CONTENTS:

1 What is SGTPy? 1
2 Installation Prerequisites 3
3 Installation 5
4 Getting Started 7
5 Latest source code 9
6 Bug reports 11
7 License information 13
WHAT IS SGTPY?

SGTPy is an open-source python package of SAFT-VR-Mie Equation of State (EOS). SGTPy allows to work with pure fluids and fluid mixtures, additionally the fluids can be modeled considering association, cross-association and polar contributions. SGTPy was built on top of phasepy’s phase equilibrium and Square Gradient Theory (SGT) functions. These functions were updated to speed-up calculations of associative mixtures.

SGTPy relies on Numpy, SciPy and PhasePy.
CHAPTER
TWO

INSTALLATION PREREQUISITES

• numpy
• scipy
• cython
Get the latest version of SGTPy from https://pypi.python.org/pypi/SGTPy/

If you have an installation of Python with pip, simple install it with:

   $ pip install SGTPy

To get the git version, run:

   $ git clone https://github.com/gustavochm/SGTPy
SGTPy easily allows you to perform phase equilibria and interfacial properties calculations using SAFT-VR-Mie EoS. First, components are defined with their molecular parameters, then a mixture can be created with them.

```python
>>> import numpy as np
>>> from SGTPy import component, mixture, saftvrmie

>>> ethanol = component('ethanol2C', ms=1.7728, sigma=3.5592, eps=224.50,
lambda_r=11.319, lambda_a=6., eAB=3018.05, rcAB=0.3547,
rdAB=0.4, sites=[1,0,1], cii=5.3141080872882285e-20)

>>> hexane = component('hexane', ms=1.96720036, sigma=4.54762477,
eps=377.60127994, lambda_r=18.41193194,
cii=3.581510586936205e-19)

>>> mix = mixture(hexane, ethanol)

>>> # fitted to experimental data
>>> kij = 0.011818492037463553
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> mix.kij_saft(Kij)

>>> eos = saftvrmie(mix)

The eos object can be used to compute phase equilibria.

```python
>>> from SGTPy.equilibrium import bubblePy

>>> # computing bubble point
>>> T = 298.15 # K
>>> x = np.array([0.3, 0.7])

>>> # initial guesses for vapor composition and pressure
>>> y0 = 1.*x
>>> P0 = 8000. # Pa
>>> sol = bubblePy(y0, P0, x, T, eos, full_output=True)

Finally, the equilibria results can be used to model the interfacial behavior of the mixture using SGT.

```python
>>> from SGTPy.sgt import sgt_mix

>>> # reading solution object
>>> y, P = sol.Y, sol.P
>>> vl, vv = sol.v1, sol.v2

>>> # density vector of each phase
>>> rhox = x/vl
>>> rhoy = y/vv
>>> bij = 0.05719272059410664

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

(continues on next page)
>>> # solving BVP of SGT with 25 colocation points
>>> solsgt = sgt_mix(rhoy, rhox, T, P, eos, n = 25, full_output = True)

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
FIVE

LATEST SOURCE CODE

The latest development version of SGTPy’s sources can be obtained at


git clone https://github.com/gustavochm/SGTPy
CHAPTER SIX

BUG REPORTS

To report bugs, please use the SGTPy’s Bug Tracker at:

https://github.com/gustavochm/SGTPy/issues
CHAPTER
SEVEN

LICENSE INFORMATION

This package is part of the article *SGTPy: A Python open-source code for calculating the interfacial properties of fluids based on the Square Gradient Theory using the SAFT-VR Mie equation of state* by Andrés Mejía, Erich A. Müller and Gustavo Chaparro. Currently under revision in J. Chem. Inf. Model.

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 SGTPy license, if it is convenient for you, please cite SGTPy if used in your work. Please also consider contributing any changes you make back, and benefit the community.

7.1 SGTPy

SGTPy is an object-oriented python software for phase equilibria and interfacial properties calculation using SAFT-VR-Mie EoS. The SAFT-VR-Mie Equation of State describes the Helmholtz free energy as follows:

\[
a = a^{IG} + a^{MONO} + a^{CHAR} + a^{ASSOC} + a^{POL}
\]

Where, \(a\) is the total Helmholtz free energy, \(a^{IG}\) is the ideal gas reference, \(a^{MONO}\) represents monomer (unbounded) contribution, \(a^{CHAR}\) accounts for the formation of chain and ring molecule while \(a^{ASSOC}\) accounts for the intermolecular association contribution and \(a^{POL}\) represents the polar contribution. For further information about each contribution, we recommend the original references\(^1,^2,^3\).

The coded equation of state was tested to pass the following molar partial property test and Gibbs-Duhem consistency:

\[
\ln \phi - \sum_{i=1}^{c} x_i \ln \hat{\phi}_i = 0
\]

\[
\frac{d \ln \phi}{dP} - \frac{Z - 1}{P} = 0
\]

\[
\sum_{i=1}^{c} x_i d \ln \hat{\phi}_i = 0
\]

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

To use SGTPy, first, it is required to create components and mixtures, and then combine them with the equation of state to create a final model object, which can be used to carry out the desired calculations.

\(^1\) Journal of Chemical Physics, 139(15), 1–37 (2013).
### 7.1.1 SGTPy.component

SGTPy.component object stores pure component information needed to evaluate SAFT-VR-Mie EoS for phase equilibria and interfacial properties computation.

The simplest scenario is coarse-grained (CG) non-associating non-polar fluids. The parameters needed for these molecules are the numbers of segments ($m_s$), well-depth of Mie potential ($\epsilon$) in K units, size parameter of Mie potential ($\sigma$) in Å ($10^{-10}$ m), attractive ($\lambda_a$) and repulsive ($\lambda_r$) exponents of Mie Potential A component can be created as follows:

```python
>>> from SGTPy import component
>>> methane = component('methane', ms=1.0, sigma=3.752, eps=170.75, lambda_r=16.39, lambda_a=6.)
>>> dodecane = component('dodecane', ms=4.0, sigma=4.351, eps=378.56, lambda_r=18.41, lambda_a=6.)
```

The molecular parameters for CG molecules can be obtained from the Bottled-saft webpage. Additionally, a simple corresponding state parametrization\(^1\) is available through SGTPy. This parametrization requires the critical temperature ($T_c$), acentric factor ($\omega$) and depend on the desired number of segments ($m_s$) and the liquid density at a reduced temperature equal to 0.7 in mol/m\(^3\).

```python
>>> hexane = component('hexane', Tc=507.6, w=0.301261)
>>> hexane.saftvrmie_forcefield(ms=2, rhol07=6973.5)
>>> # sigma in meters, epsilon in K and lambda_r...
... (4.510744361846902e-10, 373.5197182392722, 19.43604939815)
```

The SAFT-VR-Mie EoS has been extended to homonuclear rings\(^2\), for these type of fluids and additional geometric parameter ($\chi$, ring in SGTPY) is needed.

![Diagram showing different ring molecules](image)

A ring molecule can be created as follows:

```python
>>> benzene = component('benzene', ms=3, sigma=3.441, eps=230.30, lambda_r=10.45, lambda_a=6., ring=1.4938)
```

---

For the case of pure self-associating fluid, three extra parameters are needed: the association energy ($\varepsilon_{AB}$) in K units, the association range ($r_{cAB}/\sigma$) and association center position ($r_{dAB}/\sigma$). The association scheme is characterized by the triple [B, P, N], which indicates the number of bipolar, positive and negative association sites, respectively. An equivalence table for common association schemes and the [B, P, N] triplet is shown below.

<table>
<thead>
<tr>
<th>Association Scheme</th>
<th>[B, P, N]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[0, 0, 1]</td>
</tr>
<tr>
<td>2B</td>
<td>[0, 1, 1]</td>
</tr>
<tr>
<td>2C</td>
<td>[1, 0, 1]</td>
</tr>
<tr>
<td>3B</td>
<td>[0, 1, 2]</td>
</tr>
<tr>
<td>4B</td>
<td>[0, 3, 1]</td>
</tr>
<tr>
<td>4C</td>
<td>[0, 2, 2]</td>
</tr>
</tbody>
</table>

Parameters for self-associating fluid, i.e. alcohols and water, for SAFT-VR-Mie can be found in Ref. 3. An associating molecule can be created as follows:

```python
>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2])
```

Additionally, a polar contribution⁴ can be accounted for, this contribution requires the definition of a dipolar moment ($\mu$, mupol in SGTPy) in Debye units, and the number of polar sites ($n_p$, npol in SGTPy). Molecular parameters for polar molecules can be found in Ref.⁷ and⁸.

```python
>>> butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077 , eps = 277.892,
... lambda_r = 10.6689, lambda_a = 6., eAB = 3300.0, rcAB = 0.2615,
... rdAB = 0.4, sites = [1,0,1], npol = 1.45, mupol = 1.6609)
```

Finally, in order to model the interfacial behavior, the influence parameter ($c_{ii}$) in J m$^5$/mol$^2$ is required.

```python
>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2], cii = 1.5371939421515458e-20)
>>> butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077 , eps = 277.892,
... lambda_r = 10.6689, lambda_a = 6., eAB = 3300.0, rcAB = 0.2615,
... rdAB = 0.4, sites = [1,0,1], npol = 1.45, mupol = 1.6609,
... cii = 1.5018715324070352e-19)
```

**Warning:** User is required to supply the necessary parameters for the EoS to work properly.

---

⁵ Fluid Phase Equilibria, 455, 24–42 (2018)
7.1.2 SGTPy.mixture

SGTPy.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:

```python
>>> from SGTPy import component, mixture

ethanol = component('ethanol2C', ms=1.7728, sigma=3.5592, eps=224.50, 
lambda_r=11.319, lambda_a=6., eAB = 3018.05, rcAB=0.3547, 
rdAB=0.4, sites=[1,0,1], cii=5.3141080872882285e-20)

hexane = component('hexane', ms=1.96720036, sigma=4.54762477, eps=377.60127994, 
lambda_r=18.41193194, cii=3.581510586936205e-19)

# creating mixture
mix = mixture(hexane, ethanol)
```

Additional components can be added to the mixture with the SGTPy.mixture.add_component() method.

