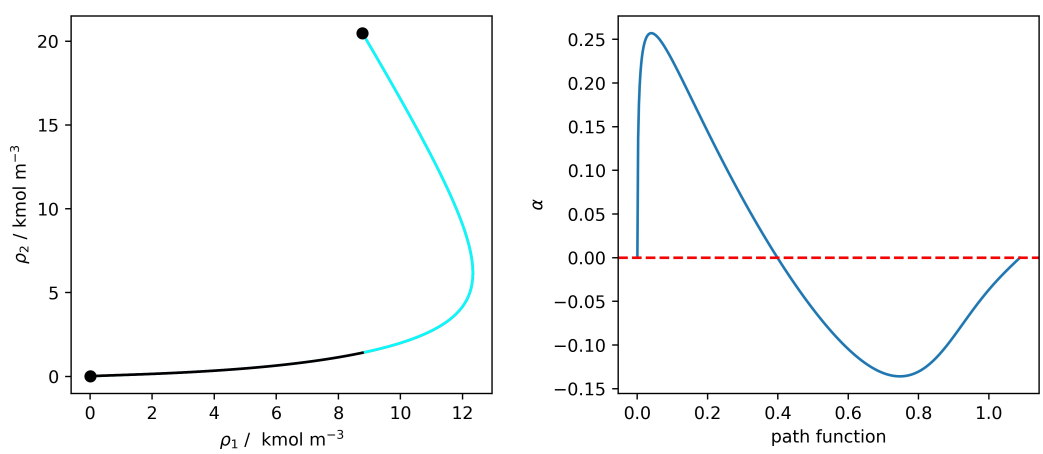
(continued from previous page)

```
>>> #and the density profiles are fully calculated
>>> solr2 = sgt_mix_beta0(rhov, rhol, T, P, eos1, s = 1, n = 100,
...  method = 'reference', full_output = True)
>>> #Using Liang path function the density profiles are computed directly
>>> soll = sgt_mix_beta0(rhov, rhol, T, P, eos1, n= 500,
...  method = 'liang', full_output = True)
>>> #Cornelisse path function also allows to compute the density profiles
>>> solc = sgt_mix_beta0(rhov, rhol, T, P, eos1, n = 500,
... method = 'cornelisse', full_output = True)
```

The following results are obtanined from each method, it can be seen that the results from path functions as reference component with component two as reference are the same.

- Reference component method (1) : 15.8164 mN/m
- Reference component method (2) : 27.2851 mN/m
- Liang path Function : 27.2845 mN/m
- Cornelisse path Function : 27.2842 mN/m

The density profiles computed from each method are plotted in the following figure. The results obtained from Cornellise's path function, Liang's path function and Reference component method with water as reference component overlaps each other, as they compute the same density profile. When selection ethanol as reference component the black line is computed. The plot from the Liang's α parameter reveal that the geometric mixing rule is suitable for the computation of the density profile of this mixture.



A more challenging mixture to analyze is ethanol and hexane. This mixture has several stationary points across the interface making its calculations tricky. Similar as before, equilibrium has to be computed.

```
>>> hexane = component(name = 'n-Hexane', Tc = 507.6, Pc = 30.25, Zc = 0.266, Vc =␣
↪371.0, w = 0.301261,
                ksv = [ 0.81185833, -0.08790848],
                cii = [ 5.03377433e-24, -3.41297789e-21,  9.97008208e-19],
                GC = {'CH3':2, 'CH2':4})
>> mix = mixture(ethanol, hexane)
>>> a12, a21 = np.array([1141.56994427,  125.25729314])
>>> A = np.array([[0, a12], [a21, 0]])
>>> mix.wilson(A)
>>> eos2 = prsveos(mix, 'mhv_wilson')
```

(continues on next page)

(continued from previous page)

```
>>> T = 320 #K
>>> X = np.array([0.3, 0.7])
>>> P0 = 0.3 #bar
>>> Y0 = np.array([0.7, 0.3])
>>> sol = bubblePy(Y0, P0, X, T, eos2, full_output = True)
>>> Y = sol.Y
>>> P = sol.P
>>> vl = sol.v1
>>> vv = sol.v2
>>> #computing the density vector
>>> rhol = X / vl
>>> rhov = Y / vv
```

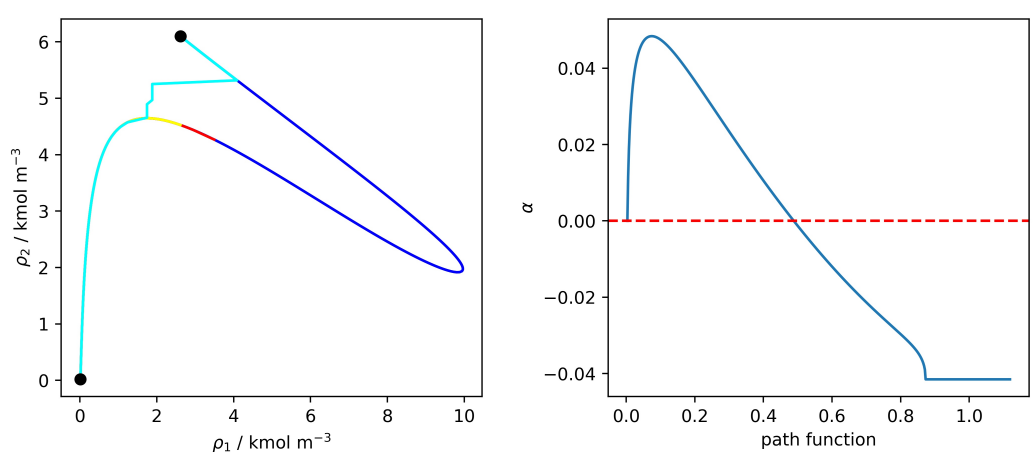
Similar as for the first example, all the posible methods will be tested.

```
>>> solr1 = sgt_mix_beta0(rhov, rhol, T, P, eos2, s = 0, n = 100,
... method = 'reference', full_output = True)
>>> solr2 = sgt_mix_beta0(rhov, rhol, T, P, eos2, s = 1, n = 100,
... method = 'reference', full_output = True)
>>> soll = sgt_mix_beta0(rhov, rhol, T, P, eos2, n= 500,
... method = 'liang', full_output = True)
>>> solc = sgt_mix_beta0(rhov, rhol, T, P, eos2, n= 500,
... method = 'cornelisse', full_output = True)
```

The interfacial tension result from each method are listed bellow.

