NTNU

Norwegian University of Science and Technology

Department of chemistry

TKJ4200 Irreversible Thermodynamics

A Computational Study of Soret Coefficients

Authors
Vegard G. Jervell
Katrina Selavko

Supervisor
Professor
Øivind Wilhelmsen

December 18, 2020

Acknowledgements

We would like to thank our supervisor Prof. Øivind Wilhelmsen for his patience, feedback and most of all the motivation and inspiration he has provided, especially towards the end of this project. Further we would like to thank PhD. Kim Kristiansen for his crash course in kinetic gas theory and introduction to the world of Chapman-Enskog, PhD. Sindre Bakke Øyen for his insight and patience regarding the most basic of questions and Morten Andreas Nome for his support when dealing with the subtleties of multivariate numerics.

Abstract

The Soret coefficient is a measurement of mass transport that occurs due to a temperature gradient. There are multiple methods of measuring and calculating the Soret coefficient. The purpose of this work is to implement two models proposed by Kempers,[1,2] and test these against experimental data. The models are tested using a variety of equations of state and are shown to replicate experimental data to some degree. It is shown that the model predictions' dependence on the supplied equation of state varies between different classes of chemical mixtures, and that the selected frame of reference heavily influences the predicted Soret coefficients. A primary focus of the work has been to develop an implementation that facilitates further testing of the models against more classes of mixtures using a wider variety of equations of state.

Contents

1 Introduction

The Soret effect is a thermodynamic phenomenon observed where a temperature gradient leads to mass transport.[3] A measurement of this, the Soret coefficient, is the ratio between the thermal diffusion coefficient ($D_{T,i}$) and the interdiffusion coefficient (D_i) of a species i. Mass transport caused by the Soret coefficient can easily be seen on a radiator below a window. The warm part of a radiator does not collect dust particles. The cold window sill nearby however does collect dust particles. The dust particles would have a negative Soret coefficient as they move towards the cold side of the radiator-window system. In general, the Soret coefficient is positive for the lighter component in the mixture and is usually negative for the heavier component of the mixture. Modeling the Soret effect and predicting the Soret coefficients of a mixture is a problem of interest, and several models have been proposed for this purpose.

1.1 UN Sustainability Goals

The Soret effect affects many different industries.[4] In particular it affects energy storage and production by affecting the amount of power produced or by affecting the lifetime of the related machinery. This relates directly to the UN Sustainability Goal of using clean energy before 2030.

1.1.1 Energy

The petroleum industry is significantly affected by the Soret coefficient.[5] The Soret effect is one of several effects that contribute to the compositional variation in petroleum reservoirs, by giving rise to separation of different chemical compounds in the reservoirs. This effect has been the subject of many studies and modeling attempts.

The Soret effect also plays a role in solar ponds.[5] A solar pond is a solar thermal collector that stores energy through a salt gradient in a pool of water. The surface, which is warmer because of the sun, has lower salinity than the bottom, which is cooler and stores most of the energy. The Soret effect leads to some of the salt migrating with the temperature gradient which mitigates the salinity gradient. This leads reduces the amount of energy the solar pond can contain.

1.1.2 Degradation of Materials

The Soret effect must also be considered in the design of nuclear power plants.[5] Fission reactors have a heat gradient that may lead to a migration of nuclear material from the center to the edges of the reactor cylinders which could cause premature degradation and leakage. Therefore, the Soret effect should be taken into account when designing and running nuclear reactors.

Turbine blades in power plants require a specific composition to avoid breakage under the harsh pressure and temperature conditions. The temperature gradient that develops in these turbines during operation can lead to migration of microconstituents by the Soret-effect, thereby weakening the material. Better ability to model this effect may make it

possible to design materials that can better withstand large temperature gradients over prolonged time.

In general, all materials that rely on microconstituents for their properties and are exposed to large temperature gradients over a prolonged period of time risk degradation due to the Soret effect, though the effect is in most cases expected to be close to negligible. Still, a reliable way to calculate the effect may enable more precise engineering of materials that are to be used in extreme conditions, or materials that are heavily dependent on microstructure and composition for their functionality.

1.2 The Soret coefficient

The Soret coefficient quantifies the coupling of heat and mass flux, and thereby the coupling of the steady state concentration and heat gradients in a system. From the entropy production derived in Appendix B, the conjugate flux-force pairs for a system without charged species can be acquired. The resulting flux equations are

$$\begin{aligned} \mathbf{J}_i &= -\sum_j L_{\mu_i \mu_j} \left(\frac{\nabla \mu_{i,T}}{T} \right) + L_{\mu_i q} \nabla \left(\frac{1}{T} \right) \\ \mathbf{J}'_q &= L_{qq} \nabla \left(\frac{1}{T} \right) - \sum_j L_{q \mu_j} \left(\frac{\nabla \mu_{i,T}}{T} \right) \end{aligned} \tag{1.1}$$

where $\mathbf{J}_i$ is the diffusive flux of chemical species i, $\mathbf{J}'_q$ is the measurable diffusive heat flux and L_{ij} are the phenomenological coefficients.[6] The Soret coefficient of a component describes the ratio of the thermal diffusion contribution to the interdiffusion contribution to the diffusive mass flux, and is defined as

$$s_{T,i} \equiv \frac{D_{T,i}}{D_i} = -\frac{\nabla x_i}{x_i(1-x_i)\nabla T}_{J_i=0}. \tag{1.2}$$

The Soret coefficient is related to the dimensionless thermal diffusion factor α_T,[1] defined as

$$\alpha_{T,i} \equiv -\frac{T \nabla x_i}{x_i(1-x_i)\nabla T} = T s_{T,i} \tag{1.3}$$

Diffusive mass transport due to the Soret effect is very small compared to convective and diffusive flows, the Soret coefficient typically has a magnitude of 1-10mK^{-1}. A graphical representation of the order of magnitude of the Soret effect is shown in Figures 1.2 and 1.1.

Due to convective flows quickly dominating the Soret effect, precise measurements of the effect are difficult to obtain, and reliable quantitative experimental data for gas-phase mixtures is practically non-existent.[2]

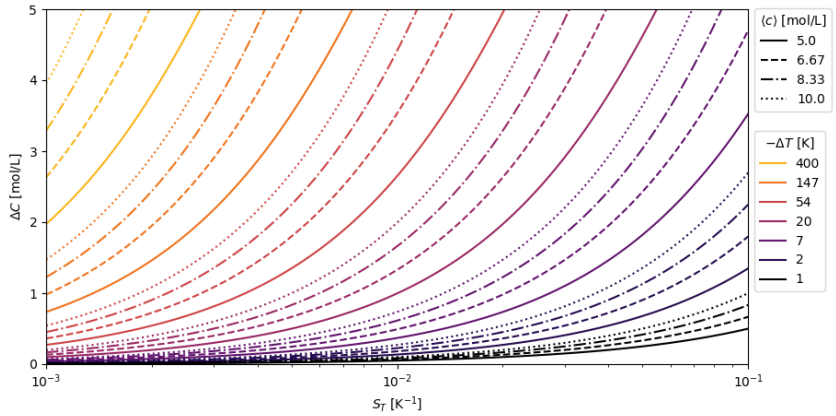


Figure 1.1: Graphical representation of the magnitude of the Soret-effect in a binary system at steady state. Δc is the concentration gradient and ΔT is the temperature gradient.

1.3 Equations of state

An equation of state (EoS) relates different thermodynamic state functions, allowing the computation of quantities such as enthalpy, entropy, Gibbs- and Helmholtz energy and their derivatives at a given macroscopic state.[7] An EoS is typically expressed as a relation $p = p(T, \mathbf{x}, V)$, where p is the pressure of the system, T is the temperature, $\mathbf{x}$ is the composition and V is the volume. Alternatively an EoS can be expressed as a virial expansion of the compressibility factor $Z = Z(p, V, T)$, where the choice of free variables may vary, both density (ρ) and the accentric factor (ω) are common choices.

An EoS contains several parameters that are fitted to experimental data to enable representation of the desired systems.[7] Cubic EoS express pressure as a cubic function of volume, and constitute the simplest class of equations of state known to give reliable results for a variety of compounds, especially in the gas phase.[8] Among these are the Van

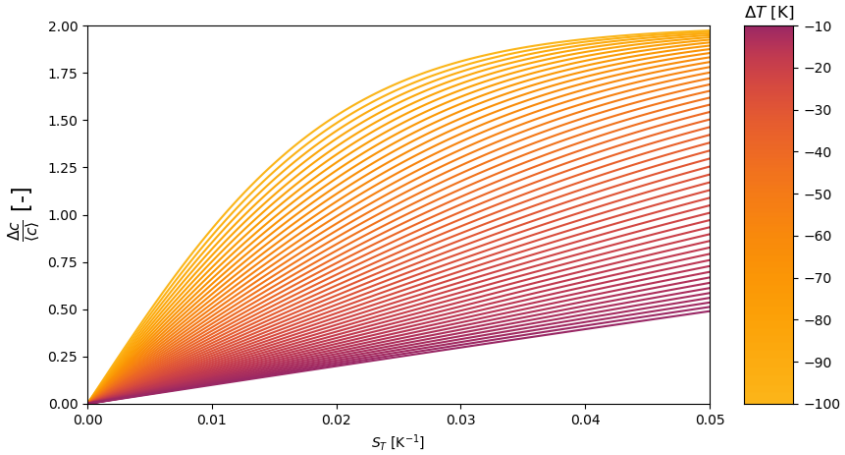


Figure 1.2: Graphical representation of the magnitude of the Soret-effect in a binary system at steady state. Δc is the concentration gradient, ΔT is the temperature gradient, $\langle c \rangle$ is mean concentration.

der Waals (VdW), Soave-Redlich-Kwong (SRK), Peng-Robinson (PR), Patel-Teja (PT), and Schmidt-Wensel (SW) equations of state. These differ in the way their parameters depend on temperature, critical properties etc. and have been developed with different intentions regarding what mixtures and what areas of the phase diagram they are intended to most accurately represent.

The classic cubic-plus-association (CPA) EoS is an expansion of the SRK equation of state intended to better represent mixtures of associating molecules, i.e. polar molar molecules and mixtures in which hydrogen bonding is prevalent.[9] The associative properties of the fluid are described by a term from Wertheim perturbation theory.[10] An equivalent approach has been applied to the PR EoS.[9] An additional EoS that has been used to successfully model associating systems is the Elliot-Suresh-Donohue (ESD) EoS, this follows the same approach of combining a cubic EoS with the Wertheim association term. In this report the abbreviations SRK-CPA and PR-CPA are used to clearly indicate what EoS has been used.

In addition to the cubic equations of state, more complex equations of state have been developed. Among these are statistical-associating-fluid-theory (SAFT), its expansion perturbed-chain-SAFT (PC-SAFT) and an equation of state based on the extended-corresponding-state approach known as the SPUNG EoS.[8,11] The SAFT-approach was developed by Chapman *et al.*[12] based on Wertheims first-order perturbation theory, and has been successfully employed to model polymer solutions and other mixtures. Its expansion PC-SAFT is modified to include the chain-length dependence of dispersion interactions.[11] Both the SAFT and PC-SAFT equations of state notably differ from the cubic equations of state in that one is not dependent on accurate data regarding the critical point to accurately determine the equation parameters. This is due to the parameters having more clear physical interpretations with regard to molecular structure, such as being related to carbon number, bond length etc.[13] The corresponding-state approach is based on the principle of corresponding states. This principle states that all chemical species display equivalent behaviour at the same reduced state. By accurately describing a reference component using a reference EoS, other systems can be relating to the reference component. This is done by expressing the compressibility factor as a function of the reduced variables $Z = Z(V_r, T_r, \omega, ...)$, where $V_r = V/V_c$ and $T_r = T/T_c$, where subscript c denotes the critical values. The extended corresponding-state EoS introduces shape factors (ϕ, θ) to describe a systems deviation from the reference component, expressing the compressibility as $Z = Z\left(\frac{V_r}{\phi}, \frac{T_r}{\theta}\right)$. Different extended-corresponding-state EoS may differ in the choice of reference fluid, reference EoS and description of the shape factors.[8]

1.4 Measurement of the Soret coefficient

The Soret effect was first measured experimentally by Charles Soret, by putting different compounds into a tube, heating one end of the tube and measuring a resulting concentration gradient.[14] Another scientist, Carl Friedrich Wilhelm Ludwig performed similar experiments with liquids.

Gases were the first mixtures to be measured by placing them in Clusius-Dickel or ther-

mogravitational columns.[14] A Clusius-Dickel column had 2 tubes inside each other. The inner tube was heated and the outer tube was cooled. This caused the gases in the mixture to separate according to the Soret effect; one component of the gas concentrated at the inner wall while the other concentrated at the outer wall. For best separation, the tube was spun at 400 rpm to counteract gravitational effects.

Several other older techniques to measure the Soret coefficient involve having two containers separated by a membrane or small tube to limit convection.[14] A temperature gradient is imposed and the mixture is allowed to equilibrate according to the Soret effect.

Using lasers to simultaneously heat the mixtures and measure the refractive index is a more modern method of measuring the Soret coefficient.[14] There are many different techniques that use methods similar to this. The optical transient method, beam deflection technique and thermal diffusion forced Rayleigh scattering are some examples.

In addition, the Soret coefficient can be measured in microgravity conditions.[15] This has occurred on the International Space Station and on parabolic flights where a plane momentarily experiences microgravity. It is believed that experiments performed in microgravity may be more accurate than those performed where gravity has an affect on the mixture being studied.

In 1999, five universities worked together to create a benchmark system to determine the accuracy of new methods for measuring Soret coefficients.[5] This is called the Fountainebleau Benchmark. Each research group measured the experimental Soret coefficient of mixtures of dodecane (C12H26), isobutylbenzene (IBB) and 1,2,3,4-tetrahydronaphthalene (THN). These specific compounds were chosen to represent alkanes, one-ring and two-ring components. Now all new methods of measuring the Soret coefficient are compared to this benchmark to measure accuracy.

1.5 Calculation of the Soret coefficient

The Soret coefficient can be calculated using a variety of methods, among the most known are the models proposed by Haase,[16] Shukla and Firoozbadi,[17] Dougherty and Drickamer,[18] and Kempers.[1,2] Haase's model is based on thermodynamics and includes a term to account for kinetic contributions, but lacks rigour in its derivation.[2] The theory developed by Shukla and Firoozabadi is a kinetic model that includes a fitting parameter that varies between 1 and 4 depending on the components in the mixture, giving the model little predictive power. Dougherty and Drickamer's model is also based on kinetics and includes a fitting parameter, giving it the same shortcoming as Shukla and Firoozabadi's model. The thermal diffusion factor in low-density, gaseous mixtures can be calculated using kinetic gas theory as described by Chapman and Cowling,[19] by utilizing an appropriate intermolecular potential model. Tompson, Tipton and Loyalka elaborate on the Chapman-Cowling equations and give explicit, easily applicable forms of the solutions for a hard sphere potential.[20,21] This model has been shown to give results that are accurate within 20% for near-ideal mixtures.[1]

The focus of this report is the models presented by Kempers in the two papers from 1989 and 2001, hereby referred to as the Kempers-89 or K89 and Kempers-01 or K01 model respectively. The models give the Soret coefficient indirectly, by providing a method

for calculating the thermal diffusion factor, which is related to the Soret coefficient by equation (1.3). They are free from fitting parameters, and do not require input other than data that is readily available for a wide variety of chemical species. However, they require input provided by an equation of state, and are stated to be very sensitive to said input. The goal of this work is to determine the sensitivity of the Kempers model to different equations of state in different systems.

1.6 Kempers 1989

The Kempers-89 model seeks to predict the thermal diffusion factor for liquid-, dense-gas- and near-critical mixtures, using purely thermodynamic considerations[1]. The derivation of the model is based on the consideration of a theoretical two-bulb system. In this system, two bulbs of equal volume are attached to each other by a tube with negligible volume with a valve in the middle. At equilibrium, each bulb is assumed to have homogeneous temperature and composition, and the pressure of the two bulbs is assumed to be equal. One of the bulbs has its temperature increased while the others' temperature is decreased by the same small amount. When the valve is opened, the system can equilibrate, giving a concentration difference between the bulbs according to the Soret effect.

Kempers derives an expression relating the difference in composition and temperature between the bulbs by maximizing the partition function of the system under the constraints of mass conservation, mechanical equilibrium and constant volume of the bulbs. The only assumptions behind this derivation are that convective flows in the tube are negligible, and that the equilibrium state is the macroscopic state that maximises the number of available microstates. The latter is equivalent to assuming that the equilibrium state is the state that minimizes the total free energy of the system. Treating the two-bulb system as fixed makes the center of volume of the system stationary. Using the center of volume frame of reference, for an N-component system yields

$$\begin{gathered}\sum_{j=1}^{N-1}\left[\frac{1}{v_i}\left(\frac{\partial\mu_i}{\partial x_i}\right)-\frac{1}{v_N}\left(\frac{\partial\mu_N}{\partial x_N}\right)\right]\nabla x_j=\left(\frac{h_i}{v_i}-\frac{h_N}{v_N}\right)\frac{\nabla T}{T}, \qquad i=\{1,...N-1\}\\ \sum_i \nabla x_i=0\end{gathered} \tag{1.4}$$

Where h_i and v_i are the partial molar enthalpy and partial molar volume of species i, and all derivatives are taken with $T, p, n, x_i, ...x_{N-1}$ constant. This model is hereby referred to as the K89-CoV model. The relation to the thermal diffusion factor becomes more apparent when inserting equation 1.3, giving the set of equations

$$\begin{gathered}\sum_{j=1}^{N-1}\left[\frac{1}{v_i}\left(\frac{\partial\mu_i}{\partial x_i}\right)-\frac{1}{v_N}\left(\frac{\partial\mu_N}{\partial x_N}\right)\right]x_j(1-x_j)\alpha_{T,j}=\frac{h_N}{v_N}-\frac{h_i}{v_i}, \qquad i=\{1,...,N-1\}\\ \sum_i \alpha_{T,i}=0\end{gathered} \tag{1.5}$$

For a binary system, inserting the relation

$$x_1\left(\frac{\partial\mu_1}{\partial x_1}\right)_{T,p,n}=x_2\left(\frac{\partial\mu_2}{\partial x_2}\right)_{T,p,n} \tag{1.6}$$

yields an explicit expression for the thermal diffusion factor of component 1

$$\alpha_T = \frac{v_1 h_2 - v_2 h_1}{(v_1 x_1 + v_2 x_2) x_1 \frac{\partial \mu_1}{\partial x_1}}. \tag{1.7}$$

Where the subscript is conventionally dropped for binary systems, due to the constraint of the thermal diffusion factors necessarily summing to zero. Applying the same treatment, but allowing the system to move so that the center of mass is stationary, and using the center of mass frame of reference gives an almost identical set of equations, only replacing the partial molar volumes with the molar masses of the respective species. The resulting model is referred to as the K89-CoM model.