```python
>>> cpme = component('cpme', ms=2.32521144, sigma=4.13606074, eps=343.91193798, lambda_r=14.15484877, 
lambda_a=6.0, npol=1.91990385, mupol=1.27, sites=[0,0,1], 
cii=3.5213681817448466e-19)

# adding a component
mix.add_component(cpme)
```

Once all components have been added to the mixture, the interaction parameters must be supplied. SGTPy considers $k_{ij}$ binary correction for the Mie potential interaction energy and $l_{ij}$ for the cross-association energy correction. The method SGTPy.mixture.kij_saft() sets the binary correction $k_{ij}$ for the Mie potential interaction energy.

```python
>>> from SGTPy import component, mixture

methane = component('methane', ms = 1.0, sigma = 3.752, eps = 170.75, 
lambda_r = 16.39, lambda_a = 6.)

dodecane = component('dodecane', ms = 4.0, sigma = 4.351, eps = 378.56, 
lambda_r = 18.41, lambda_a = 6.)

mix = mixture(methane, dodecane)

# interaction parameter optimized from phase equilibria data
kij = -0.02199102576365056
Kij = np.array([[0, kij], [kij, 0]])

# setting kij correction
mix.kij_saft(Kij)
```

The method SGTPy.mixture.lij_saft() sets the binary correction $l_{ij}$ for the cross-association energy.

```python
>>> water = component('water', ms = 1.7311, sigma = 2.4539, eps = 110.85, 
lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB=0.5624, 
rAB=0.4, sites = [0,2,2], cii = 1.5371939421515458e-20)

butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077, eps = 277.892, 
lambda_r = 10.6689, lambda_a = 6., eAB = 3300.0, rcAB = 0.2615,
```

(continues on next page)
...  rdAB = 0.4, sites = [1,0,1], npol = 1.45, mupol = 1.6609,
...  cii = 1.5018715324070352e-19)
>>> # optimized from experimental LLE
>>> kij, lij = np.array([-0.00736075, -0.00737153])
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> Lij = np.array([[0, lij], [lij, 0]])
>>> # setting interactions corrections
>>> mix.kij_saft(Kij)
>>> mix.lij_saft(Lij)

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

With the class SGTPy.component, only pure component info is saved. These parameters are required to evaluate SAFT-VR-Mie EoS. This includes numbers of segments ($m_s$), well-depth of Mie potential ($\epsilon$) in K units, size parameter of Mie potential ($\sigma$) in Å ($10^{-10}$ m), attractive ($\lambda_a$) and repulsive ($\lambda_r$) exponents of Mie Potential. If the fluid is modeled as a ring it requires a geometric factor ($\chi$). For the case of pure self-associating fluid, three extra parameters are needed: the association energy ($\epsilon_{AB}$) in K units, the association range ($r_{d AB}/\sigma$) and association center position ($r_{AB} d/\sigma$). Additionally, the association scheme is characterized by the triple [B, P, N], which indicates the number of bipolar, positive and negative association sites, respectively. The polar contribution requires the definition of a dipolar moment ($\mu$) in Debye units, and the number of polar sites ($n_p$). Finally, the influence parameter ($c_ii$) in J m$^5$/mol$^2$ is required to study the interfacial behavior using Square Gradient Theory.

A mixture can be created from two components and the SGTPy.mixture class:

```python
>>> from SGTPy import mixture
>>> mix = mixture(water, butanol)
>>> mix.sigma
... [2.4539e-10, 4.107700000000002e-10]  # stored in meters
>>> mix.mupol
... [0, 1.6609]  # dipolar moment in Debye
```

The mix object stores pure component molecular parameters and allows to set interaction corrections for the cross-association energy and the Mie potential interaction energy. Both pure component or fluid mixture can be modeled with SAFT-VR-Mie EoS, examples of how to obtain properties from the EoS are shown in the following sections.
7.1.3 SAFT-VR-Mie for pure fluids

The set up of pure components is shown here. The created object is passed to the SGTPy.saftvrmie() function, this function returns the SAFT-VR-Mie EoS object which is ready to do some basic computation such as, density, pressure, fugacity coefficient, chemical potential, as well as, some thermal derived properties such as residual heat capacities, entropy and enthalpy.

The set up the eos is straightforward, as shown bellow:

```python
>>> from SGTPy import component, saftvrmie

>>> methane = component('methane', ms = 1.0, sigma = 3.752 , eps = 170.75, 
                      lambda_r = 16.39, lambda_a = 6.)

>>> eos = saftvrmie(methane)
```

In the case of CG fluids the influence parameter can be correlated from the molecular parameters\(^1\) according to the following expression:

\[
\sqrt{\frac{c_{ii}}{N_a^2 \sigma^5}} = m_s(0.12008 + 2.21989\alpha)
\]

\[
\alpha = \frac{\lambda_r}{3(\lambda_r - 3)} \left(\frac{\lambda_r}{6}\right)^{6/\lambda_r - 6}
\]

```python
>>> # correlated influence parameter for methane
>>> eos.cii_correlation(overwrite=True)
... 1.9207509420744775e-20 # influence parameter in J m^5 / mol^2
```

The SGTPy.saftvrmie() function can identify the contributions present in the fluid. For a self-associating fluid:

```python
>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85, 
                     lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624, 
                     rdAB = 0.4, sites = [0,2,2], Mw = 18.01528, cii = 1.

...5371939421515458e-20)

>>> eos = saftvrmie(water)
```

The eos object includes useful calculations using SAFT-VR-Mie EoS. These methods work with the international system of units, that is, the temperature in Kelvin, pressure in Pascal, density in mol/m\(^3\). The density method can compute volume roots of liquid and vapor phases at given temperature and pressure. The method includes automatic initiation using Topliss’s method\(^2\), if an initial value is available the volume is computed with Newton’s method.

```python
>>> T = 300. # K
>>> P = 1e5 # Pa

>>> eos.density(T, P, 'L')
... 56938.970481192526 # mol/m3

>>> eos.density(T, P, 'V')
... 44.962772347755454 # mol/m3
```

The pressure method computes the pressure at given density and temperature. In the following codeblock, the previously computed roots are verified.

```python
>>> T = 300. # K
>>> rhov = 44.9627723477554836 # mol/m3
```

---

>>> eos.pressure(rhov, T)
... 99999.99999999964 # Pa
>>> rhol = 56938.970481192526 # mol/m3
>>> eos.pressure(rhol, T)
... 100000.00000008068 # Pa

The psat method allows to compute the saturation pressure at given temperature. It requires either an initial guess for the saturation pressure $P_0$ or for the volume of the phases $v_0$. >>> T = 300. # K >>> P0 = 1e3 # Pa >>> eos.psat(T, P0=P0) >>> # equilibrium pressure (Pa), liquid volume (mol/m3), vapor volume (mol/m3) ... (3640.841209122654, 1.756342718267362e-05, 0.6824190076059896)

The phase equilibria calculation can be verified through chemical potentials ($\mu_{ad}$) or through fugacity coefficients ($\log f_{ug}$)

>>> Psat = 3640.841209122654 # Pa
>>> vl = 1.756342718267362e-05 # m3/mol
>>> vv = 0.6824190076059896 # m3/mol
>>> np.allclose(eos.muad(1/vl, T), eos.muad(1/vv, T))
... True
>>> np.allclose(eos.logfug(T, Psat, 'L', v0=vl)[0], eos.logfug(T, Psat, 'V', v0=vv)[0])
... True

The eos object is also capable of computing other properties such as residual entropies. This is done with the EntropyR method.

>>> Sl = eos.EntropyR(T, Psat, 'L', v0=vl)
>>> Sv = eos.EntropyR(T, Psat, 'V', v0=vv)
>>> Sl - Sv
... -142.8199794928061 # J / mol K
>>> # NIST value = -146.3584 J / mol K

The calculation of residual enthalpies is done in the same manner using the EnthalpyR method.

>>> Hl = eos.EnthalpyR(T, Psat, 'L')
>>> Hv = eos.EnthalpyR(T, Psat, 'V')
>>> (Hl - Hv) / 1000.
... -42.84599384780775 # kJ / mol
>>> # NIST value = -43.9081 kJ / mol

For the calculation of heat capacities, only the residual contribution is computed from the SAFT-VR-Mie EoS using the CpR method. The ideal contribution can be obtained from correlations or data banks, such as DIPPR 801.

>>> R = 8.314 # J/mol K
>>> # Ideal Gas Heat Capacity by DIPPR
>>> k1=33363
>>> k2=26790
>>> k3=2610.5
>>> k4=8896
>>> k5=1169
>>> CpId = k1 + k2 * ((k3/T) /np.sinh(k3/T))**2
>>> CpId += k4 * ((k5/T) /np.cosh(k5/T))**2
>>> CpId /= 1000. # J / mol K
>>> CvId = CpId - R # J / mol K

(continues on next page)
Finally, the speed of sound of the phases can be computed using the `speed_sound` method. As this calculation requires the total isobaric and isochoric heat capacities, the ideal contribution must be supplied manually.