- Reference component method (1) : 12.9207 mN/m
- Reference component method (2) : 15.5870 mN/m
- Liang path Function : 12.99849 mN/m
- Cornelisse path Function : 15.64503 mN/m

It can be seen that each method computed a different value, an inspection on the calculated density profiles can help to decide if any of the is correct. Yellow line was computed with reference component 1, Cyan line was computed with reference component 2, Liang and Cornellise path function results are plotted with a red and blue line, respectively.



Clearly, online Conrnelisse's path function was able to compute the density profiles of the mixture. An inspection on the α parameter from Liang's path function reveals that its final value is different than zero, when this method fail Liang states that geometric mean rule might no be suitable for the mixture and a β_{ij} correction should be applied.

sgt_mix_beta0(*rho1, rho2, Tsat, Psat, model, n=100, method='reference', full_output=False, s=0, alpha0=None*)

SGT for mixtures and beta = 0 (rho1, rho2, T, P) -> interfacial tension

Parameters

- **rho1** (*float*) – phase 1 density vector
- **rho2** (*float*) – phase 2 density vector
- **Tsat** (*float*) – saturation temperature
- **Psat** (*float*) – saturation pressure
- **model** (*object*) – created with an EoS
- **n** (*int, optional*) – number points to solve density profiles
- **method** (*string*) – method used to solve density profiles, available options are 'reference' for using reference component method, 'cornelisse' for Cornelisse path function and 'liang' for Liang path function
- **full_output** (*bool, optional*) – wheter to outputs all calculation info
- **s** (*int*) – index of reference component used in refernce component method
- **alpha0** (*float*) – initial guess for solving Liang path function

Returns **ten** – interfacial tension between the phases

Return type float

Individual functions for each method can be accesed trought the `phasepy.sgt.ten_beta0_reference` for reference component method, `phasepy.sgt.ten_beta0_hk` for Cornellise path function, `phasepy.sgt.ten_beta0_sk` for Liang path function.

7.3.3 SGT for mixtures and $\beta_{ij} \neq 0$

When working with mixtures and at least one $\beta_{ij} \neq 0$, SGT has to be solved as a boundary value problem (BVP)with a finite interfacial lenght.

$$\sum_j c_{ij} \frac{d^2\rho_j}{dz^2} = \mu_i - \mu_i^0 \qquad i = 1, ..., c$$

$$\rho(z \to 0) = \rho^\alpha \qquad \rho(z \to L) = \rho^\beta$$

In phasepy two solution procedure are available for this purpose, both on them relies on orthogonal collocation. The first one, solve the BVP at a given interfacial lenght, then it computes the interfacial tension. After this first iteration the interfacial lenght is increased and the density profiles are solved again using the obtained solution as an initial guess, then the interfacial tension is computed again. This iterative procedure is repeated until the interfacial tension stops decreasing whithin a given tolerance (default value 0.01 mN/m). This procedure is inspired in the work of Liang and Michelsen.

First, as for any SGT computation, equilibria has to be computed.

```
>>> hexane = component(name = 'n-Hexane', Tc = 507.6, Pc = 30.25, Zc = 0.266, Vc =␣
↪371.0, w = 0.301261,
                ksv = [ 0.81185833, -0.08790848],
                cii = [ 5.03377433e-24, -3.41297789e-21,  9.97008208e-19],
                GC = {'CH3':2, 'CH2':4})
>>> ethanol = component(name = 'Ethanol', Tc = 514.0, Pc = 61.37, Zc = 0.241, Vc =␣
↪168.0, w = 0.643558,
```

(continues on next page)

(continued from previous page)

```
                ksv = [1.27092923, 0.0440421 ],
                cii = [ 2.35206942e-24, -1.32498074e-21,  2.31193555e-19],
                GC = {'CH3':1, 'CH2':1, 'OH(P)':1})
>>> mix = mixture(ethanol, hexane)
>>> a12, a21 = np.array([1141.56994427,  125.25729314])
>>> A = np.array([[0, a12], [a21, 0]])
>>> mix.wilson(A)
>>> eos = prsveos(mix, 'mhv_wilson')
>>> T = 320 #K
>>> X = np.array([0.3, 0.7])
>>> P0 = 0.3 #bar
>>> Y0 = np.array([0.7, 0.3])
>>> sol = bubblePy(Y0, P0, X, T, eos, full_output = True)
>>> Y = sol.Y
>>> P = sol.P
>>> vl = sol.v1
>>> vv = sol.v2
>>> #computing the density vector
>>> rhol = X / vl
>>> rhov = Y / vv
```

The correction β_{ij} has to be supplied to the eos with `eos.beta_sgt` method. Otherwise the influence parameter matrix will be singular and a error will be raised.

```
>>> bij = 0.1
>>> beta = np.array([[0, bij], [bij, 0]])
>>> eos.beta_sgt(beta)
```

Then the interfacial tension can be computed as follows:

```
>>> sgt_mix(rhol, rhov, T, P, eos, z0 = 10.,  rho0 = 'hyperbolic', full_output =␣
↪False)
>>> #interfacial tension in mN/m
>>> 14.367813285945807
```

In the example `z0` refers to the initial interfacial langht, `rho0` refers to the initial guess to solve the BVP. Available options are `'linear'` for linear density profiles, `hyperbolic` for density profiles obtained from hyperbolic tangent. Other option is to provide an array with the initial guess, the shape of this array has to be nc x n, where n is the number of collocation points. Finally, a TensionResult can be passed to `rho0`, this object is usually obtained from another SGT computation, as for example from a calculation with $\beta_{ij} = 0$.