The strength of this model is evident: Given an appropriate equation of state, all necessary quantities can be computed, meaning that it in principle opens the possibility of predicting the Soret coefficient in a wide variety of mixtures. The weakness of the model is tied to its high sensitivity to the input-values, the equation of state utilized is required to give highly accurate values for the partial molar quantities of the mixture. Additionally, the standard state partial molar enthalpies enthalpies of the components do not cancel, therefore these are also required as input, and equally precise values are needed.

1.7 Kempers 2001

In 2001 Kempers updated the theory for quantifying the Soret effect [2]. It was updated it to include not only thermodynamic contributions, but also kinetic contributions. The latter are more dominant in dilute gases and become completely dominant towards the limit of the ideal gas state. The need for standard state partial molar enthalpies is also removed. The system under consideration is the same as in the derivation of the K89-model, but in this treatment the ratio of the partition function to the partition function in the ideal gas state is maximised. Thereby quantifying the deviation of the thermal diffusion factor from the thermal diffusion factor in the ideal gas state. Using the center of volume frame of reference gives rise to the the set of equations

$$\begin{aligned}
&\sum_{j=1}^{N-1} \left[\frac{1}{v_i} \left(\frac{\partial \mu_i}{\partial x_j} \right) - \frac{1}{v_N} \left(\frac{\partial \mu_N}{\partial x_j} \right) \right] x_j (1 - x_j) \alpha_{T,j} \\
&= \frac{h_N - h_N^0}{v_N} - \frac{h_i - h_i^0}{v_i} + RT \left(\frac{\alpha_{T,i}^0 (1 - x_i)}{v_i} - \frac{\alpha_{T,N}^0 x_N (1 - x_N)}{v_N} \right), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad i = \{1, ..., N-1\} \\
&\sum_i \alpha_{T,i} = 0
\end{aligned} \tag{1.8}$$

Where superscript 0 indicates the ideal gas state. Essentially, the 2001-model treats the thermal diffusion factor in the ideal gas state (α_T^0) as input, and then gives an expression for the departure from this. Further, equation (1.6) can be inserted to yield an explicit expression for the thermal diffusion factor in a binary system

$$\alpha_T = \frac{v_1 v_2}{v_1 x_1 + v_2 x_2} \frac{\frac{h_2 - h0_2}{v_2} - \frac{h_1 - h0_1}{v_1}}{x_1 \frac{\partial \mu}{\partial x}} + \frac{RT}{\frac{\partial \mu}{\partial x}} \alpha_T^0. \tag{1.9}$$

Analogously to the Kempers-89 model, allowing the system to move such that its center of mass is stationary and using the center of mass frame of reference gives the same set of equations, only replacing the partial molar volumes with the molar masses of the respective species. As for the Kempers-89 model, the K01 center-of-volume and K01 center-of-mass models will be referred to as the K01-CoV and K01-CoM model respectively. It is noted that for a binary system, the K01-CoM model yields the same expression as that derived by Haase, though Haase's derivation included several assumptions that could not generally be said to hold.

The K01 model is a significant improvement compared to the K89 model, in the regard that it does not neglect kinetic contributions, and therefore can be expected to give reasonable predictions also for gaseous mixtures and highly vicious, low density liquid mixtures. It also removes the need for standard state partial molar enthalpies as input parameters. On the other hand, it depends on the thermal diffusion factor in the ideal gas state being supplied, there is limited experimental data for these and their computation introduces a new potential source of error in model predictions. Kempers states that the thermal diffusion factor in the ideal gas state is nearly independent of composition, and that values at one composition therefore can be used to approximate the value at another. However, this does not fall in agreement with experimental data compiled by Vargaftik,[22] or with results from solutions to the Chapman-Enskog equations.[20]

Notably, both the K89 and K01 model may diverge if $\frac{\partial \mu_i}{\partial x_i} = 0$. It is relevant to look more closely at the physical interpretation of a divergent Soret coefficient, to examine whether this result is a fallacy of the models or a physically meaningful result. The condition $\frac{\partial \mu_i}{\partial x_i} = 0$ naturally implies an extrema in the chemical potential of species i wrt. mole fraction. This in turn implies an inflection point in the Gibbs energy surface of the system wrt. the number of moles of i. An inflection point in the Gibbs energy surface is a criterion for the existence of an immicible mixture. Analyzing equation (1.2) at the phase boundary in such a mixture quickly reveals that the result of a divergent Soret coefficient can be physical. Across the phase boundary ∇x_i will remain non-zero even as ∇T goes to zero, making the Soret coefficient divergent and ill-defined.

2 Methods

The K89 and K01 models, both for the center of volume- and center of mass frame of reference were implemented in Python, using ThermoPack,[23] an open-source software developed by SINTEF, as the interface to a variety of equations of state.

The calculated values of Soret coefficients were then compared to experimental values taken from relevant literature to analyze the effectiveness of the two Kempers equations.[24–31] The experimental Soret coefficients were of several binary liquid mixtures at different temperatures and with different mole fractions of the components.

To implement the models, several quantities needed to be derived or calculated. Partial molar enthalpy and partial molar volume are directly available in ThermoPack, and the molar composition is regarded as input. $\left(\frac{\partial \mu_i}{\partial x_i}\right)_{T,p,n}$ is not directly available and has been derived from other values. K01 also requires α_T^0, which has been acquired both by ap-

proximation from experimental values compiled by Vargaftik,[22] and by calculation from Kinetic Gas Theory as described by Tompson, Tipton and Loyalka.[20,21]

2.1 Deriving $\left(\frac{\partial \mu_i}{\partial x_j}\right)_{T,p,n_T}$

An expression for $\left(\frac{\partial \mu_i}{\partial x_i}\right)_{T,p,n_T}$ as a function of quantities easily available in ThermoPack is derived. Then, how this is implemented in ThermoPack as well as methods of checking this derivation are discussed. To begin, make use of the fact that second order derivatives are symmetric with respect to order of differentiation

$$\left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,V} = \left(\frac{\partial}{\partial n_j}\left(\frac{\partial G}{\partial n_i}\right)_{T,p}\right)_{T,V} = \left(\frac{\partial}{\partial n_i}\left(\frac{\partial G}{\partial n_j}\right)_{T,V}\right)_{T,p}. \tag{2.1}$$

From the definition of Gibbs energy,

$$\begin{aligned} dG &= V dp - S dT + \sum_j \mu_j dn_j \\ \left(\frac{\partial G}{\partial n_j}\right)_{T,V} &= V\left(\frac{\partial p}{\partial n_j}\right)_{T,V} + \mu_j. \end{aligned} \tag{2.2}$$

Inserting this on the right hand side of equation (2.1) and differentiating gives

$$\begin{aligned} \left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,V} &= \left(\frac{\partial \mu_j}{\partial n_i}\right)_{T,p} + \left(\frac{\partial V}{\partial n_j}\right)_{T,p}\left(\frac{\partial p}{\partial n_i}\right)_{T,V} + \underbrace{V\left(\frac{\partial}{\partial n_j}\left(\frac{\partial p}{\partial n_i}\right)_{T,V}\right)_{T,p}}_{=0} \\ \left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,V} &= \left(\frac{\partial \mu_j}{\partial n_i}\right)_{T,p} + v_j\left(\frac{\partial p}{\partial n_i}\right)_{T,V}. \end{aligned} \tag{2.3}$$

Further, because $dn_T = dn_j$ when $n_{m\neq j}$ is kept constant

$$\begin{aligned} dn_j &= d(x_j n_T) \\ &= x_j dn_T + n_T dx_j \\ \left(\frac{\partial n_j}{\partial x_j}\right)_{n_{m\neq j}} &= \frac{n_T}{1-x_j}. \end{aligned} \tag{2.4}$$

For any component $k \neq j$ the corresponding $\left(\frac{\partial n_j}{\partial x_k}\right)_{n_{m\neq j}}$ can be expressed as

$$\begin{aligned} n_j &= (1 - x_k - \sum_{m\neq j,k} x_m) n_T \\ dn_j &= (1 - x_k - \sum_{m\neq j,k} x_m) dn_T - n_T dx_k - n_T \sum_{m\neq j,k} dx_m \\ x_k dn_j &= -n_T dx_k - \sum_{m\neq j,k} dn_m \\ \left(\frac{\partial n_j}{\partial x_k}\right)_{n_{m\neq j}} &= -\frac{n_T}{x_k}. \end{aligned} \tag{2.5}$$

Generally, equation (2.4) and (2.5) can be written as

$$\left(\frac{\partial n_i}{\partial x_j}\right)_{n_{k\neq i}} = \frac{n_T}{\delta_{ij} - x_j} \tag{2.6}$$

For a system at constant temperature and pressure

$$d\mu_i = \sum_j \left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,p,n_{k\neq j}} dn_j, \tag{2.7}$$

applying equations (2.4) and (2.5) gives

$$\frac{d\mu_i}{dx_j} = \left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,p,n_{k\neq j}} \frac{n_T}{1 - x_j} - \sum_{k\neq j} \left(\frac{\partial \mu_i}{\partial n_k}\right)_{T,p,n_{m\neq k}} \frac{n_T}{x_j}. \tag{2.8}$$

By combining equations (2.3) and (2.8) the required quantity $\left(\frac{\partial \mu_i}{\partial x_j}\right)_{T,p,n}$ can be calculated using available methods in ThermoPack.

2.1.1 Implementation

The calculation of $\left(\frac{\partial \mu}{\partial x}\right)_{T,p}$ has been implemented as a set of matrix-operations so that numpy can be utilized. Adopting the notation $\mu_{ij} \equiv \frac{\partial \mu_i}{\partial n_j}$, $p_i \equiv \frac{\partial p}{\partial n_i}$ and $\mu_{x,ij} \equiv \frac{d\mu_i}{dx_j}$, the set of equations arising from equation (2.3) with N components can then be written as

$$\begin{bmatrix} \mu_{11} & \dots & \mu_{1N} \\ \vdots & \ddots & \vdots \\ \mu_{N1} & \dots & \mu_{NN} \end{bmatrix}_{T,p} = \begin{bmatrix} \mu_{11} & \dots & \mu_{1N} \\ \vdots & \ddots & \vdots \\ \mu_{N1} & \dots & \mu_{NN} \end{bmatrix}_{T,V} - \begin{bmatrix} v_1 p_1 & \dots & v_N p_1 \\ \vdots & \ddots & \vdots \\ v_1 p_N & \dots & v_N p_N \end{bmatrix}_{T,V} \tag{2.9}$$

The set of equations arising from equation (2.8) with N components can be written

as

$$\begin{bmatrix} \mu_{x,11} & \cdots & \mu_{x,1N} \\ \vdots & \ddots & \vdots \\ \mu_{x,N1} & \cdots & \mu_{x,NN} \end{bmatrix}_{T,p} = n_T \begin{bmatrix} \mu_{11} & \cdots & \mu_{1N} \\ \vdots & \ddots & \vdots \\ \mu_{N1} & \cdots & \mu_{NN} \end{bmatrix}_{T,p} \begin{bmatrix} 1 & -1 & \dots & -1 \\ -1 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 \\ -1 & \dots & -1 & 1 \end{bmatrix} \tag{2.10}$$

Where subscrips on the matrices refer to the variables kept constant in the differentials in the matrix. This allows efficient computation of $\frac{d\mu_i}{dx_j}$ by utilizing the methods `numpy.dot()` and `numpy.tensordot()`.

2.1.2 Sanity Check

To start, equation (2.3) can also be derived by starting from Helmholtz energy and applying the same argumentation. Additionally it is noted that for $\left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,p} = \left(\frac{\partial \mu_j}{\partial n_i}\right)_{T,p}$ to hold, equation (2.3) implies $v_j \left(\frac{\partial p}{\partial n_i}\right)_{T,V} = v_j \left(\frac{\partial p}{\partial n_i}\right)_{T,V}$. This can be checked by substituting $v_j = \left(\frac{\partial \mu_j}{\partial p}\right)_{T,n_{k\neq j}}$

$$\begin{aligned} \left(\frac{\partial \mu_j}{\partial p}\right)_{T,n_{k\neq j}} \left(\frac{\partial p}{\partial n_i}\right)_{T,V} &= \left(\frac{\partial \mu_i}{\partial p}\right)_{T,n_{k\neq j}} \left(\frac{\partial p}{\partial n_j}\right)_{T,V} \\ \left(\frac{\partial \mu_j}{\partial n_i}\right)_{T,V} &= \left(\frac{\partial \mu_i}{\partial n_j}\right)_{T,V} \end{aligned} \tag{2.11}$$

Which holds due to the symmetry of second derivatives. To see that equations (2.4) and (2.5) hold it can be checked that

$$\sum_i \frac{dx_i}{dn_j} = 0. \tag{2.12}$$

Inserting the expressions derived gives

$$\sum_i \frac{dx_i}{dn_j} = \frac{1}{n_T}\left(1 - x_i - \sum_{j\neq i} x_j\right) = 0, \tag{2.13}$$

which indeed is comforting.

2.2 Calculating α_T^0

Kempers suggests several methods for acquiring α_T^0, including using Kinetic Gas Theory or experimental values. Both a method for approximating α_T^0 from experimental data, and the Chapman-Enskog solutions to the Boltzmann equations in Sonine polynomials as proposed by Tompson, Tipton and Loyalka have been implemented. [20,21]

Symbol	Description	Unit
m_i	Molar mass of component $i = \{1, 2\}$	$\mathrm{g\,mol^{-1}}$
m_0	$m_0 \equiv m_1 + m_2$	$\mathrm{g\,mol^{-1}}$
M_i	$M_i \equiv m_i/m_0$	–
n_i	Number of moles of component $i = \{1, 2\}$	mol
n	Total number of moles, $n \equiv n_1 + n_2$	mol
x_i	Mole fraction of component $i = \{1, 2\}$	–
k	Boltzmann constant	$\mathrm{J\,K^{-1}}$
$S_m^{(n)}(x)$	Sonine polynomial, $S_m^{(n)}(x) = \sum_{p=0}^{n} \frac{(m+n)!}{(p)!(n-p)!(m+p)!}(-x)^p$	–
$\boldsymbol{\mathscr{C}_i}$	$\boldsymbol{\mathscr{C}_i} \equiv (m_i/2kT)^{1/2}\boldsymbol{C}_i$	–
$\boldsymbol{C}_i$	$\boldsymbol{C}_i \equiv \boldsymbol{c}_i - \boldsymbol{c}_0$, where $i = \{1, 2\}$	$\mathrm{m\,s^{-1}}$
$\boldsymbol{c}_0$	$\boldsymbol{c}_0 \equiv M_1 x_1 \boldsymbol{c}_1 + M_2 x_2 \boldsymbol{c}_2$	$\mathrm{m\,s^{-1}}$
$\boldsymbol{c}_i$	Pre-collisional velocity of component $i = \{1, 2\}$	$\mathrm{m\,s^{-1}}$
k_T	Thermal diffusion factor (denoted α_T otherwise in this report)	–
σ_i	Molecular hard-sphere diameter of compontent $i = \{1, 2\}$	m
σ_{12}	$\sigma_{12} \equiv \frac{1}{2}(\sigma_1 + \sigma_2)$	m
$\Omega_i^{(l)}(r)$	Collision integrals, solved for a hard-sphere potential in equation (2.29)	

Table 2.1: Symbols used in section 2.2.1, following Tompson, Tipton and Loyalka.[20,21]