```
>>> eos.speed_sound(T, Psat, 'L', v0=vl, CvId=CvId, CpId=CpId)
... 1542.8100435020717 # m/s
>>> # NIST value = 1501.4 m/s
>>> eos.speed_sound(T, Psat, 'V', v0=vv, CvId=CvId, CpId=CpId)
... 427.1691887269907 # m/s
>>> # NIST value = 427.89 m/s
```

7.1.4 SAFT-VR-Mie for fluid mixtures

The set up of fluid mixtures is shown here. SAFT-VR-Mie EoS bases its unlike interactions using the following mixing rules to evaluate the size parameter ($\sigma_{ij}$), well depth ($\epsilon_{ij}$) and attractive and repulsive exponents ($\lambda_{ij}$).

$$\sigma_{ij} = \frac{\sigma_{ii} + \sigma_{jj}}{2}$$

$$\epsilon_{ij} = \left(1 - k_{ij}\right) \frac{\sqrt{\frac{\sigma_{i}^3 \sigma_{j}^3}{\sigma_{ij}^3}}}{\sqrt{\epsilon_{i} \epsilon_{j}}}$$

$$(\lambda_{k,ij} - 3) = \sqrt{\left(\lambda_{k,ii} - 3\right) \left(\lambda_{k,jj} - 3\right)} \quad ; \quad k = a, r$$

Here only the Mie potential interaction energy admits a binary interaction parameter $k_{ij}$, this is optimized from experimental phase equilibria data.

To use SAFT-VR-Mie EoS with fluid mixtures, a created mixture object is passed to the `SGTPy.saftvrmie()` function. This function returns the SAFT-VR-Mie EoS object and includes the needed method to compute properties such as density, pressure, residual heat capacities, entropy and enthalpy, chemical potential and fugacity coefficients. The set up the `eos` object is straightforward, as shown below:

```
>>> from SGTPy import component, mixture, saftvrmie
>>> methane = component('methane', ms = 1.0, sigma = 3.752, eps = 170.75,
... lambda_r = 16.39, lambda_a = 6.)
>>> dodecane = component('dodecane', ms = 4.0, sigma = 4.351, eps = 378.56,
... lambda_r = 18.41, lambda_a = 6.)
>>> mix = mixture(methane, dodecane)
>>> # interaction parameter optimized from phase equilibria data
>>> kij = -0.02199102576365056
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> # setting kij correction
>>> mix.kij_saft(Kij)
>>> eos = saftvrmie(mix)
```
As for the case of pure fluids, for mixtures of CG fluids, the influence parameter can be correlated from the molecular parameters\(^1\).
\[
\sqrt{\frac{\epsilon_{ii}}{N_{av}^2 \sigma_{ii}^6}} = m_{s,ii} (0.12008 + 2.21989 \alpha_{ii})
\]
\[
\alpha_{ii} = \frac{\lambda_{r,ii}}{3(\lambda_{r,ii} - 3)} \left( \frac{\lambda_{r,ii}}{6} \right)^{6/\lambda_{r,ii} - 6}
\]

```python
>>> eos.cii_correlation(overwrite = True)
... array([1.92075094e-20, 1.27211926e-18]) # influence parameter in J m5/mol2
```

For the case of cross-association the unlike site geometry \(r_{ABij}\) and energy \(\epsilon_{AB}^{ij}\) are obtained with the following mixing rules:
\[
r_c^{ABij} = \frac{r_c^{ABii} + r_c^{ABjj}}{2}
\]
\[
r_d^{ABij} = \frac{r_d^{ABii} + r_d^{ABjj}}{2}
\]
\[
\epsilon_{ij}^{AB} = (1 - l_{ij}) \sqrt{\epsilon_{ii}^{AB} \epsilon_{jj}^{AB}}
\]

The association energy admits a binary interaction parameter \(l_{ij}\), this parameter can be optimized alongside the \(k_{ij}\) parameter from experimental equilibria data.

```python
>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
...                    lambda_r = 8.308, lambda_a = 6. , eAB = 1991.07, rcAB = 0.5624,
...                    rdAB = 0.4, sites = [0,2,2], cii = 1.5371939421515458e-20)
>>> butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077 , eps = 277.892,
...                    lambda_r = 10.6689, lambda_a = 6. , eAB = 3300.0, rcAB = 0.2615,
...                    rdAB = 0.4, sites = [1,0,1], npol = 1.45, mupol = 1.6609,
...                    cii = 1.5018715324070352e-19)
>>> # optimized from experimental LLE
>>> kij, lij = np.array([-0.00736075, -0.00737153])
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> Lij = np.array([[0, lij], [lij, 0]])
>>> # setting interactions corrections
>>> mix.kij_saft(Kij)
>>> mix.lij_saft(Lij)
>>> eos = saftvrmie(mix)
```

Another possible scenario is when a fluid in the mixture doesn’t self-associate but can associate with other components in the mixture. This phenomenon occurs with mixtures of ethers and alcohols. A simple workaround of this induced association is to consider the cross association energy as half of the association energy of the self associating fluid and to optimize the geometry of the site. As in SAFT-VR-Mie the distance \(r_c^{AB}\) is usually set to \(0.4\sigma\) this implies that only \(r_c^{AB}\) is optimized.
\[
\epsilon_{ij}^{AB} = \frac{\epsilon^{AB}(self - associating)}{2}; \quad r_c^{ABij}(fitted)
\]

The ternary mixture of hexane, ethanol and CPME exhibit induced association. The mixture is created as usual.

>>> from SGTPy import component, mixture

>>> ethanol = component('ethanol2C', ms=1.7728, sigma=3.5592, eps=224.50,
  ... lambda_r=11.319, lambda_a=6., eAB = 3018.05, rcAB=0.3547,
  ... rdAB=0.4, sites=[1,0,1], cii=5.3141080872882285e-20, Mw=46.07)

>>> hexane = component('hexane', ms=1.96720036, sigma=4.54762477, eps=377.60127994,
  ... lambda_r=18.41193194, cii=3.581510586936205e-19, Mw=100.16)

>>> cpme = component('cpme', ms=2.32521144, sigma=4.13606074, eps=343.91193798, lambda_r=14.15484877,
  ... lambda_a=6.0, npol=1.91990385, mupol=1.27, sites=[0,0,1],
  ... cii=3.5213681817448466e-19, Mw=86.18)

>>> # creating mixture
>>> mix = mixture(hexane, ethanol)

>>> # adding a component
>>> mix.add_component(cpme)

The $k_{ij}$ binary correction corrections are set by pairs and the induced association is set by manually modifying the association energy matrix (eos.eABij) and the association range matrix (eos.rcij).

>>> # setting kij corrections
>>> k12 = 0.011818492037463553
>>> k13 = 0.0008700151297528677
>>> k23 = 0.01015194
>>> Kij = np.array([[0., k12, k13], [k12, 0., k23], [k13, k23, 0.]])
>>> mix.kij_saft(Kij)

>>> eos = saftvrmie(mix)

>>> # induced association set up
>>> rc = 2.23153033 # Angstrom
>>> eos.eABij[1,2] = ethanol.eAB / 2
>>> eos.eABij[2,1] = ethanol.eAB / 2
>>> eos.rcij[1,2] = rc * 1e-10
>>> eos.rcij[2,1] = rc * 1e-10

The eos object can be used for some basic properties calculations. The density of the mixture is computed with the density method at given composition, temperature and pressure. This method includes automatic initialization with Topliss's method\(^2\), on the other hand, when an initial guess is available it uses Newton's method.

>>> T = 350. # K
>>> P = 1e5 # Pa
>>> x = np.array([0.1, 0.3, 0.6])
>>> eos.density(x, T, P, 'L')
... 9311.834138400469 # liquid density in mol/m3
>>> eos.density(x, T, P, 'V')
... 35.458148587066376 # vapor density in mol/m3

The eos object also can compute the pressure of the mixture at given composition, density and temperature using the pressure method. In the following code block, the previously computed volume roots are verified.

>>> T = 350. # K
>>> x = np.array([0.1, 0.3, 0.6])
>>> rhol = 9311.834138400469 # mol/m3
>>> eos.pressure(x, rhol, T)
... 99999.99999504909 # Pa
The `eos` object is used for phase equilibria calculation through fugacity coefficient, this is done with the `logfugref` method. This function requires the composition, temperature and pressure and returns the computed fugacity coefficient and the compute volume root.

```python
>>> T = 350. # K
>>> P = 1e5 # Pa
>>> x = np.array([0.1, 0.3, 0.6])
>>> eos.logfugref(x, T, P, 'L')
>>> # fugacity coefficients and computed liquid volume root
... (array([ 0.77982905, 0.47877663, -0.79012744]), 0.00010739022894277775)
>>> eos.logfugref(x, T, P, 'V')
>>> # fugacity coefficients and computed vapor volume root
... (array([-0.02400476, -0.03146375, -0.03088407]), 0.02820226209906395)
```

For the calculation of interfacial properties using SGT, the `eos` object includes the calculation of the chemical potential using the `muad` method. This function requires the density vector and temperature as inputs and return the chemical potentials.

```python
>>> T = 350. # K
>>> x = np.array([0.1, 0.3, 0.6])
>>> rhol = 9311.834138400469 # mol/m3
>>> eos.muad(rhol*x, T)
... array([-1.2536145 , -0.45605463, -1.03181152])
>>> rhov = 35.458148587066376 # mol/m3
>>> eos.muad(rhov*x, T)
... array([-2.0574483 , -0.96629501, -0.27256815])
```

The `eos` object is also capable of computing thermal derived properties. In the following code block the residual entropy is computed with the `EntropyR` method, the residual enthalpy is computed with the `EnthalpyR` method and the residual heat capacity is computed with the `CpR` method. Finally, the speed of sound requires the total isobaric and isochoric heat capacities, for simplicity in this example the ideal gas contributions are considered as 5R/2 and 3R/2, respectively. Then the speed of sound is computed with the `speed_sound` method.