If the `full_output` option is set to `True`, all the computated information will be given in a TensionResult object. Atributes are accessed similar as SciPy OptimizationResult.

```
>>> sol = sgt_mix(rhol, rhov, T, P, eos, z0 = 10.,  rho0 = 'hyperbolic', full_output␣
↪= False)
>>> sol.tension
... 14.36781328594585 #IFT in mN/m
>>> #density profiles and spatial coordiante access
>>> sol.rho
>>> sol.z
```

sgt_mix (*rho1, rho2, Tsat, Psat, model, rho0='linear', z0=10.0, dz=1.5, itmax=10, n=20, full_output=False, ten_tol=0.01, root_method='lm', solver_opt=None*)

SGT for mixtures and beta != 0 (rho1, rho2, T, P) -> interfacial tension

Parameters

- **rho1** (*float*) – phase 1 density vector
- **rho2** (*float*) – phase 2 density vector
- **Tsat** (*float*) – saturation temperature
- **Psat** (*float*) – saturation pressure
- **model** (*object*) – created with an EoS
- **rho0** (*string, array_like or TensionResult*) – inital values to solve the BVP, avaialable options are 'linear' for linear density profiles, 'hyperbolic' for hyperbolic like density profiles. An array can also be supplied or a TensionResult of a previous calculation.
- **z0** (*float*) – initial interfacial lenght
- **dz** (*float*) – increase in interfacial lenght per interation
- **itmax** (*int*) – maximun number of iterations
- **n** (*int, optional*) – number points to solve density profiles
- **full_output** (*bool, optional*) – wheter to outputs all calculation info
- **root_method** (*string, optional*) – Method used un SciPy's root function default 'lm', other options: 'krylov', 'hybr'. See SciPy documentation for more info
- **solver_opt** (*dict, optional*) – aditional solver options passed to SciPy solver

Returns **ten** – interfacial tension between the phases

Return type float

The second solution method is based on a modified SGT system of equations, proposed by Mu et al. This system introduced a time variable s which helps to get linearize the system of equations during the first iterations.

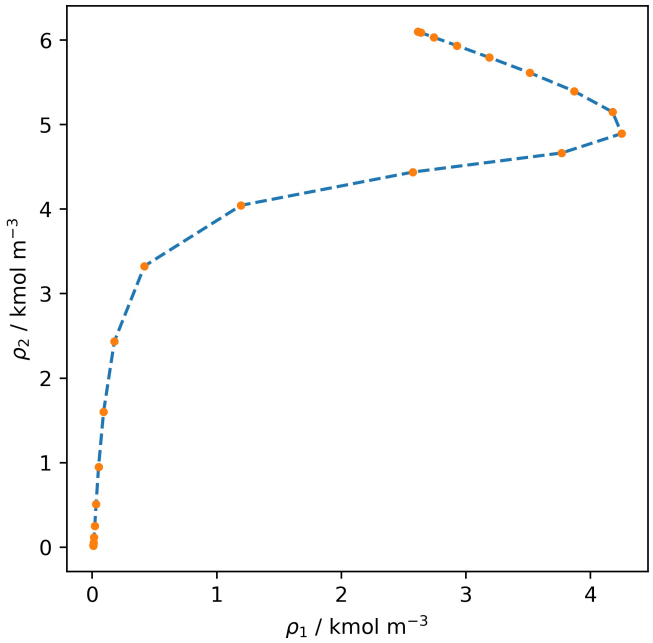
$$\sum_j c_{ij} \frac{d^2\rho_j}{dz^2} = \frac{\delta\rho_i}{\delta s} + \mu_i - \mu_i^0 \qquad i = 1, ..., c$$

$$\rho(z \to 0) = \rho^\alpha \qquad \rho(z \to L) = \rho^\beta$$

This differential equation is advanced in time until no further changes is found on the density profiles. Then the interfacial tension is computed. Its use is similar as the method described above, with the `msgt_mix` function.

```
>>> solm = msgt_mix(rhol, rhov, T, P, eos, z = 20, rho0 = sol, full_output = True)
>>> solm.tension
... 14.367827924919165 #IFT in mN/m
```

The density profiles obtained from each method are show in the following figure. The dashed line was computed solving the original BVP with increasing interfacial length and the dots were computed with the modified system.



msgt_mix(*rho1*, *rho2*, *Tsat*, *Psat*, *model*, *rho0='linear'*, *z=20.0*, *n=20*, *ds=0.1*, *itmax=30*, *rho_tol=0.01*, *full_output=False*, *root_method='lm'*, *solver_opt=None*)

SGT for mixtures and beta != 0 (rho1, rho2, T, P) -> interfacial tension

Parameters

- **rho1** (*float*) – phase 1 density vector
- **rho2** (*float*) – phase 2 density vector
- **Tsat** (*float*) – saturation temperature
- **Psat** (*float*) – saturation pressure
- **model** (*object*) – created with an EoS
- **rho0** (*string, array_like or TensionResult*) – inital values to solve the BVP, avaialable options are 'linear' for linear density profiles, 'hyperbolic' for hyperbolic like density profiles. An array can also be supplied or a TensionResult of a previous calculation.
- **z** (*float, optional*) – initial interfacial lenght
- **n** (*int, optional*) – number points to solve density profiles
- **ds** (*float, optional*) – time variable integration delta
- **itmax** (*int, optional*) – maximun number of iterations foward on time
- **rho_tol** (*float, optional*) – desired tolerance for density profiles
- **full_output** (*bool, optional*) – wheter to outputs all calculation info
- **root_method** (*string, optional*) – Method used un SciPy's root function default 'lm', other options: 'krylov', 'hybr'. See SciPy documentation for more info
- **solver_opt** (*dict, optional*) – aditional solver options passed to SciPy solver

Returns **ten** – interfacial tension between the phases

Return type float

7.4 phasepy.fit

In order to compute phase equilibria and interfacial properties it is necessary to count with pure component parameters as: Antoine correlation parameters, volume translataion constant, influence parameter for SGT, among others. Similarly, when working with mixtures interaction parameters of activity coefficients models or binary correction k_{ij} for quadratic mixing rule are necessary to predict phase equilibria. Often those parameters can be found in literature, but for many educational and research purposes there might be necessary to fit them to experimental data.