2.2.1 From kinetic gas theory

The Chapman-Enskog solutions of the Boltzmann equations can be implemented with the hard-sphere potential to compute the thermal diffusion factor in the ideal gas state. Tompson, Tipton and Loyalka have shown that the use of higher-order Sonine polynomial expansions enables evaluation of these solutions to arbitrary presicion.[20] Further, explicit summational expressions for computation of the required bracket-integrals have been derived.[21] It is worth noting that the solutions are stated to be completely general up until the omega-integrals are evaluated using the hard-sphere potential. The implemented equations will be reiterated here without derivation, but with some elaboration on relations useful in their implementation. In this section, the notation of Tompson, Tipton and Loyalka is followed, a complete list of the symbols used can be found in Table 2.1. To begin, the thermal diffusion factor can be computed as

$$k_T = -\frac{5}{2d_0}\left(\frac{x_1 d_1}{M_1^{1/2}} + \frac{x_1 d_{-1}}{M_2^{1/2}}\right) \tag{2.14}$$

where d_{-1}, d_0 and d_1 are computed by solving the matrix equation

$$\mathbf{D}\mathbf{d} = \boldsymbol{\delta} \tag{2.15}$$

with

$$\mathbf{D} = \begin{bmatrix} a_{-m-m} & \dots & a_{-m0} & \dots & a_{-mm} \\ \vdots & & \vdots & & \vdots \\ a_{0-m} & \dots & a_{00} & \dots & a_{0m} \\ \vdots & & \vdots & & \vdots \\ a_{m-m} & \dots & a_{m0} & \dots & a_{mm} \end{bmatrix}, \qquad \mathbf{d} = \begin{pmatrix} d_{-m} \\ \vdots \\ d_0 \\ \vdots \\ d_m \end{pmatrix}, \qquad \boldsymbol{\delta} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \delta_0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2.16}$$

where

$$\delta_0 = \frac{3}{2n}\left(\frac{2kT}{m_0}\right)^{\frac{1}{2}} \tag{2.17}$$

The essence of the problem consists in generating the a_{pq} matrix elements. The symmetry-relation $a_{pq} = a_{qp}$ is noted and in general

$$\begin{aligned} a_{pq} &= x_1^2\left[S_{3/2}^{(p)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1, S_{3/2}^{(q)}(\mathscr{C}_1^1)\boldsymbol{\mathscr{C}}_1\right]_1 + x_1x_2\left[S_{3/2}^{(p)}(\mathscr{C}_1^1)\boldsymbol{\mathscr{C}}_1, S_{3/2}^{(q)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1\right]_{12} \\ a_{p-q} &= x_1x_2\left[S_{3/2}^{(p)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1, S_{3/2}^{(q)}(\mathscr{C}_2^2)\boldsymbol{\mathscr{C}}_2\right]_{12} \\ a_{-pq} &= x_1x_2\left[S_{3/2}^{(p)}(\mathscr{C}_2^2)\boldsymbol{\mathscr{C}}_2, S_{3/2}^{(q)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1\right]_{21} \\ a_{-p-q} &= x_2^2\left[S_{3/2}^{(p)}(\mathscr{C}_2^2)\boldsymbol{\mathscr{C}}_2, S_{3/2}^{(q)}(\mathscr{C}_2^1)\boldsymbol{\mathscr{C}}_2\right]_2 + x_1x_2\left[S_{3/2}^{(p)}(\mathscr{C}_2^1)\boldsymbol{\mathscr{C}}_2, S_{3/2}^{(q)}(\mathscr{C}_2^2)\boldsymbol{\mathscr{C}}_2\right]_{21} \end{aligned} \tag{2.18}$$

However, only the solutions of the integrals

$$\begin{aligned} H_1^{(1)}(p,q) &\equiv \left[S_{3/2}^{(p)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1, S_{3/2}^{(q)}(\mathscr{C}_1^1)\boldsymbol{\mathscr{C}}_1\right]_1 \\ H_{12}^{(1)}(p,q) &\equiv \left[S_{3/2}^{(p)}(\mathscr{C}_1^1)\boldsymbol{\mathscr{C}}_1, S_{3/2}^{(q)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1\right]_{12} \\ H_{12}^{(12)}(p,q) &\equiv \left[S_{3/2}^{(p)}(\mathscr{C}_1^2)\boldsymbol{\mathscr{C}}_1, S_{3/2}^{(q)}(\mathscr{C}_2^2)\boldsymbol{\mathscr{C}}_2\right]_{12} \end{aligned} \tag{2.19}$$

need to be implemented, and the remaining can be solved by swapping indices. The relations are as follows

$$\begin{aligned} H_1^{(1)} &\to H_2^{(2)}, \qquad & \Omega_1^{(l)}(r) &\to \Omega_2^{(l)}(r) \\ H_{12}^{(1)} &\to H_{21}^{(2)}, \qquad & (M_1, M_2) &\to (M_2, M_1) \\ H_{12}^{(12)} &\to H_{21}^{(21)}, \qquad & (M_1, M_2) &\to (M_2, M_1) \end{aligned} \tag{2.20}$$

The subscripts and superscripts on the $H_{ij}^{(r)}(p,q)$ refer to the subscript on the bracket integral, and the subscript on the $\boldsymbol{\mathscr{C}}_i$ respectively. Tompson, Tipton and Loyalka refer to $H_1^{(1)}(p,q)$, $H_{12}^{(1)}(p,q)$ and $H_{12}^{(12)}(p,q)$ as simple gas bracket integrals, H_1-type integrals and H_{12}-type integrals, without reserving a symbol for the simple gas bracket integral. Here, the superscripts are used to more clearly indicate what indices are swapped when computing the different a_{pq}. Special attention must be paid to the coefficients a_{p0}, $a{-}p0$,

a_{0q}, a_{0-q} and a_{00}, where it is noted from symmetry relations that

$$\begin{aligned} a_{p0} &= x_1 x_2 M_1^{\frac{1}{2}} H_{12}^{(1)}(p,0) \\ a_{-p0} &= x_1 x_2 \left(-M_2^{\frac{1}{2}}\right) H_{21}^{(2)}(p,0) \\ a_{00} &= x_1 x_2 M_1 H_{12}^{(1)}(0,0) \\ a_{0q} &= x_1 x_2 M_1^{\frac{1}{2}} H_{12}^{(1)}(0,q) \\ a_{0-q} &= x_1 x_2 \left(-M_2^{\frac{1}{2}}\right) H_{21}^{(2)}(0,q) \end{aligned} \tag{2.21}$$

The summational expressions for the integrals are given as

$$H_1^{(1)}(p,q) = 8 \sum_{l=2}^{(\min[p,q]+1)} \sum_{r=l}^{(p+q+2-l)} A'''_{pqrl} \Omega_1^{(l)}(r) \tag{2.22}$$

where

$$\begin{aligned} A'''_{pqrl} = \left(\frac{1}{2}\right)^{(p+q+1)} \sum_{i=(l-1)}^{(\min[p,q,r,(p+q+1-r)])} & \frac{8^i (p+q-2i)! \left(1+(-1)^l\right)}{(p-i)!(q-i)!(l)!(i+1-l)!} \\ \times & \frac{(-1)^{(r+i)}(r+1)!(2(p+q+2-i))!2^{2r}}{(r-i)!(p+q+1-i-r)!(2r+2)!(p+q+2-i)!4^{(p+q+1)}} \\ \times & \left[(i+1-l)(p+q+1-i-r) - l(r-i)\right] \end{aligned} \tag{2.23}$$

It is noted that the factor $\left((1+(-1)^l\right)$ will equal zero when l is odd and two when l is even, so summation in equation (2.22) need only pass over even l. It is also worth noting that $A'''_{pqrl} = 0$ when $pq = 0$. Further,

$$H_{12}^{(1)}(p,q) = 8 \sum_{l=1}^{(\min[p,q]+1)} \sum_{r=l}^{(p+q+2-l)} A'_{pqrl} \Omega_{12}^{(l)}(r) \tag{2.24}$$

where

$$\begin{aligned} A'_{pqrl} = & \sum_{i=(l-1)}^{\min[p,q,r,(p+q+1-r)]} \sum_{k=(l-1)}^{\min[l,i]} \sum_{w=0}^{(\min[p,q,(p+q+1-r)-i)} \frac{8^i (p+q-2i-w)!}{(p-i-w)!(q-i-w)!} \\ & \times \frac{(-1)^{(r+i)}(r+1)!(2(p+q+2-i-w))!4^{(r+w)} F^{(i+k)} G^w M_1^i M_2^{(p+q-i-w)}}{(r-i)!(p+q+1-i-r-w)!(2r+2)!(p+q+2-i-w)!4^{(p+q+1)}(k)!(i-k)!(w)!} \\ & \times \left(M_1(p+q+1-i-r-w)\delta_{k,l} - M_2(r-i)\delta_{k,(l-1)}\right) \end{aligned} \tag{2.25}$$

where

$$F \equiv \frac{M_1^2 + M_2^2}{M_1 M_2}, \qquad G \equiv \frac{M_1 - M_2}{M_2} \tag{2.26}$$

and $\delta_{i,j}$ is the Kronecker-delta. Finally,

$$H_{12}^{(12)}(p,q) = 8 M_2^{(p+\frac{1}{2})} M_1^{(q+\frac{1}{2})} \sum_{l=1}^{(\min[p,q]+1)} \sum_{r=l}^{(p+q+2-l)} A_{pqrl} \Omega_{12}^{(l)}(r) \tag{2.27}$$

where

$$
\begin{aligned}
A_{pqrl} = & \sum_{i=(l-1)}^{\min[p,q,r,(p+q+1-r)]} \frac{8^i (p+q-2i)!}{(p-i)!(q-i)!(l)!(i+1-l)!(r-i)!} \\
& \times \frac{(-1)^{(l+r+i)}(r+1)!(2(p+q+2-i))!4^r}{(p+q+1-i-r)!(2r+2)!(p+q+2-i)!4^{(p+q+1)}} \\
& \times [(i+1-l)(p+q+1-i-r) - l(r-i)]
\end{aligned}
\tag{2.28}
$$

the omega integrals can be readily evaluated for a hard-sphere potential and are given as

$$
\begin{aligned}
\Omega_1^{(l)}(r) &= \sigma_1^2 \left(\frac{\pi kT}{m_1}\right)^{\frac{1}{2}} W(l,r) \\
\Omega_2^{(l)}(r) &= \sigma_2^2 \left(\frac{\pi kT}{m_2}\right)^{\frac{1}{2}} W(l,r) \\
\Omega_{12}^{(l)}(r) &= \frac{1}{2}\sigma_{12}^2 \left(\frac{2\pi kT}{m_0 M_1 M_2}\right)^{\frac{1}{2}} W(l,r)
\end{aligned}
\tag{2.29}
$$

where

$$
W(l,r) = \frac{1}{4}\left[2 - \frac{1}{l+1}\left(1 + (-1)^l\right)\right](r+1)! \tag{2.30}
$$

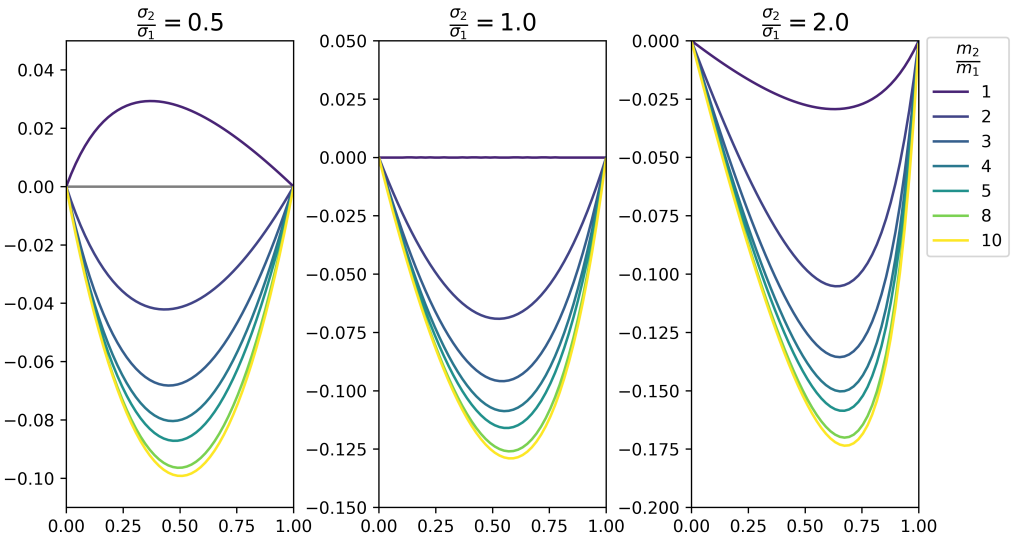


Figure 2.1: First order approximated k_T calculated for the same theoretical σ_2/σ_1 and m_2/m_1 ratios as those reported in [20].

The preceding equations have been implemented in the module kineticgas.py. Using the same σ_2/σ_1 and m_2/m_1 ratios as Tompson, Tipton and Loyalka the computed k_T are indistinguishable from those they have reported. The k_T values calculated for comparison are shown in Figure 2.1. The calculated k_T converges quickly with increasing order of

approximation, however the computational requirements increase very quickly as well, and the complexity of the calculation is estimated to $\mathbf{\Theta}(n^4)$ where n is the order of approximation. The convergence of d_{-1}, d_0 and d_1 as well as the required computation time is shown in Figure 2.2. Based on this, it is recommended to use no lower than the 5th order approximation while it is noted that running 22nd order approximation or higher in the current implementation results in an overflow error, and in general that a high computational price must be paid for small increases in precision when using higher than 8th order approximations. However, as mentioned by Tompson, Tipton and Loyalka the A_{pqrl}, A''_{pqrl} and A'''_{pqrl} can be written in a manner that is independent of molecular mass even though this is not evident from the way they have chosen to express them in the summational formulas given.[21] This enables the computation of these coefficients a single time for an arbitrary precision, such that they can be used for any mixture, and is recommended if a new implementation is to be written.

The thermal diffusion factor (D_T) and the interdiffusion factor ($D_{1,2}$) can be calculated from

$$\begin{aligned} D_{1,2} &= \frac{1}{2} x_1 x_2 \left(\frac{2kT}{m_0}\right)^{\frac{1}{2}} d_0, \\ D_T &= -\frac{5}{4} x_1 x_2 \left(\frac{2kT}{m_0}\right)^{\frac{1}{2}} \left(\frac{x_1 d_1}{M_1^{1/2}} + \frac{x_1 d_{-1}}{M_2^{1/2}}\right) \end{aligned} \tag{2.31}$$

As these only require the quantities d_1, d_{-1} and d_0, implementing these calculations is trivial once the calculation of k_T has been implemented. Therefore, these quantities have also been made readily available in the current implementation.

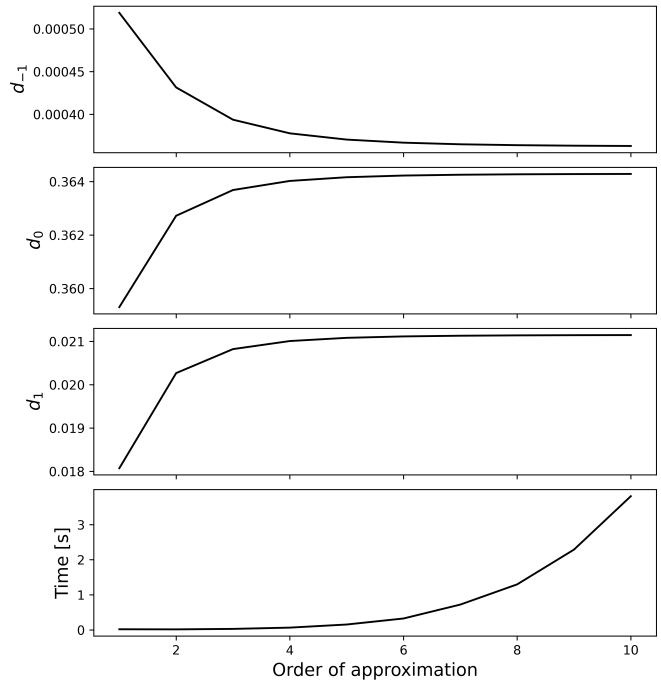


Figure 2.2: Top three plots show convergence of d_{-1}, d_0 and d_1 with increasing order of approximation. Bottom plot shows required computational time.

2.2.2 Approximating from experimental data

An algorithm to approximate α_T^0 at any given concentration and temperature from experimental data was implemented. The currently compiled set of experimental data for α_T^0 has no overlap with the experimental data found for the Soret coefficient. The method is still included here, as it may prove useful in the case of an expanded data set being available in the future. To explain the algorithm the following symbols are used: P', the point in the cT-plane to approximate; P_i, the i'th closest point to P' and $\tilde{P}_n$, the n'th closest point to P' that contributes to a polygon in the cT-plane enclosing P' that is used for approximating α_T^0. All the aforementioned have been implemented as 3D-vectors with a z-component equal to zero, the reasoning for this will become clear shortly. The method involves selecting appropriate data points for a good approximation at the required point, and computing α_T^0 from an interpolation. Appropriate points were selected by constructing a small polygon surrounding the required point using experimental points as corners, the interpolation was computed as

$$\alpha_T^0(P') = \sum_n w_n \alpha_{T,n}^0 \ , \qquad w_n = \frac{\prod_{j\neq n} r_j^2}{\sum_k \prod_{j\neq k} r_j^2} \tag{2.32}$$

where r_j is the distance from the P' to $\tilde{P}_j$ in the cT-plane. This weighting-scheme ensures that

$$w_n(P') = \begin{cases} 1, & P' = \tilde{P}_n \\ 0, & P' = \tilde{P}_{k \neq n} \end{cases} \tag{2.33}$$

thereby ensuring that the approximated value will interpolate the experimental values. The weights as calculated for a random set of points are displayed in Figure 2.4. Selection of appropriate points, hereby denoted as *valid* points, was done in three steps. First, select the data point closest to P', denoted $\tilde{P}_0$. Second, select the second closest point, P_1 and compute the scalar product $(P' - \tilde{P}_0) \cdot (P_1 - \tilde{P}_0)$. If this is positive, P_1 is a valid point and $\tilde{P}_1 := P_1$. In general, a point $n > 0$ is valid if

$$(P' - \tilde{P}_i) \cdot (P_n - \tilde{P}_i) > 0 \; \forall \; \tilde{P}_i \; \in \{\tilde{P}\}, \tag{2.34}$$

where $\{\tilde{P}\}$ denotes the set of previously selected valid points. Before selecting any point $n > 3$, check if the current set of valid points construct a polygon in the cT-plane that encloses P'. Denoting the vector from P' to $\tilde{P}_i$ as $\mathbf{P_i} = \tilde{P}_i - P'$, this is true if

$$\exists \; (\tilde{P}_i, \tilde{P}_j) \mid (\mathbf{P_0} \times \mathbf{P_i})\mathbf{e}_\perp > 0 \; \wedge \; (\mathbf{P_0} \times \mathbf{P_j})\mathbf{e}_\perp < 0 \; \wedge \; (\mathbf{P_i} \times \mathbf{P_j})\mathbf{e}_\perp > 0, \tag{2.35}$$

where $\mathbf{e}_\perp$ is the unit vector perpendicular to the cT-plane. A geometric representation of the selection of valid points and determination of whether the selected points enclose P', and a schematic of the implemented algorithm is shown in Figure 2.3. Plots that show the approximated α_T^0-values for selected binary mixtures are shown in Figure 2.5. It is evident that the method described here gives more consistently physical approximations than the Clough-Tocher implementation in scipy.interpolate.