```python
>>> T = 350. # K
>>> P = 1e5 # Pa
>>> x = np.array([0.1, 0.3, 0.6])
>>> eos.EntropyR(x, T, P, 'L')
... -96.33127812255216 # Residual entropy in J/mol K
>>> eos.EnthalpyR(x, T, P, 'L')
... -34450.62328681776 # Residual enthalpy in J/mol
>>> eos.CpR(x, T, P, 'L')
... 58.66381738561176 # Residual heat capacity in J/mol K
>>> R = 8.314 # J / mol K
>>> CvId = 3*R/2
>>> CpId = 5*R/2
>>> eos.speed_sound(x, T, P, 'L', CvId=CvId, CpId=CpId)
... 1189.1342143755487 # speed of sound in m/s
```

A complete list of available calculations in SGTPy is found [here](#).
7.1.5 Available calculations in SGTPy

- Property calculation from SAFT-VR-Mie EoS

<table>
<thead>
<tr>
<th>Property</th>
<th>Available?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density</td>
<td>✓✓✓</td>
</tr>
<tr>
<td>Pressure</td>
<td>✓</td>
</tr>
<tr>
<td>Helmholtz free energy</td>
<td>✓</td>
</tr>
<tr>
<td>Chemical Potential</td>
<td>✓</td>
</tr>
<tr>
<td>Fugacity coefficient</td>
<td>✓</td>
</tr>
<tr>
<td>Helmholtz Free Energy</td>
<td>✓</td>
</tr>
<tr>
<td>Residual Entropy*</td>
<td>✓</td>
</tr>
<tr>
<td>Residual Enthalpy*</td>
<td>✓</td>
</tr>
<tr>
<td>Residual isochoric heat capacity*</td>
<td>✓</td>
</tr>
<tr>
<td>Residual isobaric heat capacity*</td>
<td>✓</td>
</tr>
<tr>
<td>Speed of sound*</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Warning:** *Temperature derivatives of Helmholtz free energy are computed numerically

- Phase Equilibria

<table>
<thead>
<tr>
<th>Phase equilibria</th>
<th>Available?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase stability (tpd minimization)</td>
<td>✓</td>
</tr>
<tr>
<td>TP flash</td>
<td>✓</td>
</tr>
<tr>
<td>Bubble points</td>
<td>✓</td>
</tr>
<tr>
<td>Dew points</td>
<td>✓</td>
</tr>
<tr>
<td>Liquid-Liquid Equilibria (multiplash)</td>
<td>✓</td>
</tr>
<tr>
<td>Vapor-Liquid-Liquid Equilibria (multiplash)</td>
<td>✓</td>
</tr>
<tr>
<td>HP flash</td>
<td></td>
</tr>
<tr>
<td>SP flash</td>
<td></td>
</tr>
</tbody>
</table>

- Square Gradient Theory solution methods

<table>
<thead>
<tr>
<th>Calculation method</th>
<th>Available?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure fluid</td>
<td>✓</td>
</tr>
<tr>
<td>Reference component (mixtures)</td>
<td>✓</td>
</tr>
<tr>
<td>Path function (mixtures)</td>
<td>✓</td>
</tr>
<tr>
<td>Orthogonal collocation (mixtures)</td>
<td>✓</td>
</tr>
<tr>
<td>Stabilized BVP (mixtures)</td>
<td>✓</td>
</tr>
</tbody>
</table>

Guidelines to reuse SGTPy functions

SGTPy includes functions to compute phase equilibria and interfacial properties using Square Gradient Theory. Those functions can be used with the included SAFT-VR-Mie EoS, but they will also work with any user-defined model as long as it meets the following requirements. The model (eos) should be an object with attributes and methods.
Model attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eos.nc</td>
<td>integer</td>
<td>Number of components in the mixture</td>
</tr>
<tr>
<td>eos.secondordersgt</td>
<td>bool</td>
<td>whether derivatives of the chemical potential are available through the eos.dmuad method</td>
</tr>
</tbody>
</table>

Pure fluid methods

- **eos.temperature_aux(T)**

Method that compute all the temperature dependent parameters and return them as a tuple.

*Input:* Temperature

*Return:* tuple (temp_aux)

- **eos.logfug_aux(temp_aux, P, state, v0, Xass0)**

Method that computes the natural logarithm of the fugacity coefficient for pure fluids (nc=1).

*Input:* temperature dependent parameters (temp_aux), pressure (P), state (‘L’ for liquid phase and ‘V’ for vapor phase), v0 is used as initial guess to compute the volume root, if None it should automatically initiate the calculation. Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

*Return:* natural logarithm of the fugacity coefficient (lnphi), computed volume root (v) and computed fraction of non-bonded sites (Xass).

- **eos.muad_aux(rho, temp_aux, Xass0)**

Method that computes chemical potential for pure fluids (nc=1).

*Input:* density (rho), temperature dependent parameters (temp_aux). Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

*Return:* chemical potential (mu) and computed fraction of nonbonded sites (Xass).

- **eos.a0ad_aux(rho, temp_aux, Xass0)**

Method that computes the Helmholtz density free energy for pure fluids (nc=1).

*Input:* density (rho), temperature dependent parameters (temp_aux). Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

*Return:* Helmholtz density free energy (a0ad) and computed fraction of nonbonded sites (Xass).

- **eos.dOm_aux(rho, temp_aux, mu0, P, Xass0)**

Method that computes the delta of the Grand Thermodynamic Potential for pure fluids (nc=1).

*Input:* density (rho), temperature dependent parameters (temp_aux), equilibrium chemical potential at given temperature (mu0), equilibrium pressure (P). Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

*Return:* Grand Thermodynamic Potential (dOm) and computed fraction of nonbonded sites (Xass).
• eos.sgt_adim(T)
Method that computes the factor to make SGT calculations consistent.

Input: Temperature

Return: temperature factor (Tfactor), pressure factor (Pfactor), density factor (rofactor), interfacial tension factor (ten-factor), interfacial length factor (zfactor)

Fluid mixtures methods

• eos.temperature_aux(T)
Method that compute all the temperature dependent parameters and return them as a tuple.

Input: Temperature

Return: tuple (temp_aux)

• eos.logfugef_aux(x, temp_aux, P, state, v0, Xass0)
Method that computes the natural logarithm of the effective fugacity coefficient for mixtures (nc>=2).

Input: composition (x), temperature dependent parameters (temp_aux), pressure (P), state (‘L’ for liquid phase and ‘V’ for vapor phase), v0 is used as initial guess to compute the volume root, if None it should automatically initiate the calculation. Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

Return: natural logarithm of the fugacity coefficient (lnphi), computed volume root (v) and computed fraction of nonbonded sites (Xass).

• eos.muad_aux(rhoi, temp_aux, Xass0)
Method that computes the chemical potential for mixtures (nc>=2).

Input: density vector (rhoi), temperature dependent parameters (temp_aux). Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

Return: chemical potential (mu) and computed fraction of nonbonded sites (Xass).

• eos.dmuad_aux(rhoi, temp_aux, Xass0)
Method that computes the chemical potential and its derivatives matrix (d mu_i / d rho_j) for mixtures (nc>=2).

Input: density vector (rhoi), temperature dependent parameters (temp_aux). Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

Return: chemical potential (mu), its derivatives matrix (dmu) and computed fraction of nonbonded sites (Xass).

• eos.a0ad_aux(rhoi, temp_aux, Xass0)
Method that computes the Helmholtz density free energy for pure for mixtures (nc>=2).

Input: density vector (rhoi), temperature dependent parameters (temp_aux). Xass0 is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

Return: Helmholtz density free energy (a0ad) and computed fraction of nonbonded sites (Xass).

• eos.dOm_aux(rhoi, temp_aux, mu0, P, Xass0)
Method that computes the delta of the Grand Thermodynamic Potential for pure fluids (nc=1).

**Input:** density vector \( \rho_i \), temperature dependent parameters \( \text{temp}_{\text{aux}} \), equilibrium chemical potential at given temperature \( \mu_0 \), equilibrium pressure \( P \), \( X_{\text{ass}0} \) is used as initial guess to compute the fraction of nonbonded sites, None should be used for non-associating mixtures or to automatically initiate the calculation.

**Return:** Grand Thermodynamic Potential \( d\Omega_m \) and computed fraction of nonbonded sites \( X_{\text{ass}} \).

\[
\text{• eos.sgt_adim}(T)
\]
Method that computes the factor to make SGT calculations consistent.

**Input:** Temperature

**Return:** temperature factor \( T_{\text{factor}} \), pressure factor \( P_{\text{factor}} \), density factor \( \rho_{\text{factor}} \), interfacial tension factor \( \tau_{\text{factor}} \), interfacial length factor \( z_{\text{factor}} \)

\[
\text{• eos.ci}(T)
\]
Method that computes the influence parameter matrix for mixtures.

**Input:** Temperature

**Return:** influence parameter matrix \( c_{ij} \)

### 7.2 SGTPy.equilibrium

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

\[
T^\alpha = T^\beta = \ldots = T^\pi \quad P^\alpha = P^\beta = \ldots = P^\pi \quad \mu_i^\alpha = \mu_i^\beta = \ldots = \mu_i^\pi \quad i = 1, \ldots, c
\]

Where, \( T, P \) and \( \mu \) are the temperature, pressure and chemical potential. When working with EoS usually equilibrium is guaranteed by fugacity coefficients:

\[
x^\alpha_i \hat{\phi}_i^\alpha = x^\beta_i \hat{\phi}_i^\beta = \ldots = x^\pi_i \hat{\phi}_i^\pi \quad i = 1, \ldots, c
\]

Usual equilibrium calculations include vapor-liquid equilibrium (flash, bubble point, dew point), liquid-liquid equilibrium (flash and stability test) and vapor-liquid-liquid equilibrium (multiflash and stability test). Those algorithms are described in the following sections:

#### 7.2.1 Two phase Flash

This is the most classical calculation of phase equilibria. Temperature, pressure and global composition of a system are known. If the mixture is unstable it will split into two or more phases. When trying to compute two-phase separation the flash algorithm can be used. The usual approach to solve this problem is to solve the following mass balance and then update composition by successive substitution.

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

Where, \( z \) is the global composition of component \( K = x^\beta / x^\alpha \) are the equilibrium constant and \( \psi \) is the phase fraction of phase \( \beta \). Subscript refers to component index and superscript refers to the phase index. This method can be slow at...
high pressures, and in those cases, the algorithm changes to a second-order minimization of the Gibbs free energy of the system:

\[
\min G(F^\alpha, F^\beta) = \sum_{i=1}^{\alpha} (F^\alpha_i \ln \hat{f}^\alpha_i + F^\beta_i \ln \hat{f}^\beta_i)
\]

Where, \( F \) is the mole number and \( \hat{f} \) is the effective fugacity.

In the following code block a flash calculation for vapor-liquid equilibria is shown:

```python
>>> from SGTPy import component, mixture, saftvrmie
>>> from SGTPy.equilibrium import flash