Phasepy includes several functions that relies on equilibria routines included in the package and in SciPy optimization tools for fitting models parameters. These functions are explained in the following secctions for pure component and for mixtures.

7.4.1 Pure component data

Depending on the model that will be used, different information might be needed. For general purposes there might be necessary to fit saturation pressure, liquid density and interfacial tension. Experimental data is needed, in this case water saturation properties is obtanied from NIST database, more experimental data can be found in DIPPR, TDE, Knovel or by your own measurements.

```
>>> #Experimental Saturation Data of water obtained from NIST
>>> #Saturation Temperature in Kelvin
>>> Tsat = np.array([290., 300., 310., 320., 330., 340., 350.,
...  360., 370., 380.])
>>> #Saturation Pressure in bar
>>> Psat = np.array([0.0192  , 0.035368, 0.062311, 0.10546 ,
... 0.17213 , 0.27188 , 0.41682 , 0.62194 , 0.90535 , 1.2885  ])
>>> #Saturated Liquid density in mol/cm3
>>> rhol = np.array([0.05544 , 0.055315, 0.055139, 0.054919,
... 0.054662, 0.054371, 0.054049, 0.053698, 0.053321, 0.052918])
>>> #Interfacial Tension in mN/m
>>> tension = np.array([73.21 , 71.686, 70.106, 68.47 , 66.781,
... 65.04 , 63.248, 61.406, 59.517, 57.581])
```

Saturation Pressure

First, Antoine Coefficients can be fitted to the following form:

$$\ln P = A - \frac{B}{T + C}$$

The model is fitted directly with Phasepy, optionally an initial guess can be passed.

```
>>> #Fitting Antoine Coefficients
>>> from phasepy.fit import fit_ant
>>> Ant = fit_ant(Tsat, Psat)
>>> #Objection function value, Antoine Parameters
>>> 5.1205342479858257e-05, [1.34826650e+01, 5.02634690e+03, 9.07664231e-04]
>>> #Optionally an initial guess for the parameters can be passed to the function
>>> Ant = fit_ant(Tsat, Psat, x0 = [11, 3800, -44])
```

(continues on next page)

(continued from previous page)

```
>>> #Objection function value, Antoine Parameters
>>> 2.423780448316938e-07,[ 11.6573823 , 3800.11357063,  -46.77260501]
```

fit_ant (*Texp, Pexp, x0=[0, 0, 0]*)
fit Antoine parameters, base exp

Parameters

- **Texp** (*experimental temperature in K.*) –
- **Pexp** (*experimental pressure in bar.*) –
- **x0** (*array_like, optional*) – initial values.

Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Following with saturation pressure fitting, when using PRSV EoS it is necessary to fit α parameters, for these purpose a component has to be defined and then using the experimental data the parameters can be fitted.

$$\alpha = (1 + k_1[1 - \sqrt{T_r}] + k_2[1 - T_r][0.7 - T_r])^2$$

```
>>> #Fitting ksv for PRSV EoS
>>> from phasepy.fit import fit_ksv
>>> #parameters of pure component obtained from DIPPR
>>> name = 'water'
>>> Tc = 647.13 #K
>>> Pc = 220.55 #bar
>>> Zc = 0.229
>>> Vc = 55.948 #cm3/mol
>>> w = 0.344861
>>> pure = component(name = name, Tc = Tc, Pc = Pc, Zc = Zc,
... Vc =  Vc, w = w)
>>> ksv = fit_ksv(pure, Tsat, Psat)
>>> #Objection function value, ksv Parameters
>>> 1.5233471126821199e-10, [ 0.87185176, -0.06621339]
```

fit_ksv (*component, Texp, Pexp, ksv0=[1, 0]*)
fit PRSV alpha parameters

Parameters

- **component** (*object*) – created with component class
- **Texp** (*array_like*) – experimental temperature in K.
- **Pexp** (*array_like*) – experimental pressure in bar.
- **ksv0** (*array_like, optional*) – initial values.

Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Volume Translation

When working with cubic EoS, there might an interest for a better prediction of liquid densities, this can be done by a volume translation. This volume correction doesn't change equilibria results and its parameters is obtained in Phasepy as follows:

```
>>> from phasepy import prsveos
>>> from phasepy.fit import fit_vt
>>> #Defining the component with the optimized alpha parameters
>>> pure = component(name = name, Tc = Tc, Pc = Pc, Zc = Zc,
...  Vc = Vc, w = w, ksv = [ 0.87185176, -0.06621339] )
>>> vt = fit_vt(pure, prsveos, Tsat, Psat, rhol)
>>> #Objetive function and volume translation
>>> 0.001270834833817397, 3.46862174
```

fit_vt (*component*, *eos*, *Texp*, *Pexp*, *rhoexp*, *c0=0.0*)
fit Volume Translation for cubic EoS

Parameters

- **component** (*object*) – created with component class
- **eos** (*function*) – cubic eos function
- **Texp** (*array_like*) – experimental temperature in K.
- **Pexp** (*array_like*) – experimental pressure in bar.
- **rhoexp** (*array_like*) – experimental liquid density at given temperature and pressure in mol/cm3.
- **c0** (*float, optional*) – initial values.

Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Influence parameter for SGT

Finally, influence parameters are necessary to compute interfacial properties, these can be fitted with experimental interfacial tension.

```
>>> from phasepy.fit import fit_cii
>>> #Defining the component with the volume traslation parameter.
>>> pure = component(name = name, Tc = Tc, Pc = Pc, Zc = Zc,
... Vc = Vc, w = w, ksv = [ 0.87185176, -0.06621339], c =  3.46862174)
>>> eos = prsveos(pure, volume_translation = False)
>>> cii = fit_cii(tension, Tsat, eos, order = 2)
>>> #fitted influence parameter polynomial
>>> [2.06553362e-26, 2.64204784e-23, 4.10320513e-21]
```

Beware that influence parameter in cubic EoS absorbs density deviations from experimental data, if there is any volume correction the influence parameter will change.

```
>>> eos = prsveos(pure, volume_translation = True)
>>> cii = fit_cii(tension, Tsat, eos, order = 2)
>>> #fitted influence parameter polynomial with volume translation
>>> [2.74008872e-26, 1.23088986e-23, 3.05681188e-21]
```

fit_cii (*tenexp*, *Texp*, *model*, *order=2*, *n=100*, *P0=None*)
fit influence parameters of SGT

Parameters

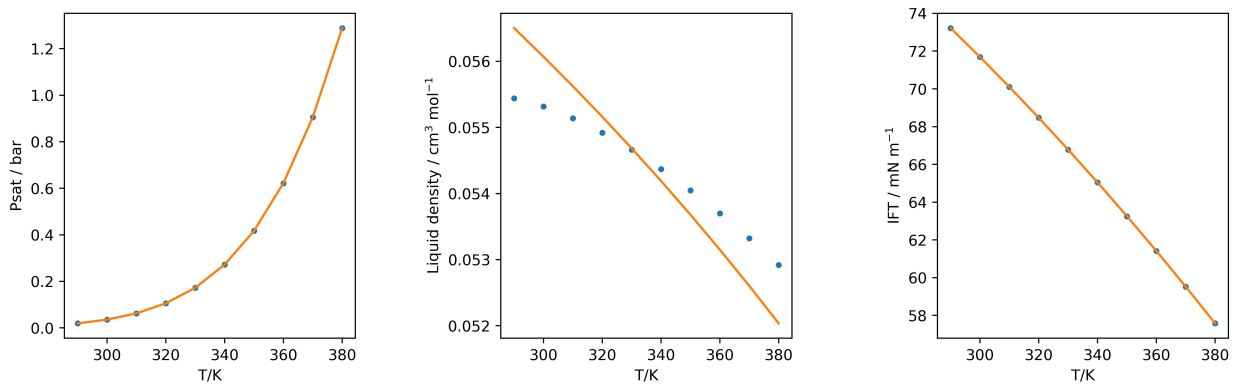
- **tenexp** (*array_like*) – experimental surface tension in mN/m
- **Texp** (*array_like*) – experimental temperature in K.

- **model** (*object*) – created from eos and component
- **order** (*int, optional*) – order of cii polynomial
- **n** (*int, optional*) – number of integration points in SGT
- **P0** (*array_like , optional*) – initial guess for saturation pressure

Returns **cii** – polynomial coefficients of influence parameters of SGT

Return type array_like

The perfomance of the fitted interaction parameters can be compared against the experimental data (dots), the following figure shows the behavior of the cubic EoS for the saturation pressure, liquid density and interfacial tension (oranges lines).



7.4.2 Equilibrium data

Phasepy includes function to fit models for phase equilibria of binary mixtures, the fitted parameters by pairs can be used then with mixtures containing more components. Depending and model, there might be more of one type of phase equilibria. Suppose that the parameters model are represented by $\underline{\xi}$, the phasepy functions will fit the given experimental data with the followings objectives functions:

If there is Vapor-Liquid Equilibria (VLE):

$$FO_{VLE}(\underline{\xi}) = \sum_{j=1}^{Np} \left[\sum_{i=1}^{c} (y_{i,j}^{cal} - y_{i,j}^{exp})^2 + \left(\frac{P_j^{cal}}{P_j^{exp}} - 1 \right)^2 \right]$$

If there is Liquid-Liquid Equilibria (LLE):

$$FO_{LLE}(\underline{\xi}) = \sum_{j=1}^{Np} \sum_{i=1}^{c} \left[(x_{i,j} - x_{i,j}^{exp})^2 + (w_{i,j} - w_{i,j}^{exp})^2 \right]$$

If there is Vapor-Liquid-Liquid Equilibria (VLLE):

$$FO_{VLLE}(\underline{\xi}) = \sum_{j=1}^{Np} \left[\sum_{i=1}^{c} \left[(x_{i,j}^{cal} - x_{i,j}^{exp})^2 + (w_{i,j}^{cal} - w_{i,j}^{exp})^2 + (y_{i,j}^{cal} - y_{i,j}^{exp})^2 \right] + \left(\frac{P_j^{cal}}{P_j^{exp}} - 1 \right)^2 \right]$$

If there is more than one type of phase equilibria, Phasepy will sum the errors of each one. As an example the parameters for the system of ethanol and water will be fitted, first the experimental data has to be loaded and then set up as a tuple as if shown in the following code block.

```
>>> #Vapor Liquid equilibria data obtanied from Rieder, Robert M. y A. Ralph Thompson␣
↪(1949).
>>> # Vapor-Liquid Equilibria Measured by a GillespieStill - Ethyl Alcohol - Water␣
↪System.
>>> #Ind. Eng. Chem. 41.12, 2905-2908.
>>> datavle = (Xexp, Yexp, Texp, Pexp)
```

If the system exhibits any other type phase equilibria the necessary tuples would have the following form: >>> datalle = (Xexp, Wexp, Texp, Pexp) >>> datavlle = (Xexp, Wexp, Yexp, Texp, Pexp)

Here, `Xexp`, `Wexp` and `Yexp` are experimental mole fraction for liquid, liquid and vapor phase, respectively. `Texp` and `Pexp` are experimental temperature and pressure, respectively.