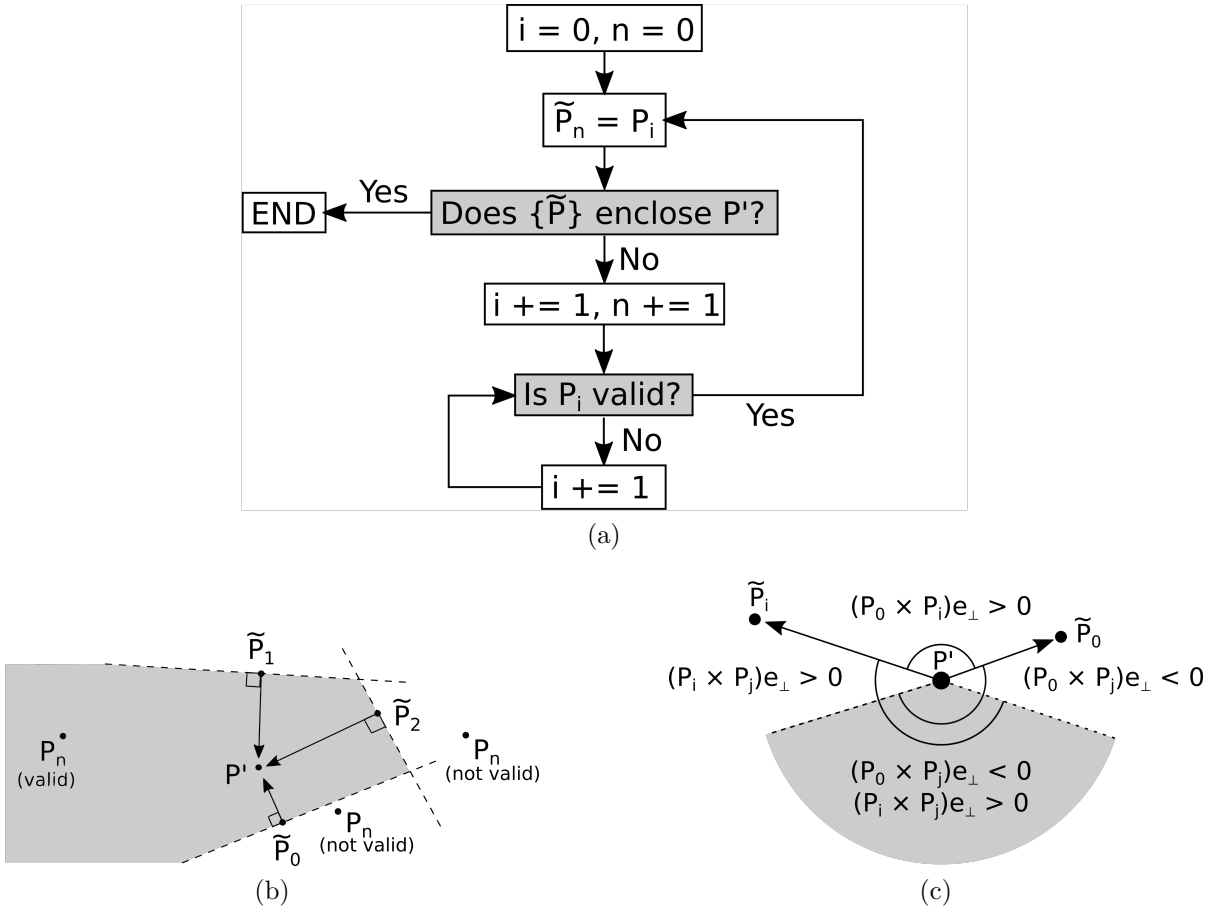


Figure 2.3: a) Schematic of the algorithm used to select points for approximating P'. b) Points in the shaded area are regarded as valid. c) Any point in the shaded area will complete a polygon enclosing P'.

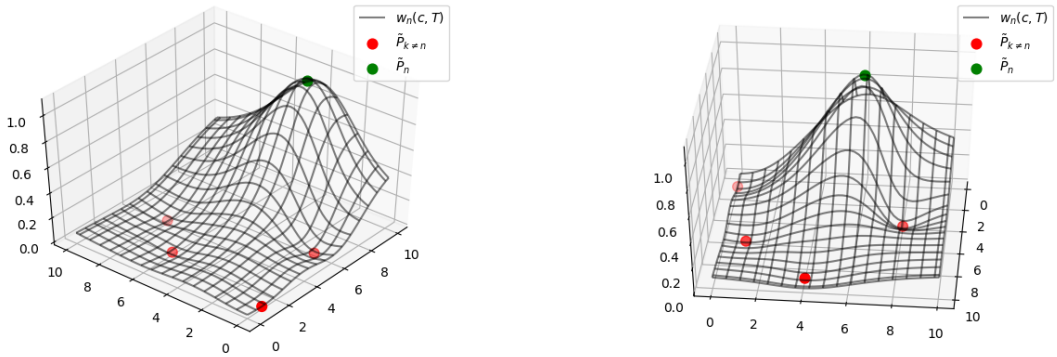
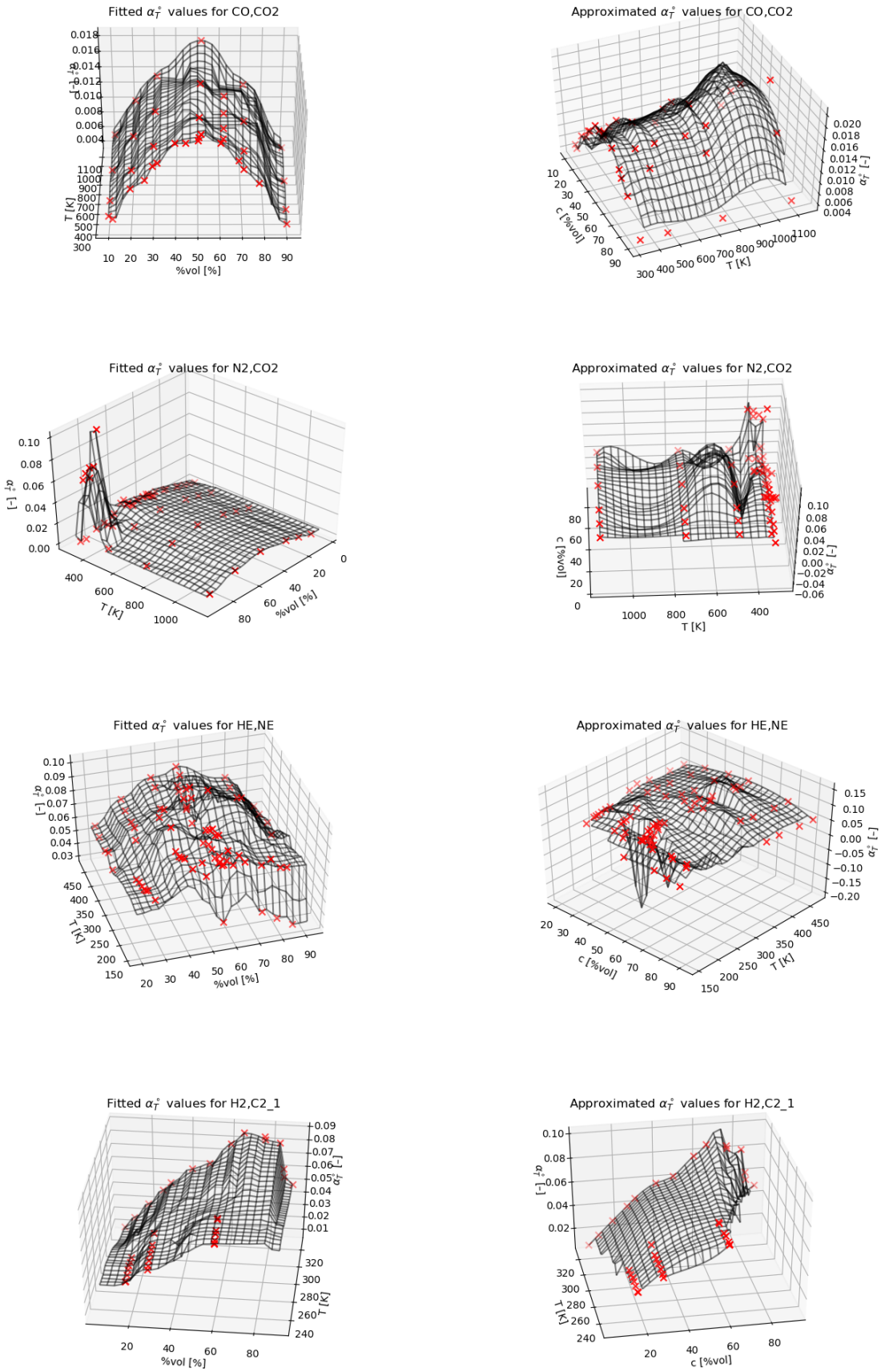


Figure 2.4: The weight distribution of two different data points. $w_n(c, T)$ is the black surface, $\tilde{P}_n$ is the green point, $\tilde{P}_{k \neq n}$ are the red points.



(a) (b)

Figure 2.5: Approximated α_T^0 values for some binary mixtures. Red marks are experimental data points, surface is approximated value at every point. a) Method described in this section. b) Clough-Tocher interpolation as implemented in scipy.interpolate

2.3 Ideal gas standard state enthalpies

The K89 model requires the absolute partial molar enthalpies of components, not only their departure from the ideal gas state. As ideal gas state enthalpies are not included in the ThermoPack database, manual corrections must be made to the partial molar enthalpies returned by ThermoPack. The corrections were made by computing the partial molar enthalpy at the desired conditions, subtracting the partial molar enthalpy ThermoPack computes at 1 µPa and adding the respective standard state enthalpy found in Table C.8. The K89-model has been tested both by computing the partial molar enthalpy as stated here, and using only the residual partial molar enthalpy. The latter has been achieved by only subtracting the partial molar enthalpy computed by ThermoPack at 1 µPa, and refraining from adding the ideal gas state enthalpies. The resulting model will be denoted as the modified Kempers-89 model or M-K89, and it is noted that it is equivalent to the Kempers-01 model when disregarding the kinetic contribution.

2.4 Implementation

The implementation of the Kempers -89 and -01 models, as well as modules for calculating and approximating α_T^0 can be found in their entirety in Appendix D. The Kempers-models require an initialized equation of state as input, and contain a variety of methods to efficiently compute the Soret coefficients of all components for a range of temperatures, pressures or compositions. The modules solve the respective sets of equations for multicomponent systems prescribed by Kempers, meaning that they are implemented to be multicomponent-compatible despite only binary systems being investigated in this report. The module that computes α_T^0 is implemented for a binary system only. As a result the K01 model, which is dependent on the computation of α_T^0, is restricted to binary systems until supplied with a multicomponent-compatible module for calculation of α_T^0. Both Kempers-modules have been tested for multicomponent systems, with placeholder values for α_T^0 in the case of the 01-module, to confirm that they indeed run for ternary and quaternary systems. However, as experimental data was only found for binary systems, the values computed in these runs are not reported.

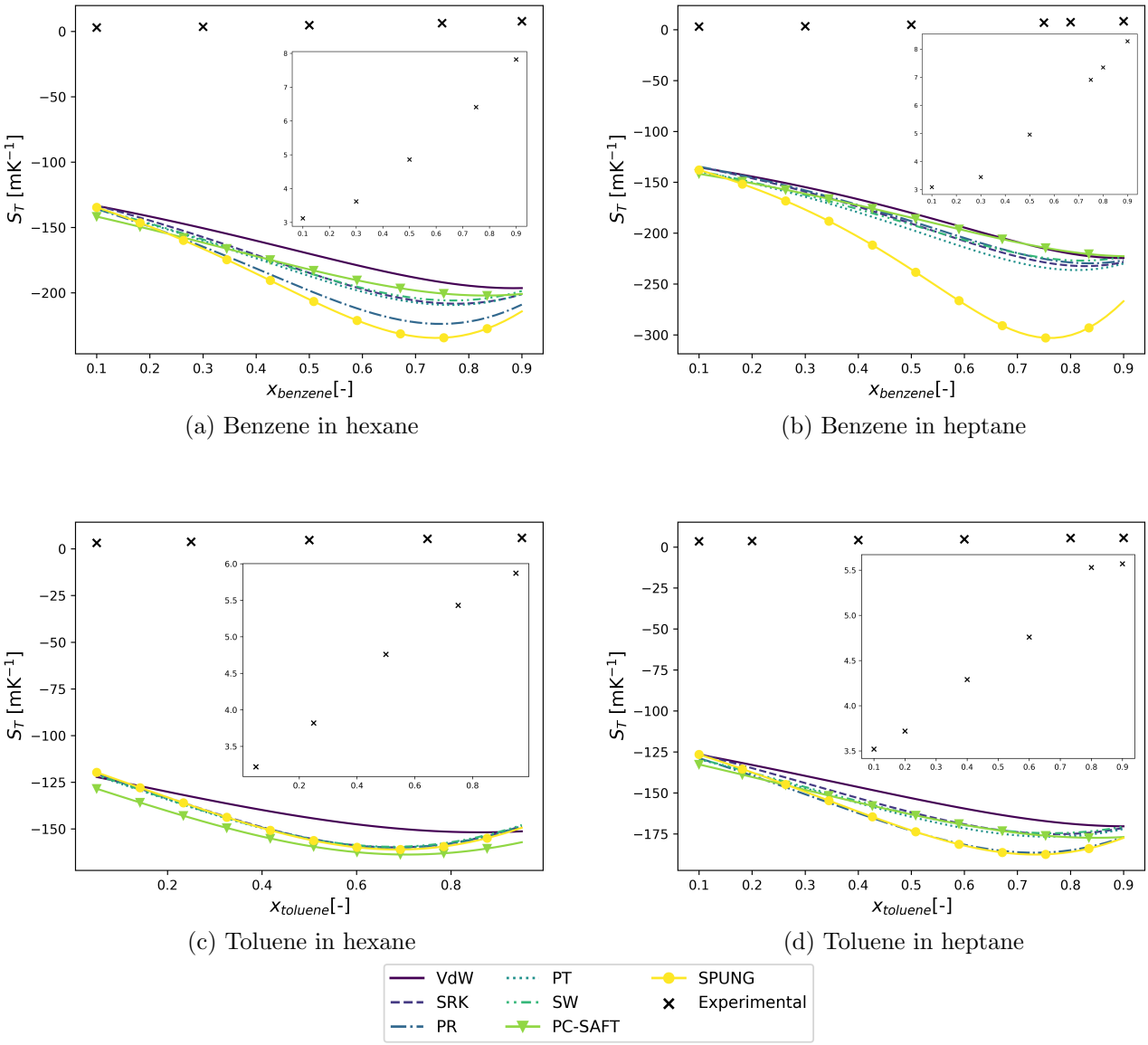
The strength of the current implementation lies in the ease with which a series of equations of state can be tested. Any EoS implemented in ThermoPack that supports the standard tv-interface and property-interface methods can be initialized with a set of parameters and a mixing rule before being fed into the Kempers model. This opens for efficiently investigating what equations of state are suitable to model the Soret effect in a multitude of systems.

3 Results

Both the K89, M-K89 and K01 model were tested against a set of aromate/n-alkane mixtures, various n-alkane/n-alkane mixtures, two associating mixtures and a mixture of liquefied gas. The tested equations of state were: SRK, PR, PT, VdW, SW, SRK-CPA, PC-SAFT and SPUNG. Not all equations of state were tested for every system, either due to their performance being known or expected to be poor for the given system, or in the case of SRK-CPA, its behaviour being equivalent to SRK for non-associating systems. The SPUNG EoS was run using SRK as the shape factor EoS, VdW as the shape factor mixing rule, NIST-MEOS as the reference EoS, propane as the reference compound and the parameter 'Classic' for the shape factor alpha. For clarity, when referring to 'the Soret coefficient of Component 1 in Component 2', it is implied that Component 1 is being discussed, and that the Soret coefficient of Component 2 has an equal magnitude and the opposite sign.

3.1 Kempers-89

The Kempers-89 model as initially implemented predicted Soret-coefficients up to a factor 15 off from experimental values. As shown by the insets in Figures 3.1 and 3.2, the model also failed to predict the correct sign and trend for aromate/n-alkane mixtures. Both the K89-CoM and the K89-CoV model were run using the SRK, VdW, PR, PT, SW, PC-SAFT and SPUNG equations of state for the conditions displayed in Figure 3.2, with none of these providing results that stand out from those shown in the figure.