>>> water = component('water', ms=1.7311, sigma=2.4539, eps=110.85,
... lambda_r=8.308, lambda_a=6., eAB=1991.07, rcAB=0.5624,
... rdAB=0.4, sites=[0,2,2], cii=1.5371939421515455e-20)
>>> ethanol = component('ethanol2C', ms=1.7728, sigma=3.5592, eps=224.50,
... lambda_r=11.319, lambda_a=6., eAB=3018.05, rcAB=0.3547,
... rdAB=0.4, sites=[1,0,1], cii=5.3141087288285e-20)
>>> mix = mixture(ethanol, water)
>>> kij, lij = np.array([-0.0069751, -0.01521566])
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> Lij = np.array([[0., lij], [lij, 0]])
>>> # setting interactions corrections
>>> mix.kij_saft(Kij)
>>> mix.lij_saft(Lij)
>>> # creating eos model
>>> eos = saftvrmie(mix)
>>> # flash conditions
>>> T = 355.  # K
>>> P = 1e5  # Pa
>>> z = np.array([0.4, 0.6])  # global composition
>>> # initial guesses
>>> x0 = np.array([0.1, 0.9])
>>> y0 = np.array([0.8, 0.2])
>>> # flash(x0, y0, 'LV', z, T, P, eos)
>>> # x, y, psi
... (array([0.2636008, 0.7363992]), array([0.55807925, 0.44192075]), 0.4631890692698939)
```

The same algorithm can be applied for liquid-liquid equilibria, as can be seen:

```python
>>> water = component('water', ms=1.7311, sigma=2.4539, eps=110.85,
... lambda_r=8.308, lambda_a=6., eAB=1991.07, rcAB=0.5624,
... rdAB=0.4, sites=[0,2,2], cii=1.5371939421515455e-20)
>>> butanol = component('butanol2C', ms=1.9651, sigma=4.1077, eps=277.892,
... lambda_r=10.6689, lambda_a=6., eAB=3300.0, rcAB=0.2615,
... rdAB=0.4, sites=[1,0,1], npol=1.45, mupol=1.6609,
... cii=1.5018715324070352e-19)
>>> mix = mixture(water, butanol)
>>> # optimized from experimental LLE
>>> kij, lij = np.array([-0.00736075, -0.00737153])
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> Lij = np.array([[0., lij], [lij, 0]])
>>> # setting interactions corrections
>>> mix.kij_saft(Kij)
```
>>> mix.lij_sfa(Lij)
>>> # creating eos model
>>> eos = saftvrmie(mix)
>>> # LLE conditions
>>> T = 298.15  # K
>>> P = 1.01325e5  # Pa
>>> # global composition
>>> z = np.array([0.8, 0.2])
>>> # initial composition of the liquid phases
>>> x0 = np.array([0.9, 0.1])
>>> w0 = np.array([0.6, 0.4])
>>> # LLE is performed as a flash
>>> flash(x0, w0, 'LL', z, T, P, eos)
... (array([0.96022175, 0.03977825]), array([0.53375333, 0.46624667]), 0.
... 3756942886430804)

7.2.2 Vapor liquid equilibrium

The two-phase flash can be used to compute vapor-liquid equilibria at fixed temperature and pressure. When dealing with saturated liquids or vapor four types of problems arise, solution methods and routines are related to a simplification of the Radford-Rice mass balance, as the phase fraction is already known.

Another option for these situations is to solve the following system of equations:

\[
\begin{align*}
    f_i &= \ln K_i + \ln \gamma_i^v(y_i, T, P) - \ln \gamma_i^l(x_i, T, P) \\
    f_{c+1} &= \sum_{i=1}^{c} (y_i - x_i)
\end{align*}
\]

**bubble points**

In this case, a saturated liquid of known composition is forming a differential size bubble. If the pressure is specified, the temperature must be found. Similarly, when the temperature is specified, equilibrium pressure has to be calculated.

The usual approach for solving this problem consists of a combined quasi-Newton for solving for temperature or pressure and successive substitution for composition with the following simplification of the Radford-Rice equation:

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

In the case of having a good initial value of the true equilibrium values, the full multidimensional system of equations can be solved.

In the following code block and example from this computation, it is shown.

```python
>>> from SGTPy import component, mixture, saftvrmie
>>> from SGTPy.equilibrium import bubbleTy

>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592 , eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii= 5.3141080872882285e-20)
>>> hexane = component('hexane', ms = 1.96720036, sigma = 4.54762477, eps = 377.60127994,
... lambda_r = 18.41193194, cii = 3.581510586936205e-19)
```
>>> mix = mixture(hexane, ethanol)
>>> # fitted to experimental data
>>> kij = 0.011818492037463553
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> mix.kij_saft(Kij)
>>> eos = saftvrmie(mix)
>>> # bubble point conditions
>>> P = 1.01325e5  # Pa
>>> x = np.array([0.2, 0.8])
>>> # initial guess for temperature and vapor composition
>>> T0 = 320.
>>> y0 = np.array([0.8, 0.2])
>>> bubbleTy(y0, T0, x, P, eos)
>>> # vapor composition and equilibrium temperature
... (array([0.58026242, 0.41973758]), 333.45268641828727)

In the following case, a saturated liquid of known composition and temperature is forming a differential size bubble. We need to find the composition and equilibrium pressure.

```python
>>> from SGTPy.equilibrium import bubblePy
>>> # bubble point conditions
>>> T = 350.  # K
>>> x = np.array([0.2, 0.8])
>>> # initial guess for temperature and vapor composition
>>> P0 = 1e5  # Pa
>>> y0 = np.array([0.8, 0.2])
>>> bubblePy(y0, P0, x, T, eos)
>>> # vapor composition and equilibrium pressure
... (array([0.52007469, 0.47992531]), 178461.90299494946)
```

dew points

In this case, a saturated vapor of known composition and temperature is forming a differential size dew. We need to find the composition and equilibrium pressure.

The usual approach for solving this problem consists of a combined quasi-Newton for solving for Pressure and successive substitution for composition with the following simplification of the Radford-Rice equation:

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

In the case of having a good initial value of the true equilibrium values, a full multidimensional system can be solved.

In the following code block and example from this computation, it is shown for composition and equilibrium pressure.

```python
>>> from SGTPy.equilibrium import dewPx
>>> ethanol = component('ethanol12C', ms = 1.7728, sigma = 3.5592, eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii= 5.3141080872882285e-20)
>>> cpme = component('cpme', ms = 2.32521144, sigma = 4.13606074, eps = 343.91193798,
... lambda_r = 14.15484877,
... lambda_a = 6.0, npol = 1.91990385,mupol = 1.27, sites =[0,0,1], cii= 3.5213681817448466e-19)
```
```python
>>> mix = mixture(ethanol, cpme)
>>> kij = 0.01015194
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> mix.kij_saft(Kij)
>>> eos = saftvrmie(mix)
>>> # induced association set up
>>> rc = 2.23153033 # Angstrom
>>> eos.eABij[0,1] = ethanol.eAB / 2
>>> eos.eABij[1,0] = ethanol.eAB / 2
>>> eos.rcij[0,1] = rc * 1e-10
>>> eos.rcij[1,0] = rc * 1e-10
>>> # dew point conditions
>>> T = 350. # K
>>> y = np.array([0.4, 0.6])
>>> # initial guess for temperature and liquid composition
>>> P0 = 1e5 # Pa
>>> x0 = np.array([0.2, 0.8])
>>> dewPx(x0, P0, y, T, eos)
>>> # vapor composition and equilibrium pressure
... (array([0.10431595, 0.89568405]), 62927.01280107427)

Similarly, the calculation can be carried out for equilibria composition and temperature:

```python
>>> from SGTPy.equilibrium import dewTx
>>> # dew point conditions
>>> P = 1.01325e5 # Pa
>>> y = np.array([0.4, 0.6])
>>> # initial guess for temperature and liquid composition
>>> T0 = 350.
>>> x0 = np.array([0.2, 0.8])
>>> dewTx(x0, T0, y, P, eos)
>>> # vapor composition and equilibrium temperature
... (array([0.10611092, 0.89388908]), 364.3596395673508)
```

### 7.2.3 Multiphase Flash

Meta-stable solutions of the isofugacity method is an important concern when dealing with more than two liquid phases. Stability verification during the equilibria computation must be performed. In SGTPy liquid-liquid equilibria and vapor-liquid-liquid equilibria are solved similarly with a modified Radford-Rice mass balance system of equations that allows verifying the stability and equilibria of the phases simultaneously.

\[
\sum_{i=1}^{c} \frac{z_i(K_{ik} \exp(\theta_k - 1))}{1 + \sum_{j=1}^{\pi} \sum_{j \neq r} \psi_j(K_{ij} \exp(\theta_j - 1))} = 0 \quad k = 1, ..., \pi, k \neq r
\]

This system of equations was proposed by Gupta et al, and it is a modified Radford-Rice mass balance that introduces stability variables \( \theta \). This allows to solve the mass balance for phase fraction and stability variables and then update composition similar to a regular two-phase flash. The stability variable gives information about the phase if it takes a positive value the phase is unstable, on the other hand, if it is zero then the phase is stable. The algorithm of successive substitution and Newton method can be slow in some cases, in that situation the function will attempt to minimize the
Gibbs free energy of the system.

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

**Liquid Liquid Equilibrium**

The two-phase flash can be used for solving liquid-liquid equilibrium, but it is important to consider the stability of the phases. For that reason, an algorithm that can compute stability and equilibrium simultaneously was implemented in this package.