After the experimental data is available, the mixture with the components is created.

```
>>> water = component(name = 'Water', Tc = 647.13, Pc = 220.55, Zc = 0.229,
... Vc = 55.948, w = 0.344861, ksv = [ 0.87292043, -0.06844994],
... Ant =  [  11.72091059, 3852.20302815,  -44.10441047])
>>> ethanol = component(name = 'Ethanol', Tc = 514.0, Pc = 61.37, Zc = 0.241,
... Vc = 168.0, w = 0.643558, ksv = [1.27092923, 0.0440421 ],
... Ant = [  12.26474221, 3851.89284329,  -36.99114863])
>>> mix = mixture(ethanol, water)
```

Fitting QMR mixing rule

As an scalar is been fitted, SciPy recommends to give a certain interval where the minimum could be found, the function `fit_kij` handles this optimization as follows:

```
>>> from phasepy.fit import fit_kij
>>> mixkij = mix.copy()
>>> fit_kij((-0.15, -0.05), prsveos, mixkij, datavle)
>>> #optimized kij value
>>> -0.10726854855665718
```

fit_kij(*kij_bounds, eos, mix, datavle=None, datalle=None, datavlle=None, weights_vle=[1.0, 1.0], weights_lle=[1.0, 1.0], weights_vlle=[1.0, 1.0, 1.0, 1.0], minimize_options={}*)

fit_kij: attemps to fit kij to VLE, LLE, VLLE

Parameters

- **kij_bounds** (*tuple*) – bounds for kij correction
- **eos** (*function*) – cubic eos to fit kij for qmr mixrule
- **mix** (*object*) – binary mixture
- **datavle** (*tuple, optional*) – (Xexp, Yexp, Texp, Pexp)
- **datalle** (*tuple, optional*) – (Xexp, Wexp, Texp, Pexp)
- **datavlle** (*tuple, optional*) – (Xexp, Wexp, Yexp, Texp, Pexp)
- **weights_vle** (*list or array_like, optional*) – weights_vle[0] = weight for Y composition error, default to 1. weights_vle[1] = weight for bubble pressure error, default to 1.
- **weights_lle** (*list or array_like, optional*) – weights_lle[0] = weight for X (liquid 1) composition error, default to 1. weights_lle[1] = weight for W (liquid 2) composition error, default to 1.

- **weights_vlle** (*list or array_like, optional*) – weights_vlle[0] = weight for X (liquid 1) composition error, default to 1. weights_vlle[1] = weight for W (liquid 2) composition error, default to 1. weights_vlle[2] = weight for Y (vapor) composition error, default to 1. weights_vlle[3] = weight for equilibrium pressure error, default to 1.
- **minimize_options** (*dict*) – Dictionary of any additional spefication for scipy minimize_scalar

Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Fitting NRTL parameters

As an array is been fitted, multidimentional optimization alogirthms are used, the function `fit_nrtl` handles this optimization with several options available. If a fixed value of the aleatory factor is used the initial guess has the following form:

```
>>> nrtl0 = np.array([A12, A21])
```

```
>>> from phasepy.fit import fit_nrtl
>>> mixnrtl = mix.copy()
>>> #Initial guess of A12, A21
>>> nrtl0 = np.array([-80.,  650.])
>>> fit_nrtl(nrtl0, mixnrtl, datavle, alpha_fixed = True)
>>> #optimized values
>>> [-84.77530335, 648.78439102]
```

By default bubble points using activity coefficient models use Tsonopoulos virial correlation, if desired ideal gas or Abbott correlation can be used.

```
>>> from phasepy import ideal_gas, Abbott
>>> #Initial guess of A12, A21
>>> nrtl0 = np.array([-80.,  650.])
>>> fit_nrtl(nrtl0, mixnrtl, datavle, alpha_fixed = True, virialmodel = 'ideal_gas')
>>> #optimized values
>>> [-86.22483806, 647.6320968 ]
>>> fit_nrtl(nrtl0, mixnrtl, datavle, alpha_fixed = True, virialmodel = 'Abbott')
>>> #optimized values
>>> [-84.81672981, 648.75311712]
```

By default the aleatory factor is set to 0.2, this value ca be changed by passing another value to `alpha0` to the fitting function.

```
>>> fit_nrtl(nrtl0, mixnrtl, datavle, alpha_fixed = True, alpha0 = 0.3)
>>> #optimized values
>>> [-57.38407954, 664.29319445]
```

If the aleatory factor needs to be optimized it can be included setting alpha_fixed to False, in this case the initial guess has the following form:

```
>>> nrtl0 = np.array([A12, A21, alpha])
```

```
>>> #Initial guess of A12, A21
>>> nrtl0 = np.array([-80.,  650.,  0.2])
>>> fit_nrtl(nrtl0, mixnrtl, datavle, alpha_fixed = False)
```

(continues on next page)

(continued from previous page)

```
>>> #optimized values for A12, A21, alpha
>>> [-5.53112687e+01,  6.72701992e+02,  3.19740734e-01]
```