(a) Benzene in hexane

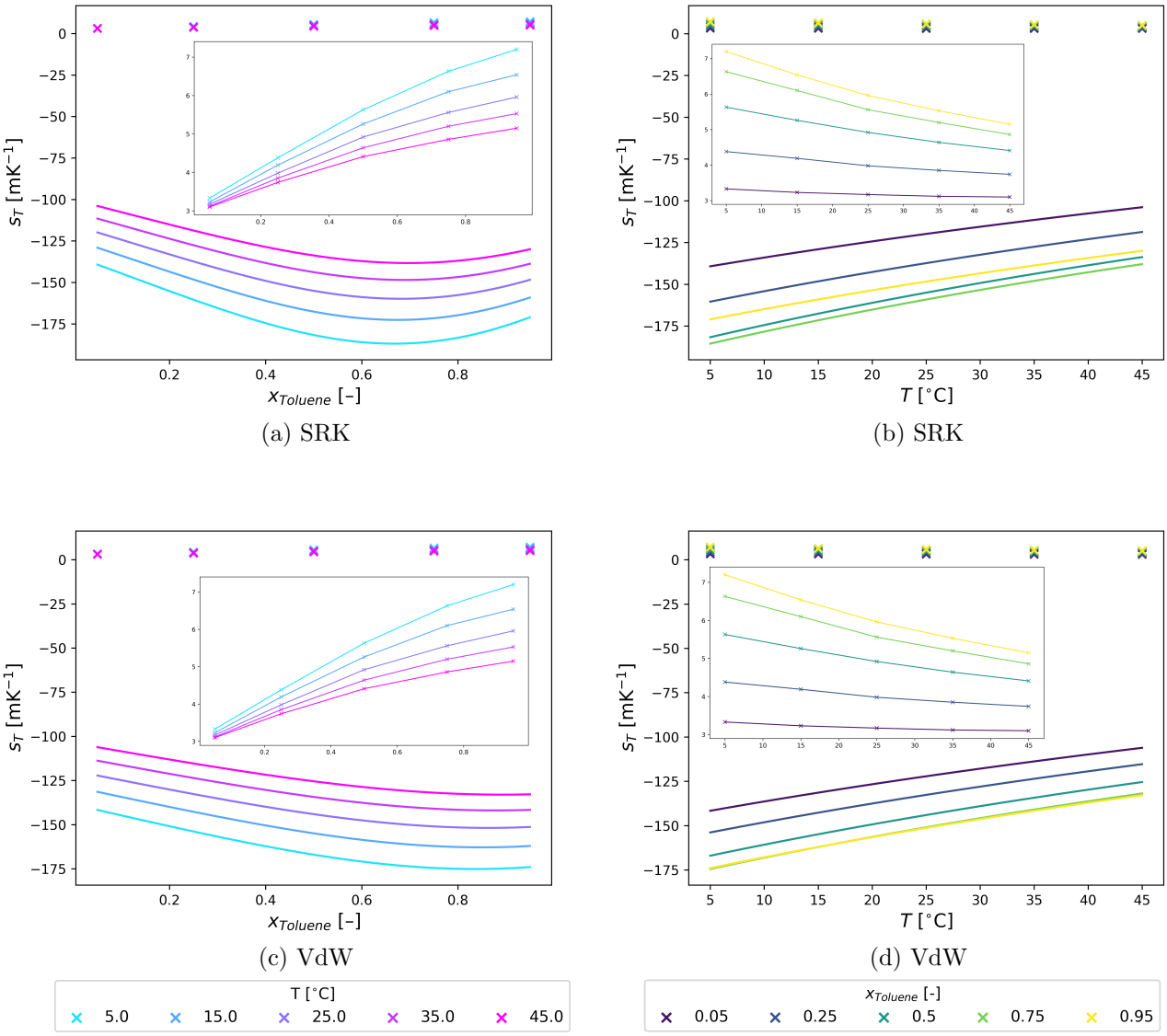
(b) Benzene in heptane

(c) Toluene in hexane

(d) Toluene in heptane

Figure 3.1: K89-CoV model predictions for several aromate/n-alkane mixtures at at 298 K, 1 atm.Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display model predictions for different EoS as indicated in the legend. Insets show experimental data, found in Table C.3.

When tested against the n-alkane/n-alkane mixtures displayed in Figure 3.3, the K89-CoV model over-predicted the Soret coefficient by up to one order of magnitued, except when supplied with the SW, PT or PC-SAFT EoS. The correct sign was predicted in almost all cases. However, the correct trend was only predicted when using the EoS that gave absolute values far away from experimental data. The predictions made by the K89-CoM model, shown in Figure 3.4 missed experimental values by up to one order of magnitude for all equations of state. However, all EoS gave the correct sign and trend.

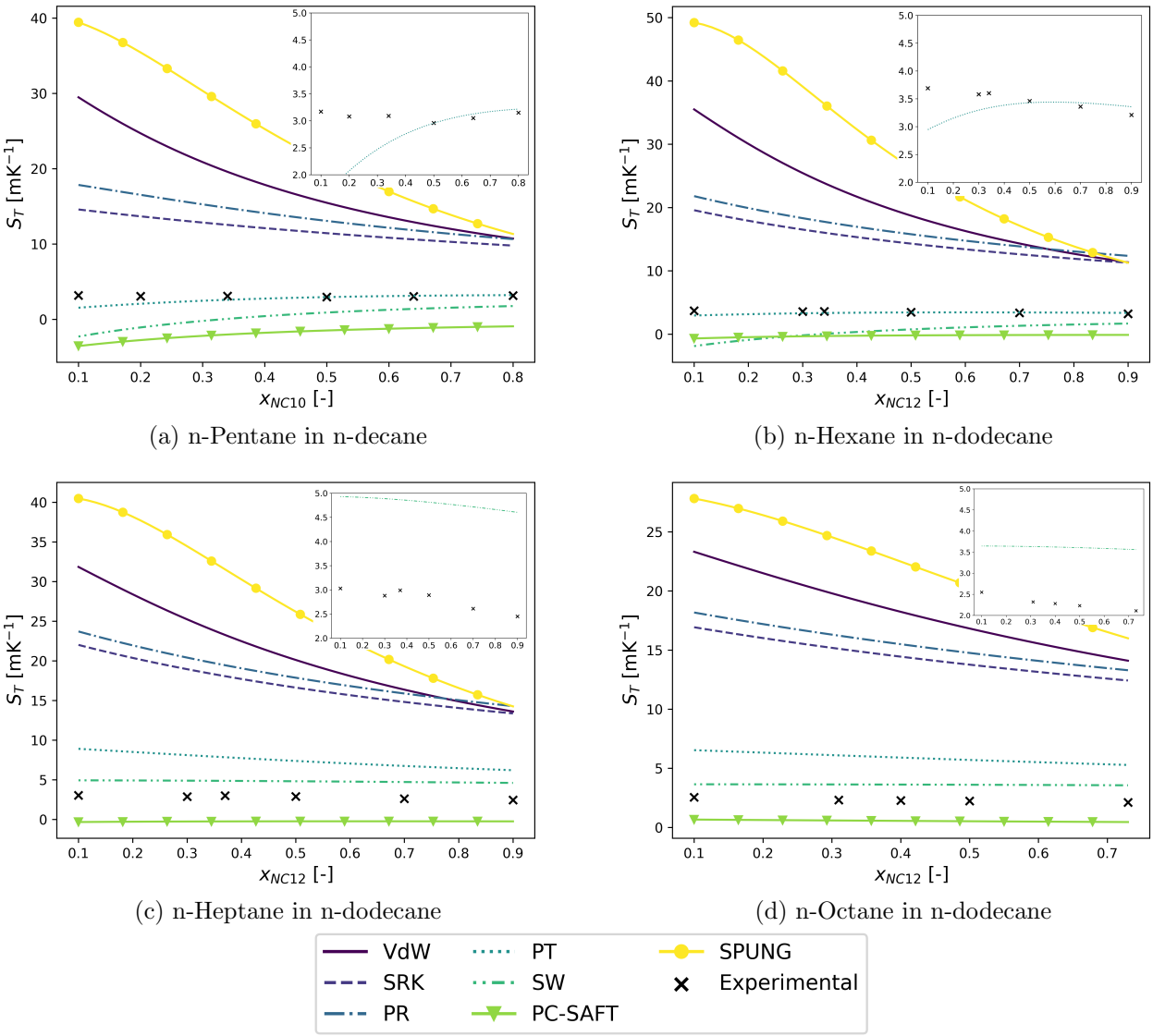


(a) SRK (b) SRK

(c) VdW (d) VdW

Figure 3.2: K89-CoV predicted Soret coefficient of toluene in hexane at 1 atm and various temperatures and compositions. Marks are experimental data, lines are the Soret coefficient as predicted using a-b) SRK EoS, c-d) VdW EoS. Insets show experimental data with lines to clearly indicate trends. Data is found in Table C.2.

For the ethanol-water mixture, running the K89-model with any other EoS than the SRK-CPA gave divergent results, as shown in Figure 3.5 for the PR EoS. This was expected, as the ethanol-water system has strongly prevalent hydrogen bonding, that is not captured except by the CPA-EoS. Using the SRK-CPA EoS, the model failed to reproduce the change in sign of the Soret coefficient, and the absolute value of the predictions missed the experimental values by approximately a factor of 15 as shown in Figure 3.6.



(a) n-Pentane in n-decane

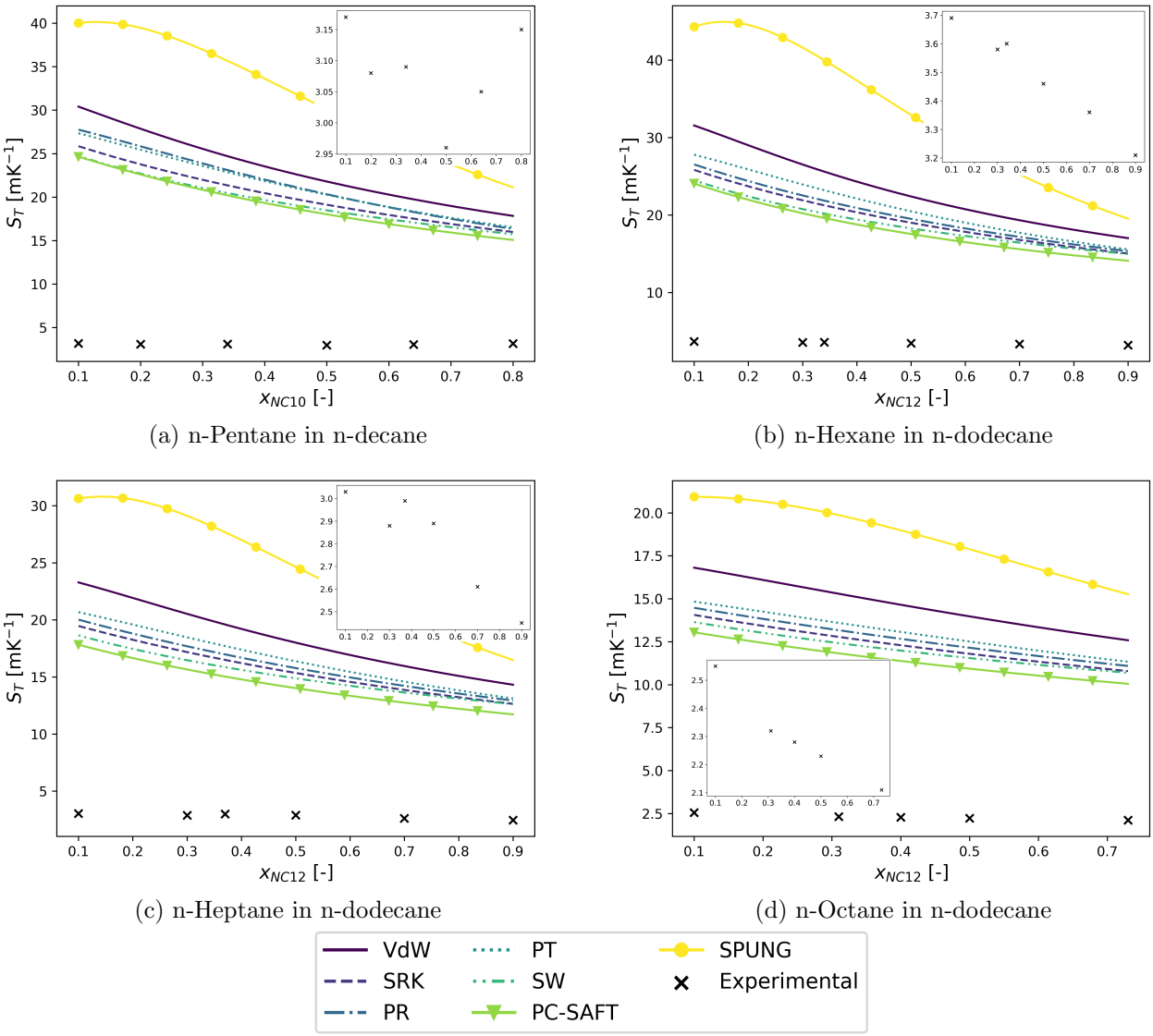
(b) n-Hexane in n-dodecane

(c) n-Heptane in n-dodecane

(d) n-Octane in n-dodecane

Figure 3.3: K89-CoV model predictions of Soret coefficient in several n-alkane/n-alkane mixtures at 298 K, 1 atm. Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display predicted Soret coefficient using the EoS as indicated in the legend. Inset shows experimental data found in Table C.4.

In testing against the liquid argon-methane mixture, the K89-model reproduced the correct sign of the Soret-coefficient, as shown in Figure 3.7. There was a wide spread in the Soret coefficient predicted when using different equations of state, both regarding order of magnitude and trend. The SPUNG EoS stands out as it gives predictions far closer to experimental data than the remaining EoS, but predicts an incorrect trend. It is noted that the SPUNG EoS predicted the mixture to be in the gas phase, and that the remaining correctly simulated the liquid phase.



(a) n-Pentane in n-decane

(b) n-Hexane in n-dodecane

(c) n-Heptane in n-dodecane

(d) n-Octane in n-dodecane

Figure 3.4: K89-CoM predicted Soret coefficient in several n-alkane/n-alkane mixtures at 298 K, 1 atm. Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display predicted Soret coefficient using the EoS as indicated in the legend. Inset shows experimental data, found in Table C.4.

In general predictions by the K89 model were poor, in many cases giving predictions with an absolute value a factor of 10-15 times to large. Additionally, there was little overlap between the EoS that gave predictions closer to experimental data, and EoS that predicted the correct trends.

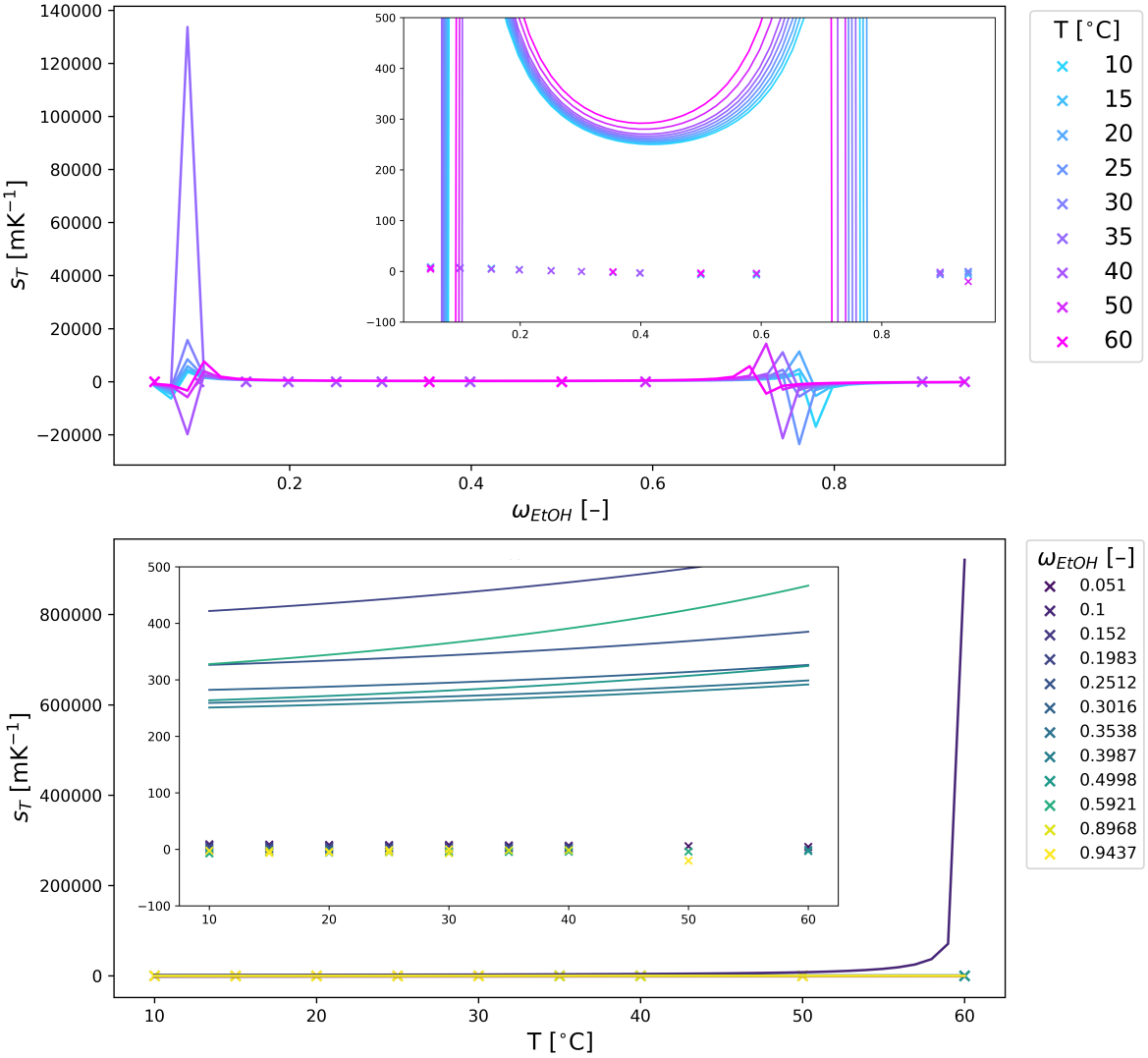


Figure 3.5: K89-CoV model predictions of Soret coefficient of ethanol in water using the PR EoS. Conditions are 1 atm, various temperatures and compositions. Marks are experimental data, lines display predicted Soret coefficient. Inset shows a magnified ordinate axis. ω_{ETOH} is the weight fraction of ethanol. Experimental data is found in Table C.1.