In the following code block and an example of how to solve this problem, it is shown.

```python
>>> import numpy as np
>>> from SGTPy import component, mixture, saftvrmie
>>> from SGTPy.equilibrium import lle

>>> water = component('water', ms = 1.7311, sigma = 2.4539, eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2], cii = 1.5371939421515458e-20)

>>> butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077, eps = 277.892,
... lambda_r = 10.6689, lambda_a = 6., eAB = 3300.0, rcAB = 0.2615,
... rdAB = 0.4, sites = [1,0,1], npol = 1.45, mupol = 1.6609,
... cii = 1.5018715324070352e-19)

>>> mix = mixture(water, butanol)

>>> # optimized from experimental LLE
>>> kij, lij = np.array([-0.00736075, -0.00737153])
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> Lij = np.array([[0., lij], [lij, 0]])

>>> # setting interactions corrections
>>> mix.kij_saft(Kij)
>>> mix.lij_saft(Lij)

>>> # creating eos model
>>> eos = saftvrmie(mix)

>>> # LLE conditions
>>> T = 298.15 # K
>>> P = 1.01325e5 # Pa

>>> # global composition
>>> z = np.array([0.8, 0.2])

>>> # initial composition of the liquid phases
>>> x0 = np.array([0.9, 0.1])
>>> w0 = np.array([0.6, 0.4])

>>> # LLE is performed as a flash that search stable phases
>>> lle(x0, w0, z, T, P, eos, full_output=False)
... #(array([0.96022175, 0.03977825]), array([0.53375333, 0.46624667]), 0.
... 37569428828046253)
```
Three phase equilibrium

Binary mixtures

For degrees of freedom’s restriction, a system of equations has to be solved for three phase equilibrium of binary mixtures. In the following code block, an example of how to do it is shown.

```python
>>> from SGTPy.equilibrium import vlleb

>>> P = 1.01325e5  # Pa
>>> # initial guesses
>>> x0 = np.array([0.96, 0.06])
>>> w0 = np.array([0.53, 0.47])
>>> y0 = np.array([0.8, 0.2])
>>> T0 = 350.  # K
>>> vlleb(x0, w0, y0, T0, P, 'P', eos, full_output=False)
... # (array([0.94267074, 0.05732926]), array([0.61296229, 0.38703771]),
... # array([0.77529982, 0.22470018]), array([367.2045658]))
```

Multicomponent mixtures

When working with multicomponent mixtures (3 or more) a multiflash has to be performed in order to compute three-phase equilibrium. This algorithm ensures that stable phases are computed.

```python
>>> import numpy as np
>>> from SGTPy import component, mixture, saftvrmie
>>> from SGTPy.equilibrium import vlle

>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... npol = 2.95094686, mupol = 1.3611, cii = 3.5779968517655445e-19)

>>> butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077 , eps = 277.892,
... lambda_r = 10.6689, lambda_a = 6., eAB = 3300.0, rcAB = 0.2615,
... npol = 1.45, mupol = 1.6609, cii = 1.5018715324070352e-19)

>>> mtbe = component('mtbe', ms =2.17847383, sigma= 4.19140014, eps = 306.52083841,
... lambda_r = 14.74135198, lambda_a = 6.0, npol = 2.95094686, 
... mupol = 1.3611, cii = 1.5018715324070352e-19)

>>> mix = mixture(water, butanol)
>>> mix.add_component(mtbe)
>>> #butanol water
>>> k12, l12 = np.array([-0.00736075, -0.00737153])
>>> # mtbe butanol
>>> k23 = -0.0029995
>>> l23 = 0.
>>> rc23 = 1.90982649
>>> # mtbe water
>>> k13 = -0.07331438
>>> l13 = 0.
>>> rc13 = 2.84367922
>>> # setting up interaction corrections
>>> Kij = np.array([[0., k12, k13],[k12, 0., k23],[k13, k23, 0.]])
>>> Lij = np.array([[0., l12, l13],[l12, 0., l23],[l13, l23, 0.]])
>>> mix.kij_saft(Kij)
```
>>> mix.lij_saft(Lij)
>>> eos = saftvrmie(mix)
>>> # setting up induced association
>>> # mtbe water
>>> eos.eABij[0,2] = water.eAB / 2
>>> eos.eABij[2,0] = water.eAB / 2
>>> eos.rcij[0,2] = rc13 * 1e-10
>>> eos.rcij[2,0] = rc13 * 1e-10
>>> # mtbe butanol
>>> eos.eABij[2,1] = butanol.eAB / 2
>>> eos.eABij[1,2] = butanol.eAB / 2
>>> eos.rcij[2,1] = rc23 * 1e-10
>>> eos.rcij[1,2] = rc23 * 1e-10
>>> # three phase equilibria conditions
>>> T = 345.  #K
>>> P = 1.01325e5  # Pa
>>> # global composition
>>> z = np.array([0.5, 0.3, 0.2])
>>> # initial guesses
>>> x0 = np.array([0.9, 0.05, 0.05])
>>> w0 = np.array([0.45, 0.45, 0.1])
>>> y0 = np.array([0.3, 0.1, 0.6])
>>> vlle(x0, w0, y0, z, T, P, eos, full_output = False)
>>> # phase compositions
... (array([0.96430196, 0.03056118, 0.00513686]), array([0.44365858, 0.40405065, 0.15229077]),
... array([0.32687062, 0.06433222, 0.60879716]))

7.2.4 Stability

Stability analysis plays a fundamental role during phase equilibria computation. Most stability tests are based on the fact that a consistent equilibrium must minimize the Gibbs free energy of the system at a given temperature and pressure. Within this idea Michelsen proposed the tangent plane distance function which allows testing the relative stability of a mixture at given composition \( z \), temperature \( T \), and pressure \( P \).

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

The tpd function is evaluated for a trial composition \( w \) and if the tpd takes a negative value it implies that the energy of the system decreased with the formation of the new phase, i.e. the original phase was unstable. In order to test the stability of a mixture, the usual method is to find a minimum of the function and verify the sign of the tpd function at the minimum. Minimization recommendations for this purpose were given by Michelsen and they are included in Phasepy’s implementation.
Minimization of \( \text{tpd} \) function

As this is an iterative process, in order to find a minimum an initial guess of it has to be supplied. In the following code block, the stability of a liquid mixture is tested against the formation of another liquid.

```python
>>> import numpy as np
>>> from SGTPy import component, mixture, saftvrmie
>>> from SGTPy.equilibrium import lle

>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2], cii = 1.5371939421515458e-20)

>>> butanol = component('butanol2C', ms = 1.9651, sigma = 4.1077 , eps = 277.892,
... lambda_r = 10.6689, lambda_a = 6., eAB = 3300.0, rcAB = 0.2615,
... rdAB = 0.4, sites = [1,0,1], npol = 1.45, mupol = 1.6609,
... cii = 1.5018715324070352e-19)

>>> mix = mixture(water, butanol)

>>> # optimized from experimental LLE
>>> kij, lij = np.array([-0.00736075, -0.00737153])

>>> Kij = np.array([[0, kij], [kij, 0]])

>>> Lij = np.array([[0., lij], [lij, 0]])

>>> # setting interactions corrections
>>> mix.kij_saft(Kij)

>>> mix.lij_saft(Lij)

>>> # creating eos model
>>> eos = saftvrmie(mix)

>>> T = 320  # K
>>> P = 1.01e5  # Pa
>>> z = np.array([0.8, 0.2])

>>> tpd_min(w, z, T, P, eos, stateW = 'L', stateZ = 'L')

As the \( \text{tpd} \) value at the minimum is negative it means that the phase is unstable at will split into two liquids. Similarly, the relative stability can be tested against a vapor phase, in which case is found that the original phase was more stable than the vapor.

```python
>>> w = np.array([0.99, 0.01])

>>> tpd_min(w, z, T, P, eos, stateW = 'L', stateZ = 'L')

Finding all minimums

Sometimes might be useful to find all the minimums of a given mixture, for that case phasepy will try to find them using different initial guesses until the number of requested minimums is found. In the next example, three minimums were requested.

```python
>>> from SGTPy.equilibrium import tpd_minimas

>>> T = 320  # K
>>> P = 1.01e5  # Pa

(continues on next page)
Similar to the first example, all the minimums in vapor phase can be found, in this case there only one minimum.

```
>>> tpd_minimas(nmin, z, T, P, model, 'V', 'L')
>>> #composition of minimums found and tpd values
... ((array([0.82414939, 0.17585061]), array([0.82414939, 0.17585061]), array([0.82414939, 0.17585061]))
... array([0.86629349, 0.86629349, 0.86629349]))
```

Liquid liquid equilibrium initiation

Using the same principles stated above, tpd function can be used to generate initial guesses for liquid-liquid equilibria, the function ell_init allows to find two minimums of the mixture.

```
>>> from SGTPy.equilibrium import lle_init
>>> T = 320 # K
>>> P = 1.01e5 # Pa
>>> z = np.array([0.8, 0.2])
>>> lle_init(z, T, P, eos)
>>> #initial values for lle computation
... (array([0.95593125, 0.04406875]), array([0.55571917, 0.44428083]))
```

7.3 SGTPy.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 \(j\), \(\rho\) is density vector and \(z\) is the length 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:

\[ \sum_j c_{ij} \frac{d^2\rho_j}{dz^2} = \mu_i - \mu^0_i \quad i = 1, ..., c \]

\[ \rho(z \to -\infty) = \rho^0 \quad \rho(z \to \infty) = \rho^\beta \]

Here \(\mu\) represents the chemical potential and the superscript indicates its value evaluated at the bulk phase. \(\alpha\) and \(\beta\) are the bulk phases index.
Once the density profiles were solved the interfacial tension, $\sigma$, 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 on if you are working with a pure component or a mixture. In the latter, the correction value of $\beta_{ij}$ 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 using the density as an independent variable.

$$\sigma = \sqrt{2} \int_{\rho^0}^{\rho} \sqrt{c_r \Delta \Omega} d\rho$$

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

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

Where $P^0$ is the equilibrium pressure.

In SGTPy this integration is done using orthogonal 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.

```python
>>> from SGTPy import component, saftvrmie
>>> # The pure component is defined with the influence parameter
>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2], cii = 1.5371939422641703e-20)
>>> eos = saftvrmie(water)
```

First, vapor-liquid equilibria have 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.