Temperature dependent parameters can be fitted setting the option Tdep = True in `fit_nrtl`, when this option is used the parameters are computed as:

$$A12 = A12_0 + A12_1 T$$
$$A21 = A21_0 + A21_1 T$$

The initial guess passed to the fit function has the following form:

```
>>> nrtl0 = np.array([A12_0, A21_0, A12_1, A21_1, alpha])
```

or, if alpha fixed is used.

```
>>> nrtl0 = np.array([A12_0, A21_0, A12_1, A21_1])
```

fit_nrtl(*x0, mix, datavle=None, datalle=None, datavlle=None, alpha_fixed=False, alpha0=0.2, Tdep=False, virialmodel='Tsonopoulos', weights_vle=[1.0, 1.0], weights_lle=[1.0, 1.0], weights_vlle=[1.0, 1.0, 1.0, 1.0], minimize_options={}*)

fit_nrtl: attemps to fit nrtl parameters to VLE, LLE, VLLE

Parameters

- **x0** (*array_like*) – initial values interaction parameters (and aleatory factor)
- **mix** (*object*) – binary mixture
- **datavle** (*tuple, optional*) – (Xexp, Yexp, Texp, Pexp)
- **datalle** (*tuple, optional*) – (Xexp, Wexp, Texp, Pexp)
- **datavlle** (*tuple, optional*) – (Xexp, Wexp, Yexp, Texp, Pexp)
- **alpha_fit** (*bool, optional*) – if True fix aleatory factor to the value of alpha0
- **alpha0** (*float*) – value of aleatory factor if fixed
- **Tdep** (*bool, optional*) – Wheter the energy parameters have a temperature dependence
- **virialmodel** (*function*) – function to compute virial coefficients, available options are 'Tsonopoulos', 'Abbott' or 'ideal_gas'
- **weights_vle** (*list or array_like, optional*) – weights_vle[0] = weight for Y composition error, default to 1. weights_vle[1] = weight for bubble pressure error, default to 1.
- **weights_lle** (*list or array_like, optional*) – weights_lle[0] = weight for X (liquid 1) composition error, default to 1. weights_lle[1] = weight for W (liquid 2) composition error, default to 1.
- **weights_vlle** (*list or array_like, optional*) – weights_vlle[0] = weight for X (liquid 1) composition error, default to 1. weights_vlle[1] = weight for W (liquid 2) composition error, default to 1. weights_vlle[2] = weight for Y (vapor) composition error, default to 1. weights_vlle[3] = weight for equilibrium pressure error, default to 1.
- **minimize_options** (*dict*) – Dictionary of any additional spefication for scipy minimize

Notes

if Tdep True parameters are treated as: a12 = a12_1 + a12T * T a21 = a21_1 + a21T * T

if alpha_fixed False and Tdep True: x0 = [a12, a21, a12T, a21T, alpha]

if alpha_fixed False and Tdep False: x0 = [a12, a21, alpha]

if alpha_fixed True and Tdep False: x0 = [a12, a21]

Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Fitting Wilson's model parameters

As an array is been fitted, multidimentional optimization alogirthms are used, the function `fit_wilson` handles this optimization.

```
>>> from phasepy.fit import fit_wilson
>>> mixwilson = mix.copy()
>>> #Initial guess of A12, A21
>>> wilson0 = np.array([-80.,  650.])
>>> fit_wilson(wilson0, mixwilson, datavle)
>>> #optimized values
>>> [163.79243953, 497.05518499]
```

Tsonopoulos virial correlation is used by default, if desired ideal gas or Abbott correlation can be used.

```
>>> fit_wilson(wilson0, mixwilson, datavle, virialmodel = 'ideal_gas')
>>> #optimized value
>>> [105.42279401, 517.2221969 ]
```

fit_wilson (*x0, mix, datavle, virialmodel='Tsonopoulos', weights_vle=[1.0, 1.0], minimize_options={}*)

fit_wilson: attemps to fit wilson parameters to VLE

Parameters

- **x0** (*array_like*) – initial values a12, a21 in K
- **mix** (*object*) – binary mixture
- **datavle** (*tuple*) – (Xexp, Yelv, Texp, Pexp)
- **virialmodel** (*function*) – function to compute virial coefficients, available options are 'Tsonopoulos', 'Abbott' or 'ideal_gas'
- **weights_vle** (*list or array_like, optional*) – weights_vle[0] = weight for vapor composition error, default to 1. weights_vle[1] = weight for bubble pressure error, default to 1.
- **minimize_options** (*dict*) – Dictionary of any additional spefication for scipy minimize

Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Fitting Redlich-Kister interaction parameters

As an array is been fitted, multidimentional optimization alogirthms are used, the function `fit_rk` handles this optimization. Redlich-Kister expansion is programmed for n terms of the expansion, this fitting function will optimize considering the lenght of the array passed as an initial guess.

If rk0 is an scalar it reduced to Porter model, if it is array of size 2 it reduces to Margules Model.

```
>>> from phasepy.fit import fit_rk
>>> mixrk = mix.copy()
>>> rk0 = np.array([0, 0])
>>> fit_rk(rk0, mixrk, datavle, Tdep =  False)
>>> #optimized values
>>> [ 1.1759649 , -0.44487888]
```

Temperature dependent parameters can be fitted in which case the initial guess will be splitted into two arrays.

```
>>> c, c1 = np.split(rk0, 2)
```

Finally the parameters are computed as $G = c + c1/T$.

Similarly as NRTl and Wilson's model, virial correlation can be changed by passing the desired function to the `virialmodel` argument.

```
>>> fit_rk(rk0, mixrk, datavle, Tdep =  False, virialmodel = 'ideal_gas')
>>> [ 1.16854714, -0.43874371]
```

fit_rk (*inc0, mix, datavle=None, datalle=None, datavlle=None, Tdep=False, virialmodel='Tsonopoulos', weights_vle=[1.0, 1.0], weights_lle=[1.0, 1.0], weights_vlle=[1.0, 1.0, 1.0, 1.0], minimize_options={}*)