3.2 Modified Kempers-89

The M-K89 model provided far more accurate predictions of the Soret coefficient than the K89 model. As shown in Figure 3.8, predictions by the M-K89-CoV model are close to experimental values for several aromate/n-alkane mixtures when the model is supplied with the VdW EoS. The predicted Soret coefficient has the correct sign using all displayed equations of state with the exception of PC-SAFT. The latter gives values in the order of $-1\,\mu K^{-1}$ to $-0.1\,\mu K^{-1}$, with values following the same trend as the other EoS. It is noted that the VdW EoS predicts the mixture to be in the gas phase, while all other equations of state correctly model the liquid phase.

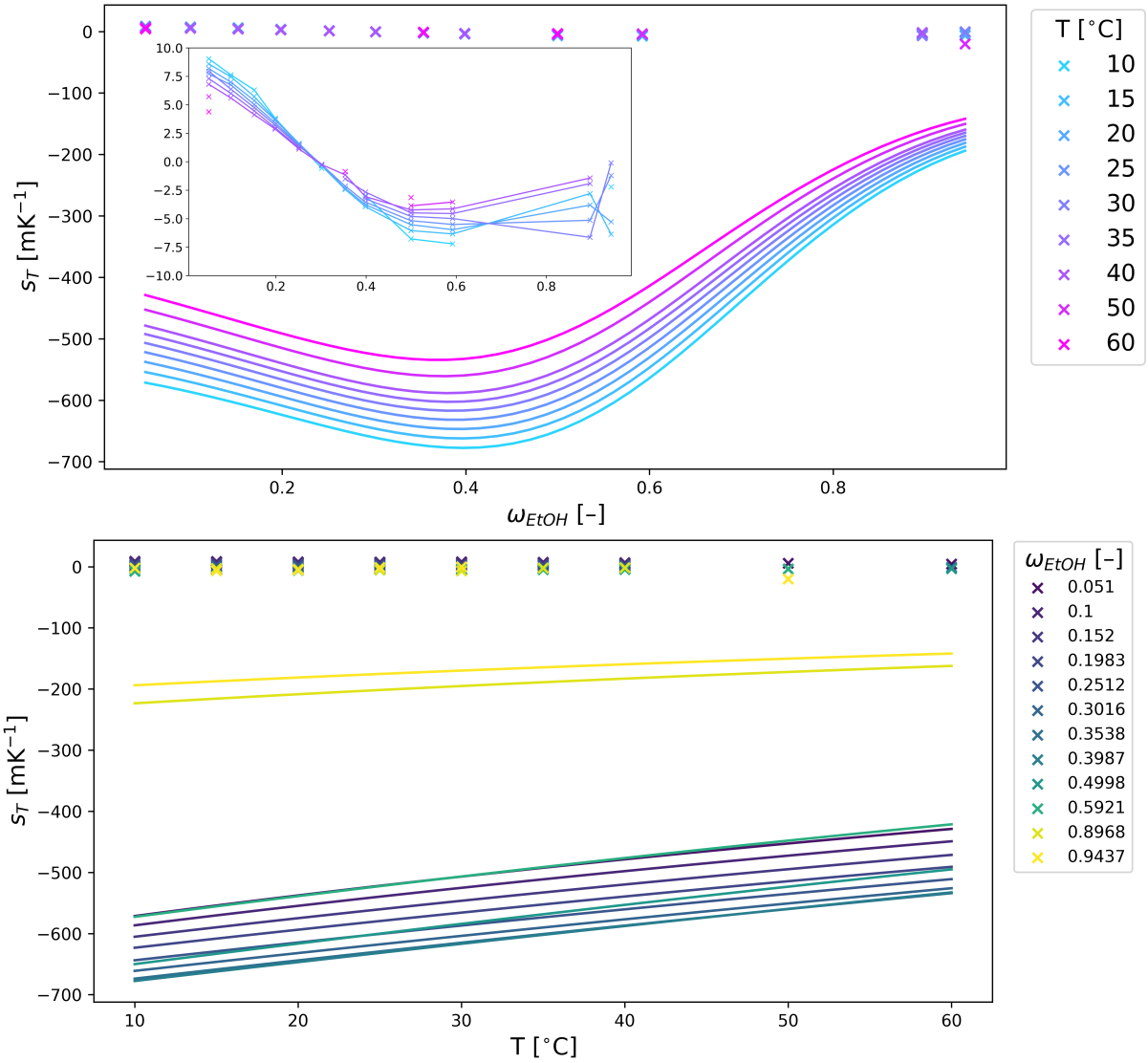
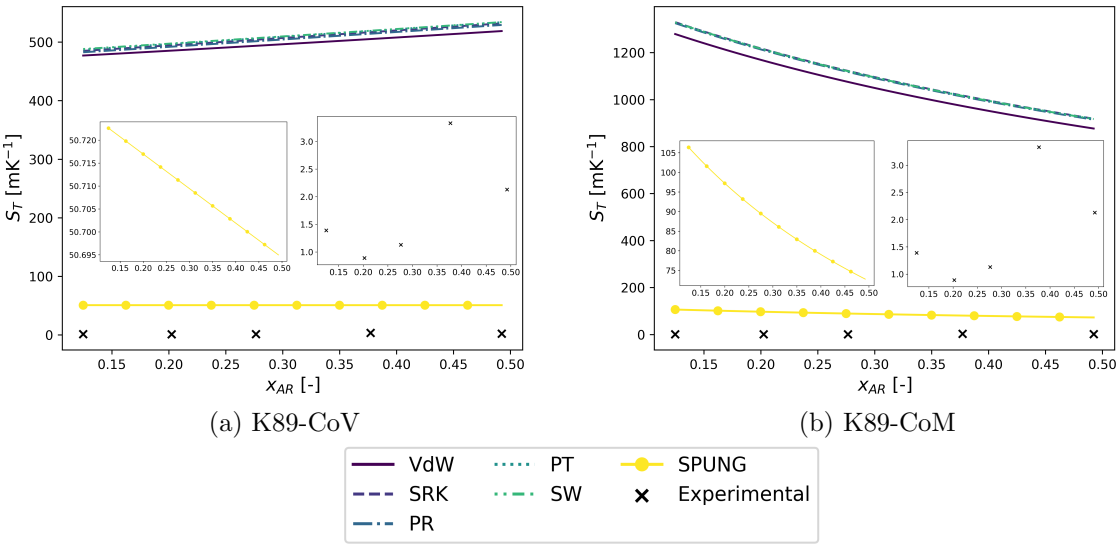


Figure 3.6: K89-CoV model predictions of Soret coefficient of ethanol in water using the SRK-CPA EoS. Conditions are 1 atm, various temperatures and compositions. Marks are experimental data, lines display predicted Soret coefficient. Inset shows a magnified ordinate axis. ω_{ETOH} is the weight fraction of ethanol. Experimental data is found in Table C.1.

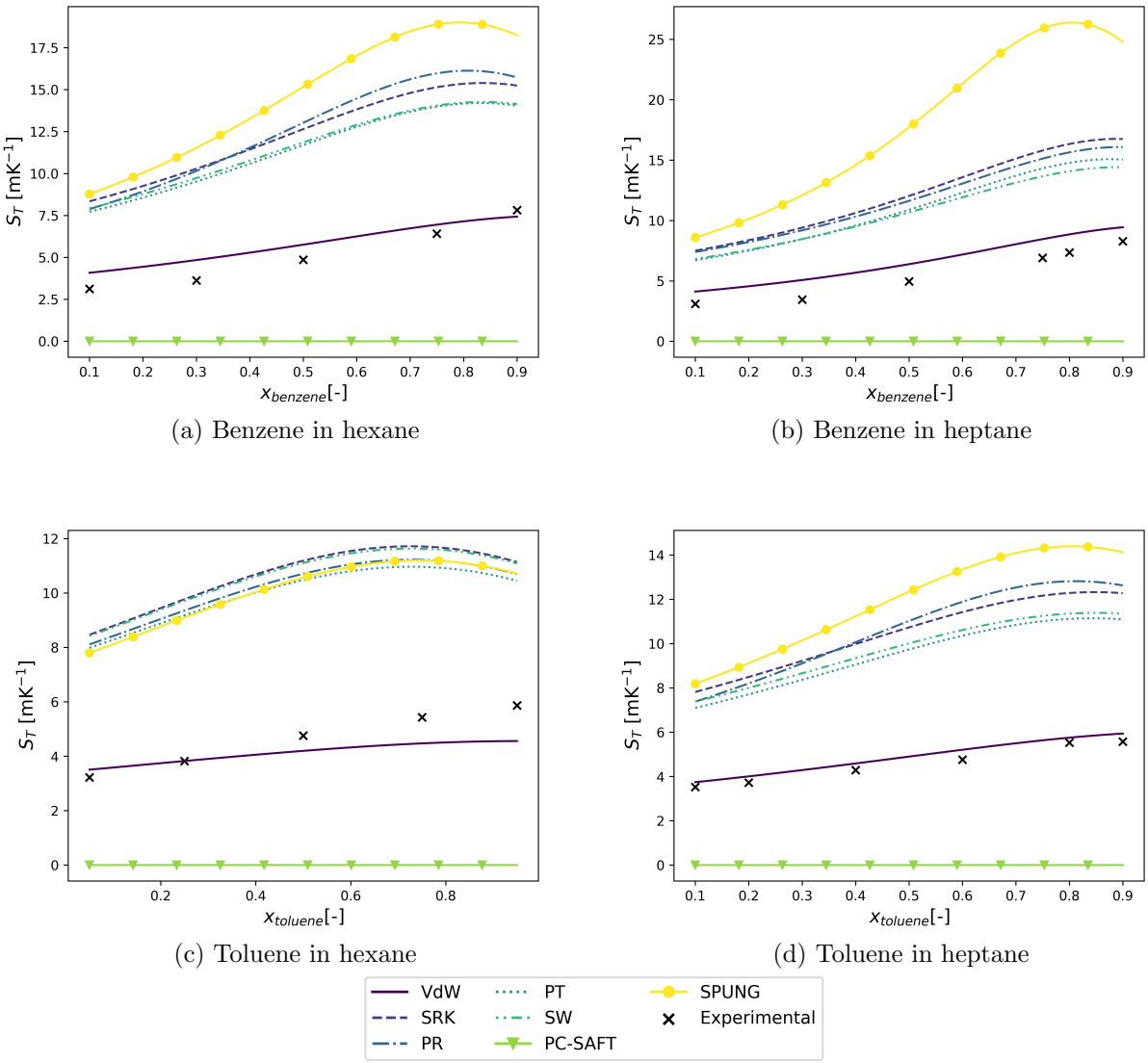
Using the M-K89-CoM model produces notably different results. As shown in Figure 3.9, using the center of mass frame of reference reduced the predicted Soret coefficient such that the SRK, PR, PT and SW equations of state provide better fits better to the experimental data than the VdW EoS. Additionally, M-K89-CoM supplied with PC-SAFT predicted the correct sign of the Soret coefficient, but with values in the order of 0.5-1.3 µK^{-1}, following approximately the same trend as the values predicted when using the SPUNG EoS.



(a) K89-CoV (b) K89-CoM

Figure 3.7: K89-CoV (a) and -CoM (b) predicted Soret coefficient of argon in methane at 88 K, 1 atm. Marks are experimental data, lines are predicted values equations of state as indicated in the legend. Insets show trend in experimental data and predicted trend using the SPUNG EoS. Experimental data is found in Table C.6.

The trends found in Figures 3.8 and 3.9, are found also when varying temperature. As shown in Figures 3.10-3.13, the M-K89-CoV model gives accurate predictions for aromate/n-alkane systems when using the VdW EoS, while using SRK, PT, PR, SW and SPUNG leads to the model over-predicting the Soret coefficient. The M-K89-CoM model, on the other hand gives predictions that agree best with experimental data when using the SW and SPUNG EoS, and generally under-predicts the Soret coefficient.



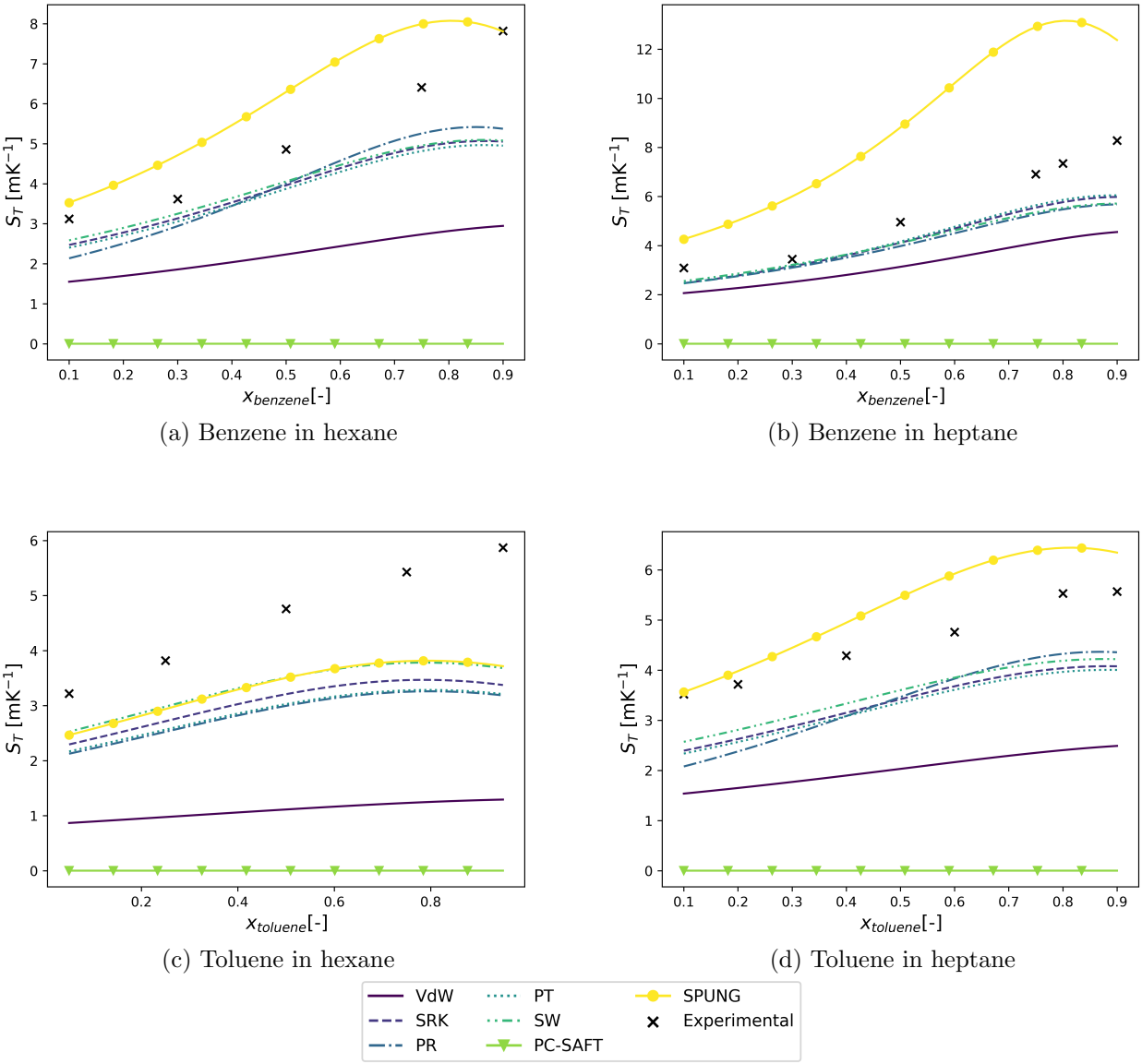
(a) Benzene in hexane

(b) Benzene in heptane

(c) Toluene in hexane

(d) Toluene in heptane

Figure 3.8: M-K89-CoV predicted Soret coefficient in several aromate/n-alkane mixtures at 298 K, 1 atm. Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display the predicted Soret coefficient using different EoS as indicated in the legend. Insets show experimental data found in Table C.3.