```python
>>> from SGTPy import sgt_pure
>>> T = 373. # K
>>> P0 = 1e5 # Pa
>>> P, vl, vv = eos.psat(T, P0=P0)
>>> rhol = 1/vl
>>> rhov = 1/vv
>>> sgt_pure(rhol, rhov, T, P, eos)
```

Optionally, `full_output` allows getting all the computation information as the density profile, interfacial length and grand thermodynamic potential.

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

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

When working with mixtures, SGT solution procedure depends on whether the influence 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 have to be used. One of the options is the reference component method, which is the most popular. For this method the following system of equations has to be solved:

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

7.3. SGTPy.sgt
Where the subscript $r$ refers to the reference component and $j$ to the other components present in the mixture. Although the 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, 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 an auxiliary variable $\alpha = (\mu_i - \mu_r)/\sqrt{c_i}$. 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:

```python
>>> from SGTPy import component, mixture, saftvrmie
>>> from SGTPy.equilibrium import flash
>>> water = component('water', ms = 1.7311, sigma = 2.4539, eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2], cii = 1.5371939421515455e-20)
>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592, eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii = 5.3141080872882285e-20)
>>> mix = mixture(ethanol, water)
>>> kij, lij = np.array([-0.0069751, -0.01521566])
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> Lij = np.array([[0, lij], [lij, 0]])
>>> # setting interactions corrections
>>> mix.kij_saft(Kij)
>>> mix.lij_saft(Lij)
>>> # creating eos model
>>> eos = saftvrmie(mix)
>>> T = 298.15 # K
>>> x = np.array([0.2, 0.8])
>>> P0 = 1e4 # Pa
>>> y0 = np.array([0.8, 0.2])
>>> sol = bubblePy(y0, P0, x, T, eos, full_output=True)
>>> y, P = sol.Y, sol.P
>>> vl, vv = sol.v1, sol.v2
>>> #computing the density vectora
>>> rhol = x/vl
>>> rhov = y/vv
```

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

```python
>>> #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, eos, s=0, method='reference', full_output=True)
>>> #water doesn't show surface activity across the interface
>>> #and the density profiles are fully calculated
```
>>> solr2 = sgt_mix_beta0(rhov, rhol, T, P, eos, s=1, method='reference', full_output=True)
>>> #Using Liang path function the density profiles are computed directly
>>> soll = sgt_mix_beta0(rhov, rhol, T, P, eos, n=300, method='liang', full_output=True)

The following results are obtained from each method, it can be seen that the results from the path function and with the reference component method using component two as reference are the same.

- Reference component method (1) : 14.698312812670265 mN/m
- Reference component method (2) : 32.94702316866835 mN/m
- Liang path Function : 32.944051807272245 mN/m

The density profiles computed from each method are plotted in the following figure. Black line was computed with reference component 1, Cyan line was computed with reference component 2 and Liang path function results are plotted with a red line.

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

>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592, eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii= 5.314080872882285e-20)
>>> hexane = component('hexane', ms = 1.96720036, sigma = 4.54762477, eps = 377.60127994,
... lambda_r = 18.41193194, cii = 3.581510586936205e-19)
>>> mix = mixture(hexane, ethanol)
>>> kij = 0.011818492037463553
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> eos = saftvrmie(mix)
>>> T = 298.15  # K
>>> x = np.array([0.3, 0.7])
>>> y0 = 1.*x
>>> P0 = 8000.  # Pa
>>> sol = bubblePy(y0, P0, x, T, eos, full_output=True)
>>> y, P = sol.Y, sol.P
>>> vl, vv = sol.v1, sol.v2

(continues on next page)
Similar to the first example, all the possible methods will be tested.

```python
>>> rhox = x/vl
>>> rhoy = y/vv
```

The interfacial tension results from each method are listed below.

- Reference component method (1) : 9.550416968670147 mN/m
- Reference component method (2) : 14.807217440751549 mN/m
- Liang path Function: 16.826107984574268 mN/m

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

Clearly, only Liang’s path function was able to compute the density profiles of the mixture. An inspection on the $\alpha$ parameter from Liang’s path function reveals that its final value is zero, when this is not fulfilled Liang states that the geometric mean rule might no be suitable for the mixture and a $\beta_{ij}$ correction should be applied.

Individual functions for each method can be accessed through the SGTPy.sgt.ten_beta0_reference for reference component method and SGTPy.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 length.

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

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

In SGTPy 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 length, then it computes the interfacial tension. After this first iteration, the interfacial length 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 within a given tolerance (default value 0.01 mN/m). This procedure is inspired by the work of Liang and Michelsen.

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

```python
>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592, eps = 224.50, ...
...     lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547, ...
...     rdAB = 0.4, sites = [1,0,1], cii= 5.3141080872882285e-20)
>>> hexane = component('hexane', ms = 1.96720036, sigma = 4.54762477, eps = 377.60127994, ...
...     lambda_r = 18.41193194, cii = 3.581510586936205e-19)
>>> mix = mixture(hexane, ethanol)
>>> kij = 0.011818492037463553
>>> Kij = np.array([[0, kij], [kij, 0]])
>>> mix.kij_saft(Kij)
>>> eos = saftvrmie(mix)
>>> T = 320 #K
>>> X = np.array([0.3, 0.7])
>>> P0 = 3e4 # Pa
>>> 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 $\beta_{ij}$ has to be supplied to the eos with `eos.beta_sgt` method. Otherwise, the influence parameter matrix will be singular and an error will be raised.

```python
>>> bij = 0.05719272059410664
>>> beta = np.array([[0, bij], [bij, 0]])
>>> eos.beta_sgt(beta)
```

Then the interfacial tension can be computed as follows:

```python
>>> sgt_mix(rhol, rhov, T, P, eos, z0 = 10., rho0 = 'hyperbolic', full_output = False)
>>> #interfacial tension in mN/m
... 15.062218899775173
```

In the example $z0$ refers to the initial interfacial length, $rho0$ 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 transformation. Another 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 \( \text{rho0} \), this object is usually obtained from another SGT computation, as from a calculation with \( \beta_{ij} = 0 \).

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

```
>>> sol = sgt_mix(rhol, rhov, T, P, eos, z0 = 10., rho0 = 'hyperbolic', full_output = False)
>>> sol.tension
... 15.0621889775173 #IFT in mN/m
>>> #density profiles and spatial coordinate access
>>> sol.rho
>>> sol.z
```

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.

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

\[
\rho(z \to 0) = \rho^0 \quad \rho(z \to L) = \rho^3
\]

This differential equation is advanced in time until no further changes are found on the density profiles. Then the interfacial tension is computed. Its use is similar to 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
... 15.0621889775173 #IFT in mN/m
```

The density profiles obtained from each method are shown 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.
7.4 SGTPy.fit

In order to compute phase equilibria and interfacial properties it is necessary to count with the molecular parameters of pure component parameters as: the numbers of segments \( m_s \), well-depth of Mie potential \( \epsilon \), size parameter of Mie potential \( \sigma \), attractive \( \lambda_a \) and repulsive \( \lambda_r \) exponents of Mie Potential. For the case of pure self-associating fluid, three extra parameters are needed: the association energy \( \epsilon_{AB} \) in K units, the association range \( r_{AB} \) and association center position \( r_{AB}c \). In the case of polar fluids, this contribution requires the number of polar sites \( n_p \). Additionally, to model the fluid with SGT the influence parameter is required.

In the case of mixtures, the \( k_{ij} \) and \( l_{ij} \) binary interaction parameters are required as the optimized \( r_{ABij} \) for induced associating mixtures.

SGTPy includes several functions that rely on equilibria routines included in the package and in SciPy optimization tools for fitting model parameters. These functions are explained in the following sections for pure components and for mixtures.

7.4.1 Pure component data

Molecular EoS, like SAFT-VR-Mie, can be parametrized using experimental saturation data (pressure and densities). Additionally, thermal derived properties can be included to improve the modeling of the fluid, this data includes heat capacities, vaporization enthalpy and speed of sound. The experimental data can be obtained from NIST database, DIPPR 801, TDE, Knovel or by your own measurements.

In the following example, the molecular parameters of n-hexane will be fitted. First, the saturation pressure and density, as well as, the speed of sound and enthalpies are obtained from NIST database.