fit_rk: attemps to fit RK parameters to VLE, LLE, VLLE

Parameters

- **inc0** (*array_like*) – initial values to RK parameters
- **mix** (*object*) – binary mixture
- **datavle** (*tuple, optional*) – (Xexp, Yexp, Texp, Pexp)
- **datalle** (*tuple, optional*) – (Xexp, Wexp, Texp, Pexp)
- **datavlle** (*tuple, optional*) – (Xexp, Wexp, Yexp, Texp, Pexp)
- **Tdep** (*bool,*) – wheter the parameter will have a temperature dependence
- **virialmodel** (*function*) – function to compute virial coefficients, available options are 'Tsonopoulos', 'Abbott' or 'ideal_gas'
- **weights_vle** (*list or array_like, optional*) – weights_vle[0] = weight for Y composition error, default to 1. weights_vle[1] = weight for bubble pressure error, default to 1.
- **weights_lle** (*list or array_like, optional*) – weights_lle[0] = weight for X (liquid 1) composition error, default to 1. weights_lle[1] = weight for W (liquid 2) composition error, default to 1.
- **weights_vlle** (*list or array_like, optional*) – weights_vlle[0] = weight for X (liquid 1) composition error, default to 1. weights_vlle[1] = weight for W (liquid 2) composition error, default to 1. weights_vlle[2] = weight for Y (vapor) composition error, default to 1. weights_vlle[3] = weight for equilibrium pressure error, default to 1.

- **minimize_options** (*dict*) – Dictionary of any additional spefication for scipy minimize

Notes

if Tdep true: C = C' + C'T

if Tdep true: inc0 = [C'0, C'1, C'2, . . . , C'0T, C'1T, C'2T. . .]

if Tdep false: inc0 = [C0, C1, C2. . .]

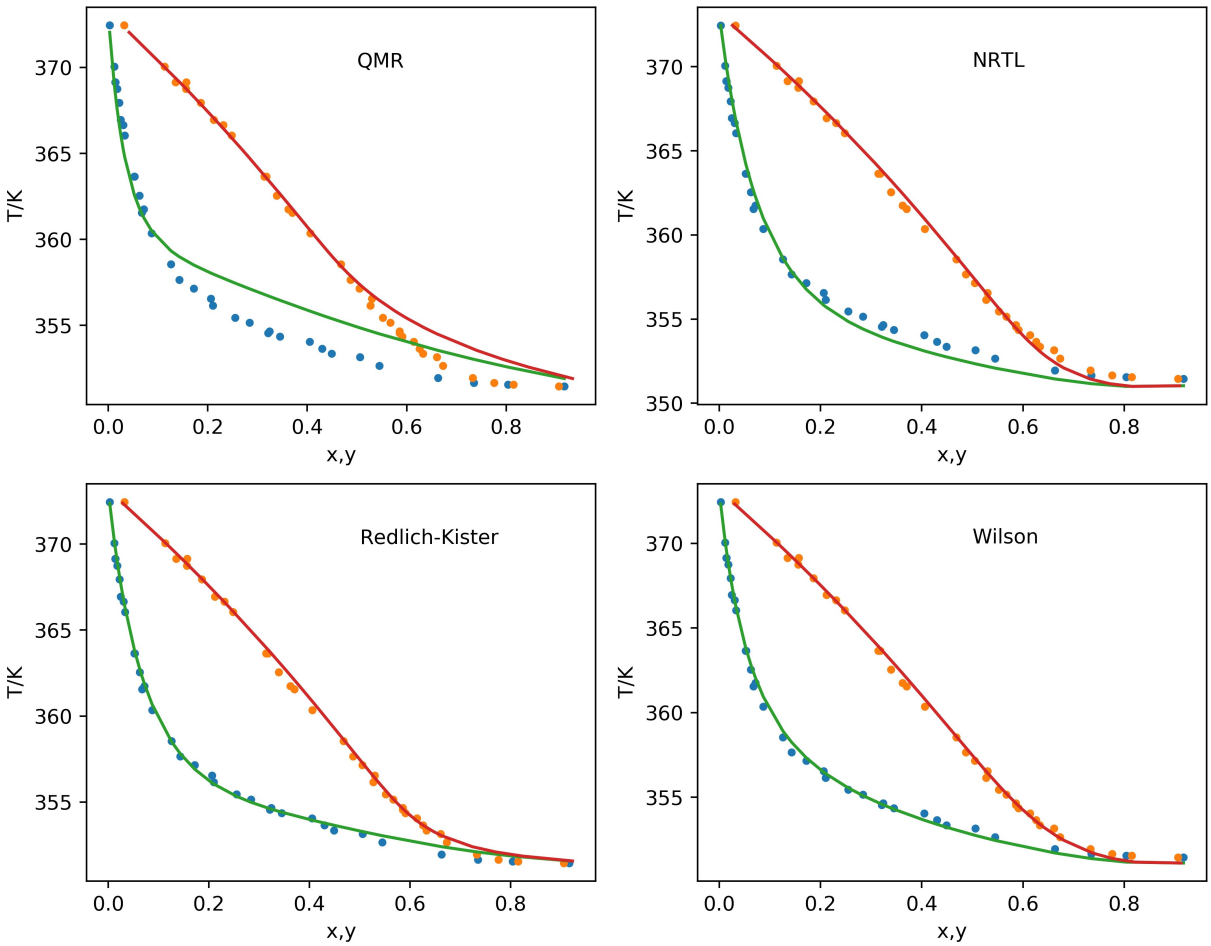
Returns **fit** – Result of SciPy minimize

Return type OptimizeResult

Multidimentional minimization in SciPy are perfomed with `minimize` function, aditional command can be passed to this function in order to change tolerance, number of function evaluations or mimization method. This is done by passing a dictionary with the settings to `minimize_options` in `fit_nrtl`, `fit_wilson` or `fit_rk`. For example:

```
>>> #changing the minimization method
>>> minimize_options = {'method' : 'Powell'}
>>> fit_rk(rk0, mixrk, datavle, Tdep =  False, minimize_options = minimize_options )
```

The fitted parameters can be compared against the equilibrium data for each model. The following figure shows the perfomance of QMR, NRTL model, Wilson model and Redlich Kister expansion.



7.5 Indices and Search

- genindex
- modindex
- search

Python Module Index

p

Index