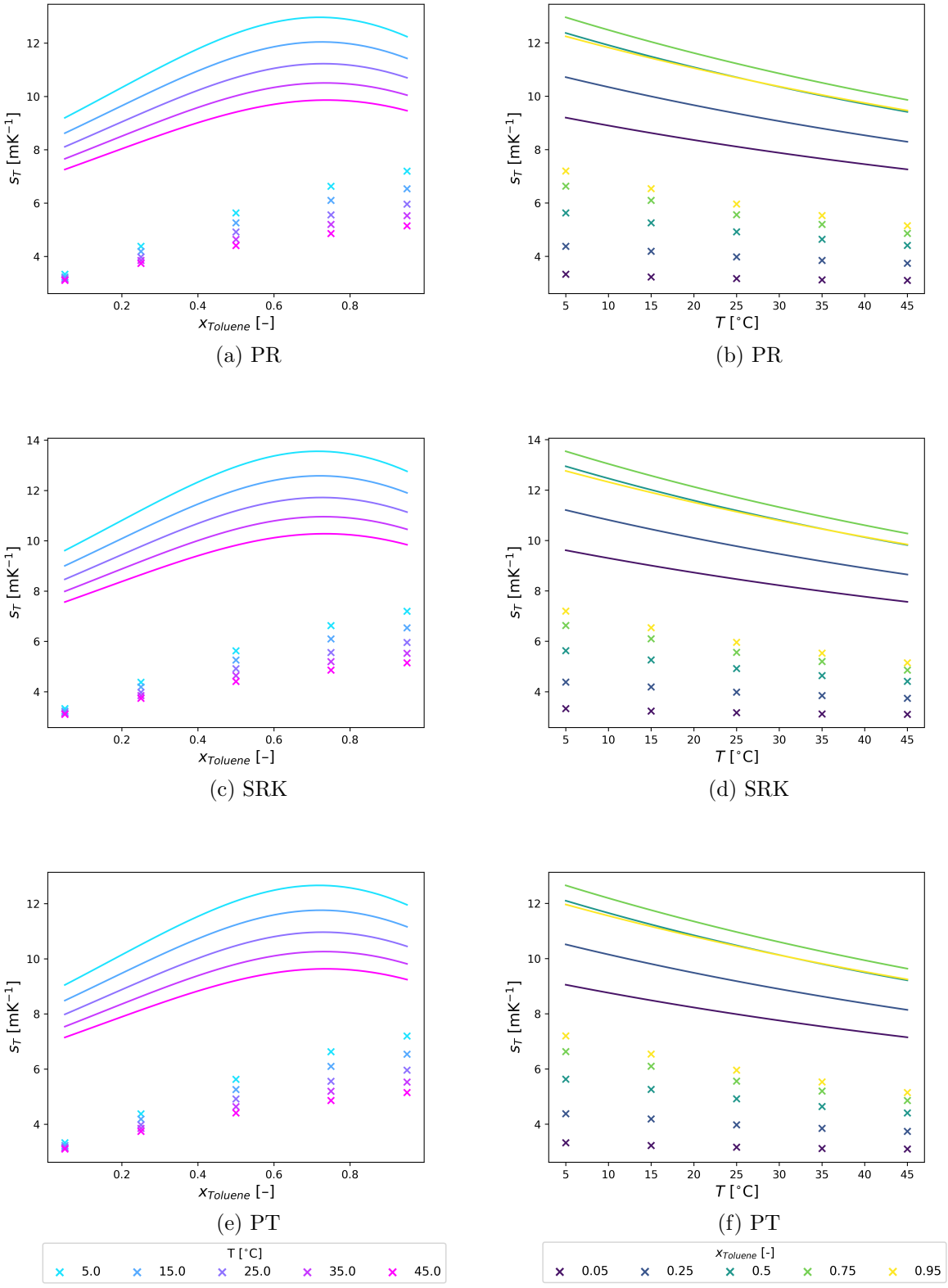
(a) Benzene in hexane

(b) Benzene in heptane

(c) Toluene in hexane

(d) Toluene in heptane

Figure 3.9: M-K89-CoM predicted Soret coefficient in several aromate/n-alkane mixtures at 298 K, 1 atm. Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display the predicted Soret coefficient using different EoS as indicated in the legend. Insets show experimental data found in Table C.3.

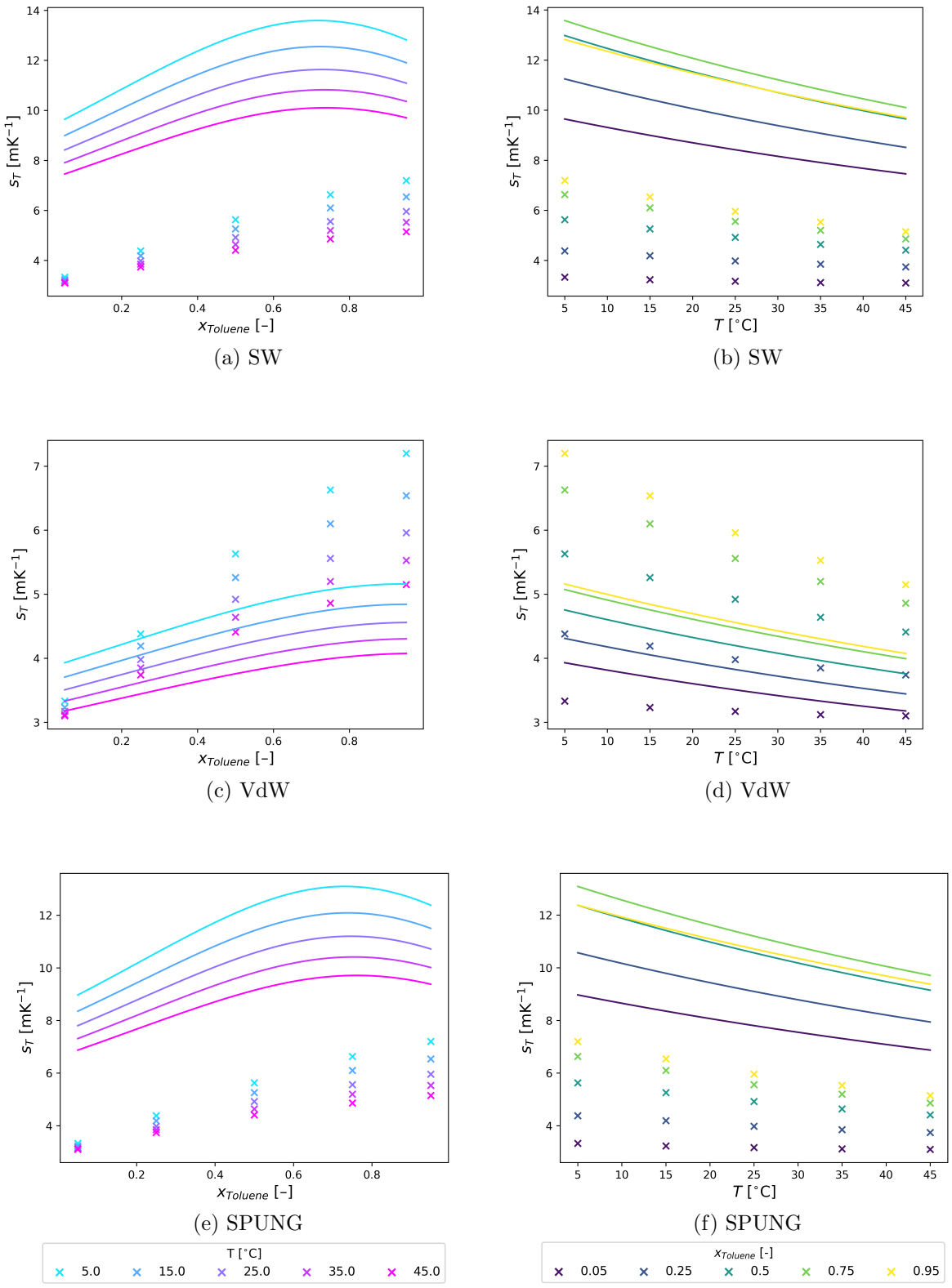


(a) PR (b) PR

(c) SRK (d) SRK

(e) PT (f) PT

Figure 3.10: M-K89-CoV predictions of Soret coefficient of toluene in hexane using a-b) PR, c-d) SRK, e-f) PT. Conditions are 1 atm and various temperatures and compositions. Marks are experimental data, lines are model predictions. Experimental data is found in Table C.2.

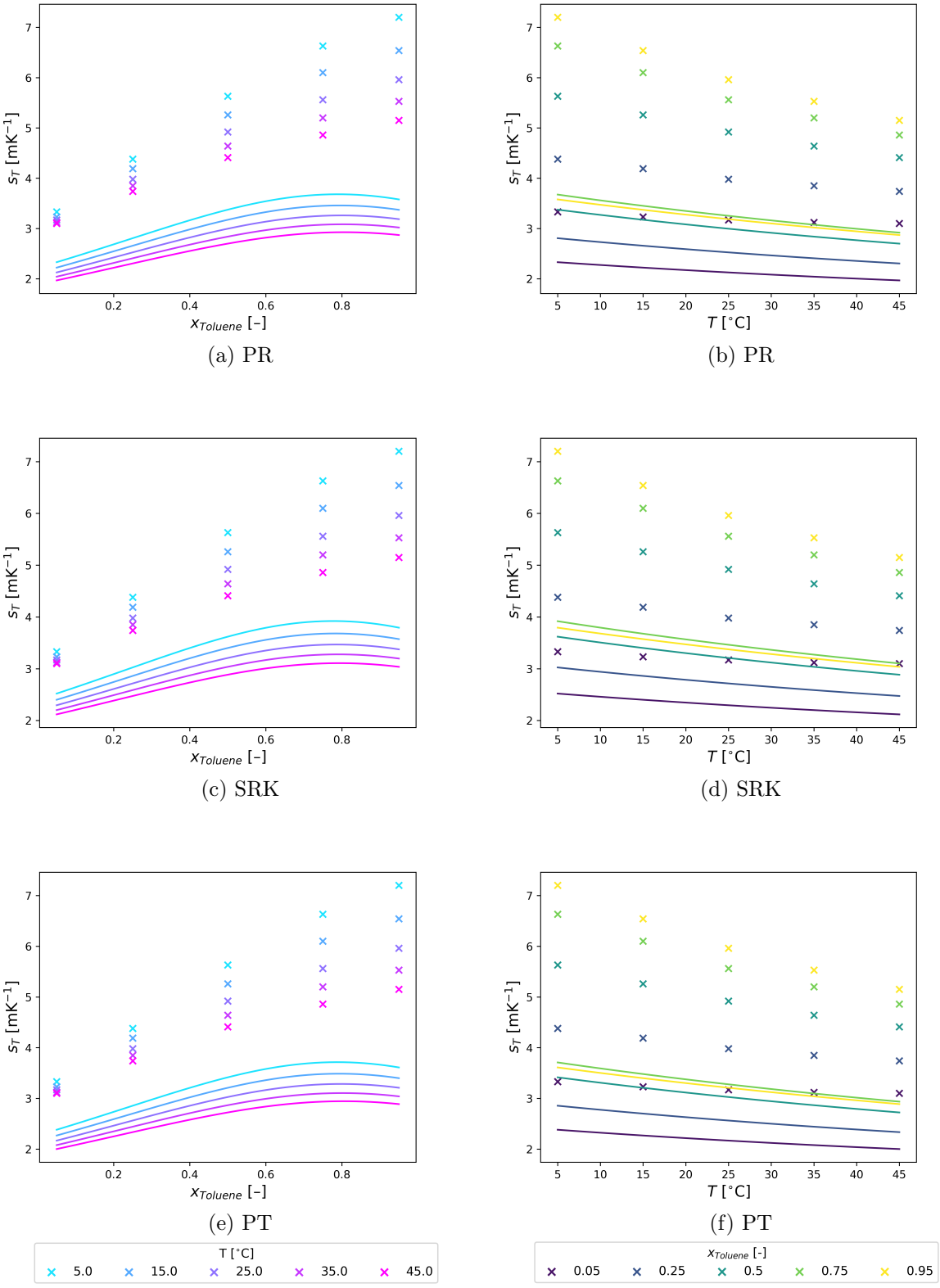


(a) SW (b) SW

(c) VdW (d) VdW

(e) SPUNG (f) SPUNG

Figure 3.11: M-K89-CoV predictions of Soret coefficient of toluene in hexane using a-b) SW, c-d) VdW, e-f) SPUNG. Conditions are 1 atm and various temperatures and compositions. Marks are experimental data, lines are model predictions. Experimental data is found in Table C.2.

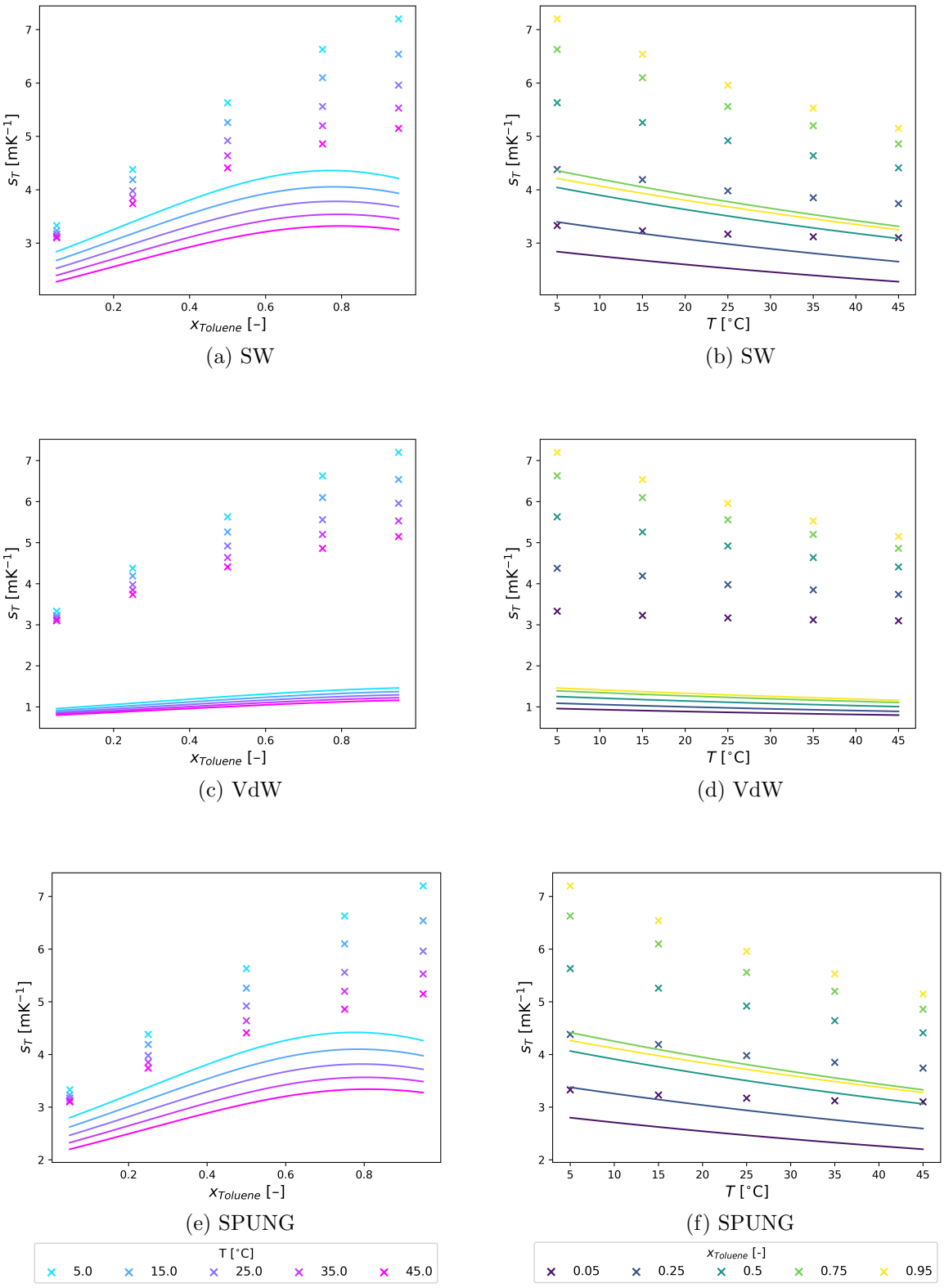


(a) PR (b) PR

(c) SRK (d) SRK

(e) PT (f) PT

Figure 3.12: M-K89-CoM predictions of Soret coefficient of toluene in hexane using a-b) PR, c-d) SRK, e-f) PT. Conditions are 1 atm and various temperatures and compositions. Marks are experimental data, lines are model predictions. Experimental data is found in Table C.2.



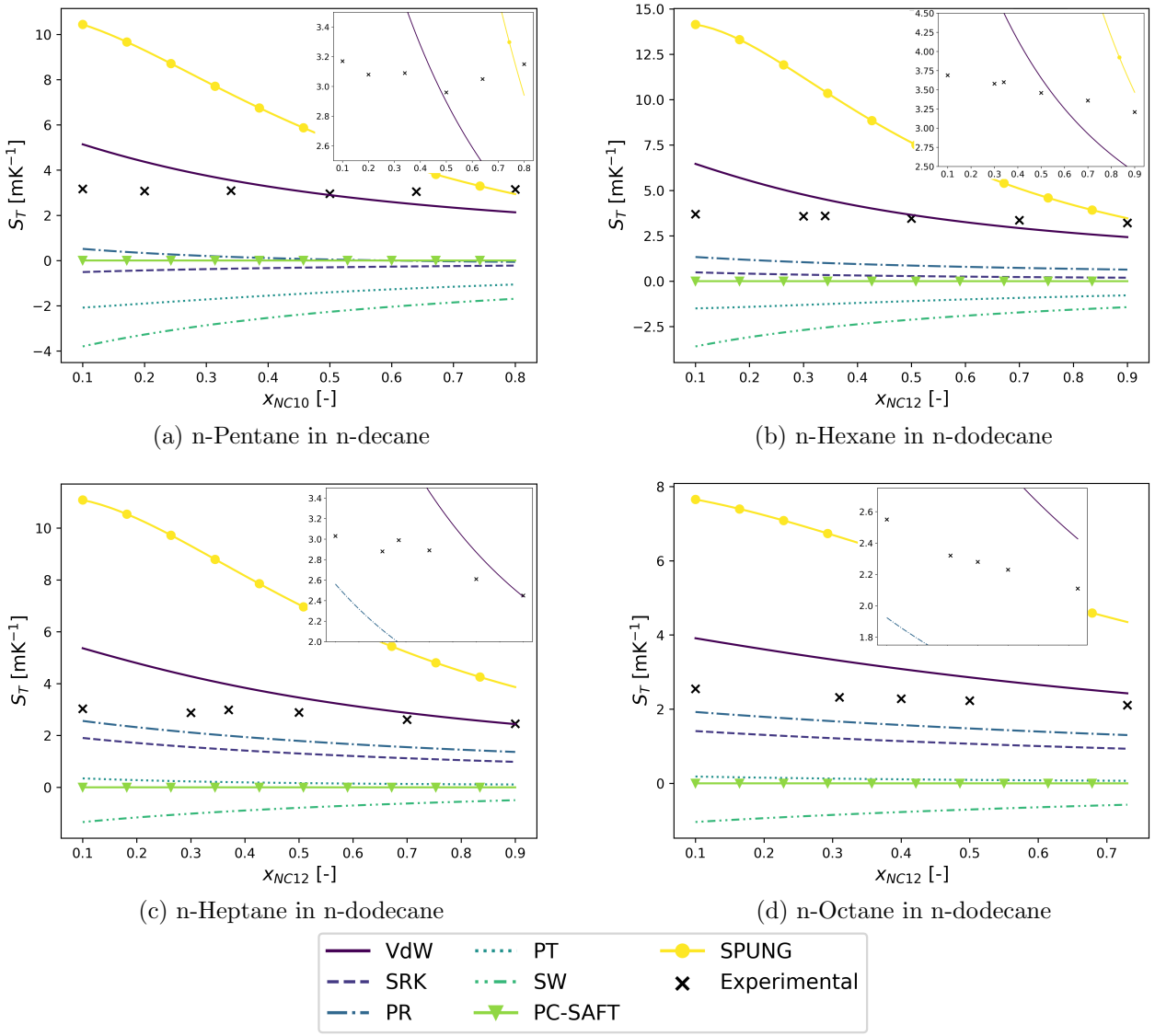
(a) SW (b) SW

(c) VdW (d) VdW

(e) SPUNG (f) SPUNG

Figure 3.13: M-K89-CoM predictions of Soret coefficient of toluene in hexane using a-b) SW, c-d) VdW, e-f) SPUNG. Conditions are 1 atm and various temperatures and compositions. Marks are experimental data, lines are model predictions. Experimental data is found in Table C.2.