```python
>>> import numpy as np
>>> # Experimental data obtained from NIST WebBook
>>> Tsat = np.array([290., 300., 310., 320., 330., 340., 350., 360.]) # K
>>> Psat = np.array([14016., 21865., 32975., 48251., 68721., 95527., 129920., 173260.]) # Pa
>>> rhol = np.array([7683.6, 7577.4, 7469.6, 7360.1, 7248.7, 7135.0, 7018.7, 6899.5]) # mol/m3
>>> rhov = np.array([5.8845, 8.9152, 13.087, 18.683, 26.023, 35.466, 47.412, 62.314]) # mol/m3
>>> # speed of sound
>>> speed_liq = np.array([1090.6, 1047.6, 1005.1, 963.05, 921.37, 880.0, 838.85, 797.86]) # m/s
>>> speed_vap = np.array([170.71, 172.79, 174.62, 176.17, 177.42, 178.33, 178.87, 179.01]) # m/s
>>> # enthalpies
>>> Hl = np.array([-10427., -8496.9, -6530.1, -4525., -2480., -393.68, 1735.3, 3908.5]) # J/mol
>>> Hv = np.array([21525., 22893., 24285., 25699., 27135., 28589., 30062., 31549.]) # J/mol
>>> Hvap = Hv - Hl
>>> # interfacial tension
>>> iftexp = np.array([18.842, 17.746, 16.665, 15.6, 14.549, 13.516, 12.499, 11.5]) # mN/m
```

As the available experimental data could vary from one fluid to another, we suggest creating a simple objective function depending on the available information. The following objective function `fobj` measures the error from the saturation pressure, liquid density and vapor density (weighted) for a given set of chain length \( m_s \), potential energy \( \epsilon \), size parameter \( \sigma \) and repulsive exponent \( \lambda_r \). The attractive exponent is set to 6.
>>> from SGTPy import component, saftvrmie

>>> # objective function to optimize molecular parameters
>>> def fobj(inc):
    ms, sigma, eps, lambda_r = inc
    pure = component(ms = ms, sigma = sigma, eps = eps, lambda_r = lambda_r, lambda_a = 6.)
    eos = saftvrmie(pure)

>>> # Pure component pressure and liquid density
>>> P = np.zeros_like(Psat)
>>> vl = np.zeros_like(rhol)
>>> vv = np.zeros_like(rhov)
>>> n = len(Psat)
    for i in range(n):
        P[i], vl[i], vv[i] = eos.psat(Tsat[i], Psat[i])

>>> rhosaftl = 1/vl
>>> rhosaftv = 1/vv

>>> error = np.mean(np.abs(P/Psat - 1))
>>> error += np.mean(np.abs(rhosaftl/rhol - 1))
>>> error += 0.1*np.mean(np.abs(rhosaftv/rhov - 1))

>>> return error

The objective function fobj is optimized with SciPy’s minimize function.

>>> from scipy.optimize import minimize

>>> # initial guess for ms, sigma, eps and lambda_r
>>> inc0 = np.array([2.0, 4.52313581, 378.98125026, 19.00195008])
>>> method = 'Nelder-Mead'
>>> minimize(fobj, inc0, method = method)

More information about the fluid can be included in the objective function. fobj incorporates the speed of sounds to fit the molecular parameters. In order to include this calculation, it is required to compute the ideal gas contribution to the heat capacities and also to supply the molecular weight to the component object.

>>> # Ideal gas heat capacities from DIPPR 801
>>> R = 8.314 # J/mol K
>>> k1=104400
>>> k2=352300
>>> k3=1694.6
>>> k4=236900
>>> k5=761.6
>>> CpId = k1 + k2 * ((k3/Tsat) / np.sinh(k3/Tsat))**2
>>> CvId = k4 * ((k5/Tsat) / np.cosh(k5/Tsat))**2
>>> CpId = 1000. # J/mol K
>>> CvId = CpId - R # J/ mol K

>>> # objective function to optimize molecular parameters
>>> def fobj2(inc):
    ms, sigma, eps, lambda_r = inc
    pure = component(ms = ms, sigma = sigma, eps = eps, lambda_r = lambda_r, lambda_a = 6., Mw = 86.18)
```python
>>> eos = saftvrmie(pure)
>>> #Pure component pressure and liquid density
>>> P = np.zeros_like(Psat)
>>> vl = np.zeros_like(rhol)
>>> vv = np.zeros_like(rhov)
>>> ul = np.zeros_like(speed_liq)
>>> uv = np.zeros_like(speed_vap)
>>> n = len(Psat)
>>> for i in range(n):
>>>     P[i], vl[i], vv[i] = eos.psat(Tsat[i], Psat[i])
>>>     ul[i] = eos.speed_sound(Tsat[i], P[i], 'L', v0=vl[i], CvId=CvId[i], →
˓     →CpId=CpId[i])
>>>     uv[i] = eos.speed_sound(Tsat[i], P[i], 'V', v0=vv[i], CvId=CvId[i], →
˓     →CpId=CpId[i])
>>> rhosaftl = 1/vl
>>> rhosaftv = 1/vv
>>> error = np.mean(np.abs(P/Psat - 1))
>>> error += np.mean(np.abs(rhosaftl/rhol - 1))
>>> error += 0.2*np.mean(np.abs(ul/speed_liq - 1))
>>> error += 0.2*np.mean(np.abs(uv/speed_vap - 1))
>>> return error
```

The optimization procedure is similar to the one explained above.

```python
>>> # initial guess for ms, sigma, eps and lambda_r
>>> inc0 = np.array([ 1.96834567, 4.5462554 , 376.94027466, 18.34400374])
>>> method = 'Nelder-Mead'
>>> minimize(fobj2, inc0, method = method)
>>> # x: array([ 2.24922708, 4.30485061, 323.02126482, 15.14312848])
```

The computed equilibria, enthalpy of vaporization and speed of sound using the optimized molecular parameters are compared against experimental data in the following figure.
Influence parameter for SGT

Finally, influence parameters are necessary to compute interfacial properties, these can be fitted with experimental interfacial tension and the computed phase equilibria.

```python
>>> from SGTPy.fit import fit_cii
>>> ms, sigma, eps, lambda_r = np.
>>> pure = component(ms = ms, sigma = sigma , eps = eps, lambda_r = lambda_r , lambda_a = 6., Mw = 86.18)
>>> eos = saftvrmie(pure)
>>> # P, rhosaftl, rhosaftv were computed previously
>>> fit_cii(iftexp, Tsat, P, rhosaftl, rhosaftv, eos)
>>> # 3.5861724177599335e-19
```
The performance of the fitted influence parameters can be compared against the experimental data (squares).

### 7.4.2 Equilibrium data

SGTPy includes functions to the interaction parameters for binary mixtures, the fitted parameters by pairs can be used then with mixtures containing more components. Depending on the mixture there might be more of one type of phase equilibria. Supposing that the parameters model are represented by $\xi$, the SGTPy functions will fit the given experimental data with the followings objectives functions:

If there is Vapor-Liquid Equilibria (VLE):

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

If there is Liquid-Liquid Equilibria (LLE):

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

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

$$FO_{VLLE}(\xi) = \sum_{j=1}^{N_p} \left[ \sum_{i=1}^{c} \left[ (x_{i,j}^{cal} - x_{i,j}^{exp})^2 + (w_{i,j}^{cal} - w_{i,j}^{exp})^2 + (y_{i,j}^{cal} - y_{i,j}^{exp})^2 + \left( \frac{P_{j}^{cal}}{P_{j}^{exp}} - 1 \right)^2 \right] \right]$$
If there is more than one type of phase equilibria, SGTPy will sum the errors of each one.

As an example the parameters for the system of hexane and ethanol 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.

```python
>>> import numpy as np

# Experimental temperature saturation in K
>>> Texp = np.array([351.45, 349.15, 346.35, 340.55, 339.05, 332.95, 332.55, 331.85, 331.5, 331.25, 331.15, 331.4, 331.6, 332.3, 333.35, 336.65, 339.85, 341.85, ...
... 339.85, 341.85])

# Experimental pressure in Pa
... 101330., 101330.])

# Experimental liquid composition
>>> Xexp = np.array([[0. , 0.01 , 0.02 , 0.06 , 0.08 , 0.152, 0.245, 0.333, 0.452, ...
... 0.588, 0.67 , 0.725, 0.765, 0.898, 0.955, 0.99, 0.994, 1. ],
... [1. , 0.99 , 0.98 , 0.94 , 0.92 , 0.848, 0.755, 0.667, 0.548, ...
... 0.412, 0.33 , 0.275, 0.235, 0.102, 0.045, 0.01 , 0.006, 0. ]])

# Experimental vapor composition
>>> Yexp = np.array([[0. , 0.095, 0.193, 0.365, 0.42 , 0.532, 0.695, 0.63 , 0.64 , 0.65 , 0.66 , 0.67 , 0.675, 0.71 , 0.745, 0.84 , 0.935, 1. ],
... [1. , 0.905, 0.807, 0.635, 0.58 , 0.468, 0.395, 0.37 , 0.36 , 0.35 , 0.34 , 0.33 , 0.325, 0.29 , 0.255, 0.16 , 0.065, 0. ]])

>>> datavle = (Xexp, Yexp, Texp, Pexp)
```

If the system exhibits any other type phase equilibria the necessary tuples would have the following form:

```python
>>> datallle = (Xexp, Wexp, Texp, Pexp)

>>> datavlle = (Xexp, Wexp, Yexp, Texp, Pexp)
```

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

**Fitting $k_{ij}$**

For the mixture of hexane and ethanol, the hexane is modeled as a non-associating fluid and the ethanol is modeled as a self-associating fluid. As there is no cross-association only the $k_{ij}$ binary correction can be optimized. First, the mixture is created.

```python
>>> from SGTPy import component

>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592 , eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii= 5.3141080872882285e-20)

>>> hexane = component('hexane', ms = 1.96720036, sigma = 4.54762477, eps = 377.60127994,
... lambda_r = 18.41193194, cii= 3.581510586936205e-19)

>>> mix = mixture(hexane, ethanol)
```

As a 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:

```python
>>> from SGTPy.fit import fit_kij

>>> # bounds for kij

>>> kij_bounds = (-0.01, 0.01)
```

(continues on next page)
Fitting $k_{ij}$ and $l_{ij}$

There are other mixtures where there is cross-association, as in the mixture of water and ethanol. In this case both $k_{ij}$ and $l_{ij}$ can be optimized. First, the mixture is created.

```python
>>> water = component('water', ms = 1.7311, sigma = 2.4539 , eps = 110.85,
... lambda_r = 8.308, lambda_a = 6., eAB = 1991.07, rcAB = 0.5624,
... rdAB = 0.4, sites = [0,2,2], cii = 1.5371939421515455e-20)
```

```python
>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592 , eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii = 5.3141080872882285e-20)
```

```python
>>> mix = mixture(ethanol, water)
```

The parameters are optimized simultaneously using the `fit_asso` function.

```python
>>> from SGTPy.fit import fit_asso
>>> #initial guess for kij and lij
>>> x0 = np.array([0, 0])
>>> min_options = {'method':'Nelder-Mead'}
>>> fit_asso(x0, mix, datavle = datavle, minimize_options=min_options)
```

Fitting $k_{ij}$ and $r_{cA^{ij}}$

Finally, in the case of mixtures that exhibit induced association, both $k_{ij}$ and $r_{cA^{ij}}$ should be optimized. This optimization is illustrated for a mixture of ethanol and CPME.

```python
>>> ethanol = component('ethanol2C', ms = 1.7728, sigma = 3.5592 , eps = 224.50,
... lambda_r = 11.319, lambda_a = 6., eAB = 3018.05, rcAB = 0.3547,
... rdAB = 0.4, sites = [1,0,1], cii = 5.3141080872882285e-20)
```

```python
>>> cpme = component('cpme', ms = 2.32521144, sigma = 4.13606074, eps = 343.91193798,␣
... lambda_r = 14.15484877,
... lambda_a = 6.0, npol = 1.91990385,mupol = 1.27, sites =[0,0,1], cii␣
... = 3.5213681817448466e-19)
```

```python
>>> mix = mixture(ethanol, cpme)
```

The parameters are optimized simultaneously using the `fit_cross` function.

```python
>>> from SGTPy.fit import fit_cross
>>> #initial guesses for kij and rcij
>>> x0 = [0.01015194, 2.23153033]
>>> fit_cross(x0, mix, assoc=0, datavle=datavle)
```
7.5 Indices and Search

- genindex
- modindex
- search