In the n-alkane/n-alkane mixtures shown in Figure 3.14 it is clear that the M-K89-CoV model gives predictions that to are in agreement with experimental data when supplied with the VdW EoS. Additionally, SPUNG, PR and SRK EoS allowed prediction of the correct sign, trend and order of magnitude. Deviation from experimental data increases as the components of the mixture become more similar. When the PC-SAFT EoS is supplied, predicted values lie in the range $2.5\,\mu K^{-1}$ to $0.9\,\mu K^{-1}$ for the n-decane/n-pentane mixture; $1.8\,\mu K^{-1}$ to $0.7\,\mu K^{-1}$ for the n-dodecane/n-hexane mixture; $1.2\,\mu K^{-1}$ to $0.6\,\mu K^{-1}$ for the n-dodecane/n-heptane mixture, and $0.7\,\mu K^{-1}$ to $0.5\,\mu K^{-1}$ for the n-dodecane/n-octane mixture. The predictions when using PC-SAFT decreased with increasing mole fraction of the heaviest component, following the trend of the experimental data.



(a) n-Pentane in n-decane

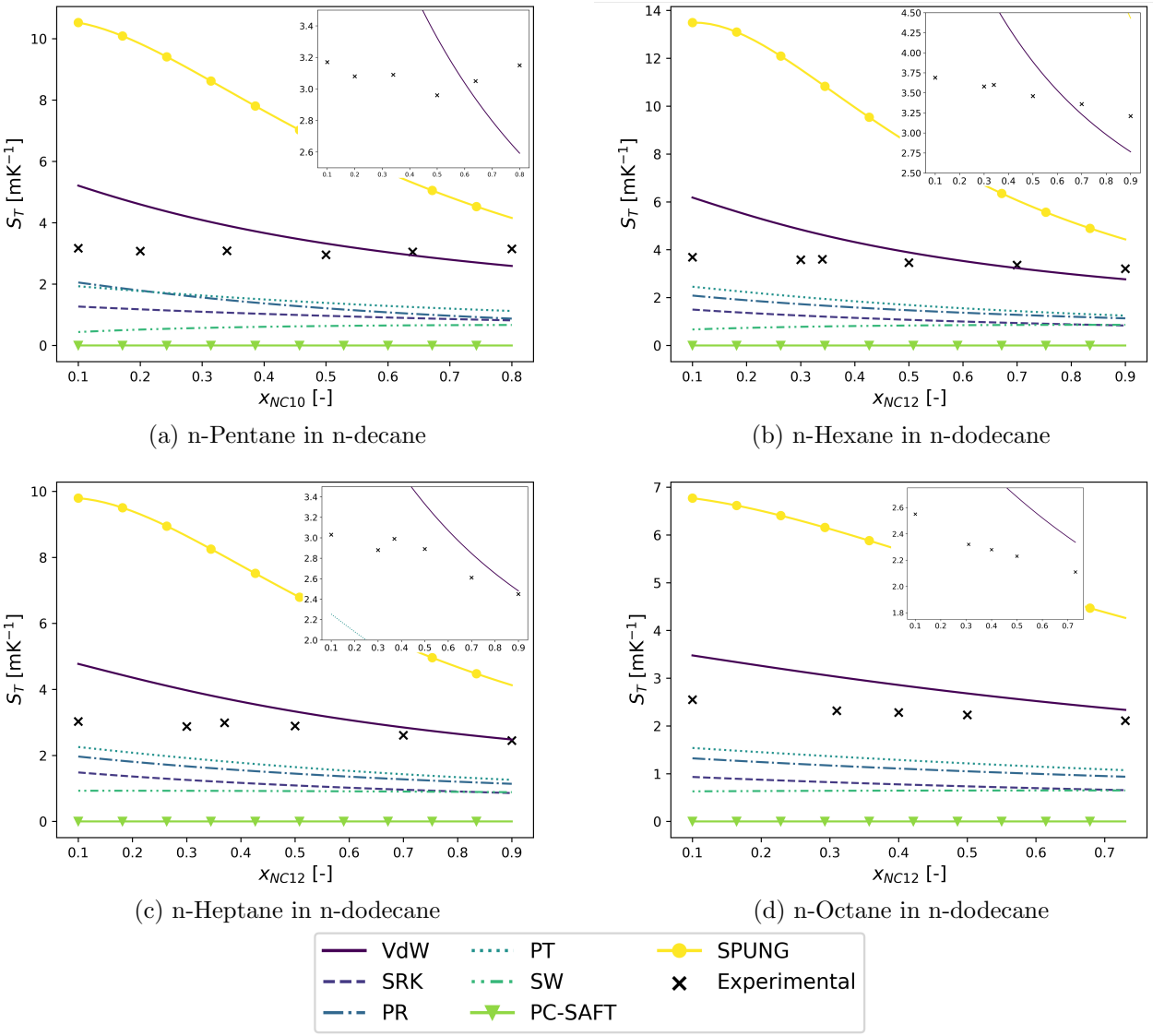
(b) n-Hexane in n-dodecane

(c) n-Heptane in n-dodecane

(d) n-Octane in n-dodecane

Figure 3.14: M-K89-CoV model prediction of Soret coefficient in several n-alkane/n-alkane mixtures at 298 K, 1 atm, from Table C.4. Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display Soret coefficient as predicted using EoS as indicated in the legend. Insets show magnification of ordinate axis.

Using the center of mass frame of reference improved the model predictions for the n-alkane/n-alkane mixtures, as shown in Figure 3.15. The M-K89-CoM model predicted the correct sign, trend and order of magnitude using all equations of state with the exeption of PC-SAFT. The latter gave slightly decreased predictions that lie in the ranges: 1.3-0.02 μK^{-1} for the n-decane/n-pentane mixture; 0.6$-$0.1 μK^{-1} for the n-dodecane/n-hexane mixture; 0.2 to $-0.1\,\mu K^{-1}$ for the n-dodecane/n-heptane mixture, and 0.05 to $-0.07\,\mu K^{-1}$ for the n-dodecane/n-octane mixture. In all cases of using the PC-SAFT EoS the predicted value decreased with increasing mole fraction of the heavier component.



(a) n-Pentane in n-decane

(b) n-Hexane in n-dodecane

(c) n-Heptane in n-dodecane

(d) n-Octane in n-dodecane

Figure 3.15: M-K89-CoM model prediction of Soret coefficient in several n-alkane/n-alkane mixtures at 298 K, 1 atm, from Table C.4. Soret coefficient refers to the first component in the mixture. Marks are experimental data, lines display Soret coefficient predicted using EoS as indicated in the legend. Insets show magnification of experimental data.

The M-K89 model, just as the K89 model, gave divergent results for the ethanol-water system when supplied with other EoS than CPA. This was expected for the same reasons as was the case for the K89 model. the results of these runs are not displayed here but are noted to be similar to those displayed in Figure 3.5. However, when supplied with the CPA EoS, M-K89 produced results closer to experimental values than the K89 model, as shown for M-K89-CoV in Figure 3.16. Using the center of mass frame of reference i.e. the M-K89-CoM model, moved predictions further towards the experimental values, while the model still failed to capture the change in sign of the Soret coefficient. The M-K89-CoM predictions are displayed in Figure 3.17.

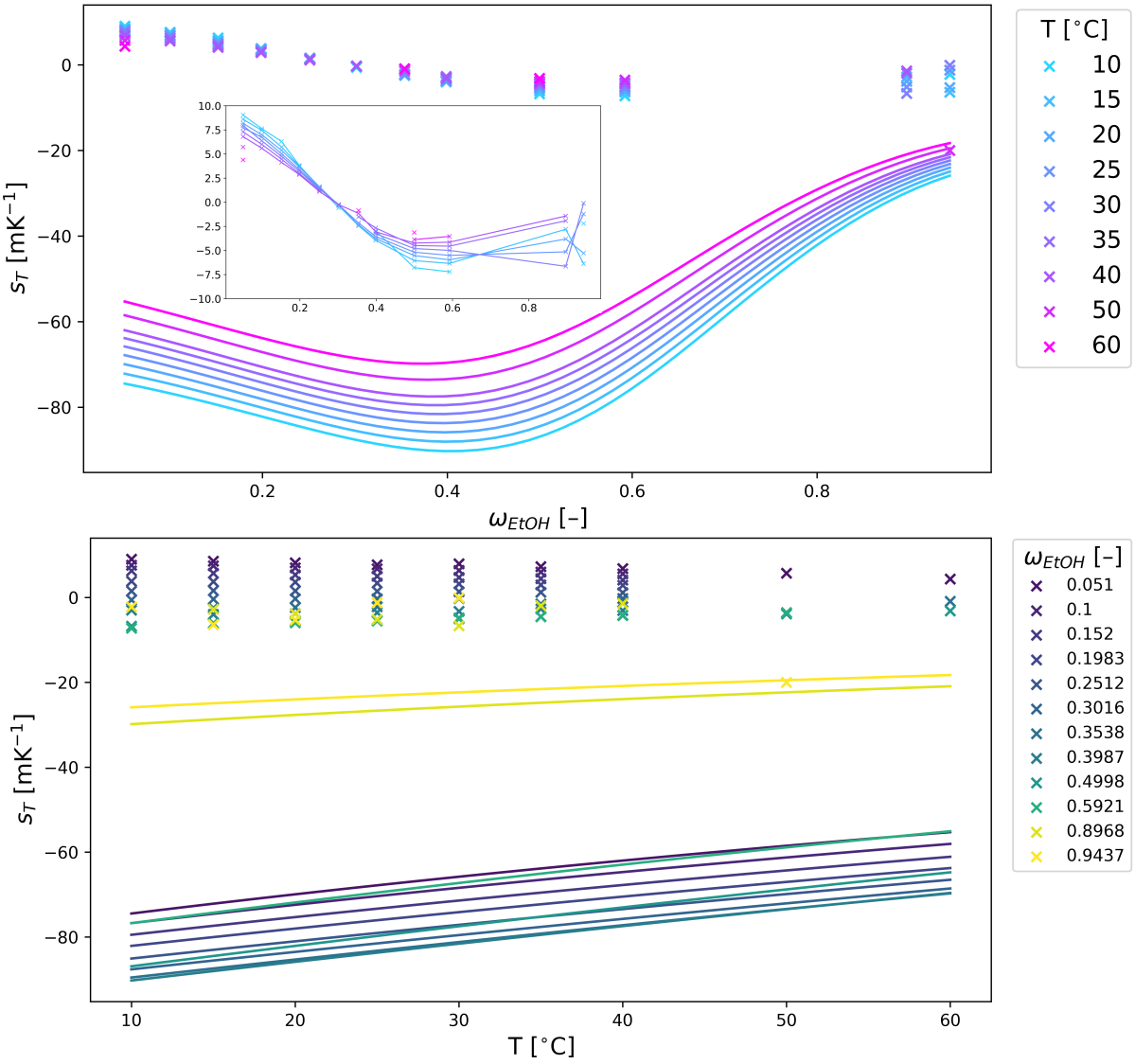


Figure 3.16: M-K89-CoV predicted Soret coefficient of ethanol in water using SRK-CPA EoS, at 1 atm, various temperatures and compositions. Marks are experimental data, lines display predicted Soret coefficient. Inset shows experimental data with lines to clearly display trends. ω_{ETOH} is the weight fraction of ethanol. Experimental data is found in Table C.1.

All models failed to accurately simulate the Soret effect in the isopropanol water system, as displayed for the M-K89-CoV model in Figure 3.18. Flash calculations by the ThermoPack method `two_phase_tpflash` revealed that all EoS predicted a two-phase region in the mixture, and that the area in which the model diverged when supplied with different EoS coincided with the edge of the two phase region, where an inflection point in the Gibbs energy surface is expected to be found.

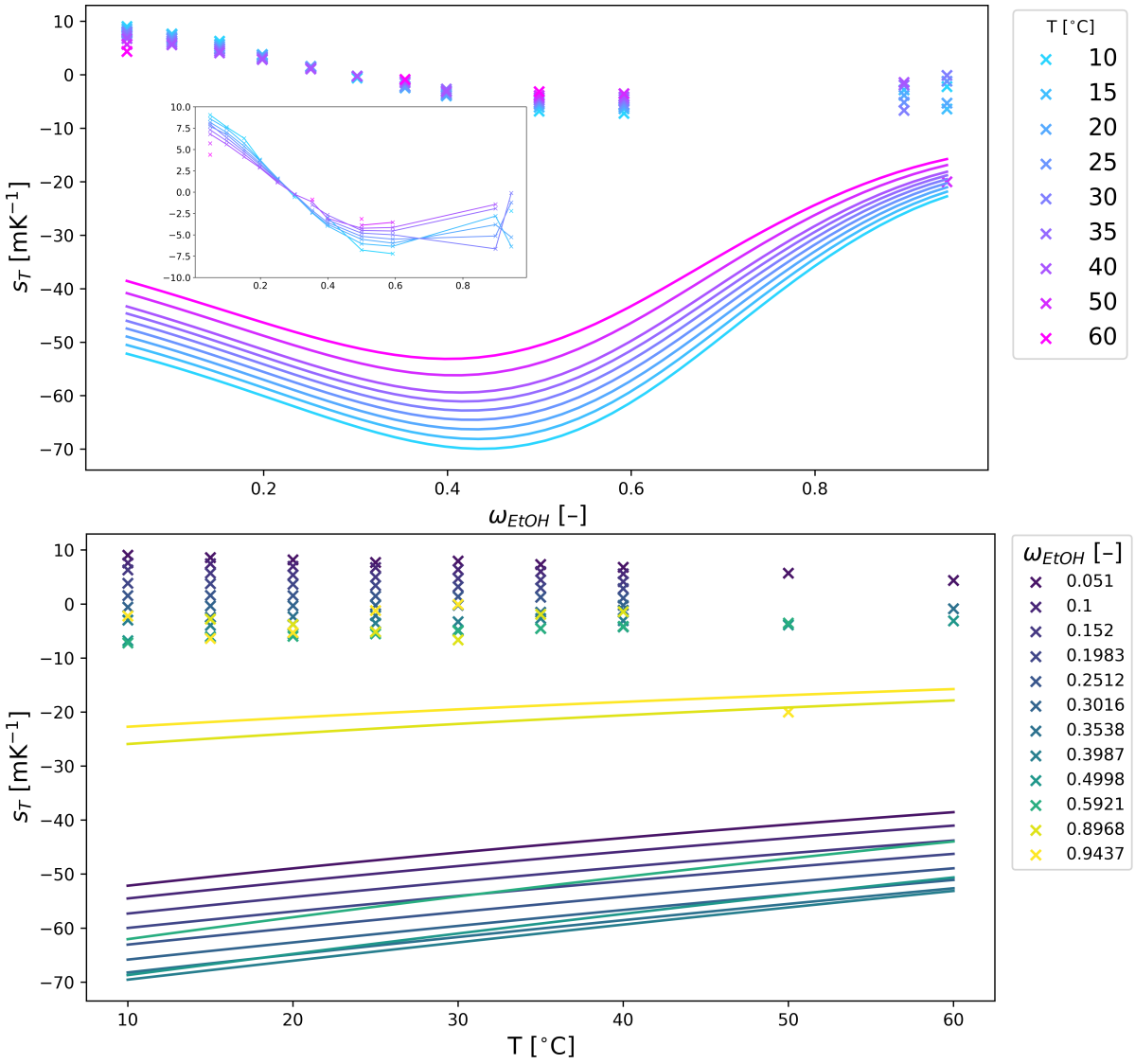


Figure 3.17: M-K89-CoM predicted Soret coefficient of ethanol in water using SRK-CPA EoS, at 1 atm, various temperatures and compositions. Marks are experimental data, lines display predicted Soret coefficient. Inset shows experimental data with lines to clearly display trends. ω_{ETOH} is the weight fraction of ethanol. Experimental data is found in Table C.1.

3.3 Kempers-01

The K01 model only differs from the M-K89 in the existence of a kinetic contribution. This section will therefore focus primarily on the behaviour of the kinetic term as opposed to the difference between the K01-CoV and the K01-CoM models. As it turns out, the kinetic contribution is negligible in most of the systems tested, and the results produced by the K01 model are therefore in most cases indistinguishable from those produced by

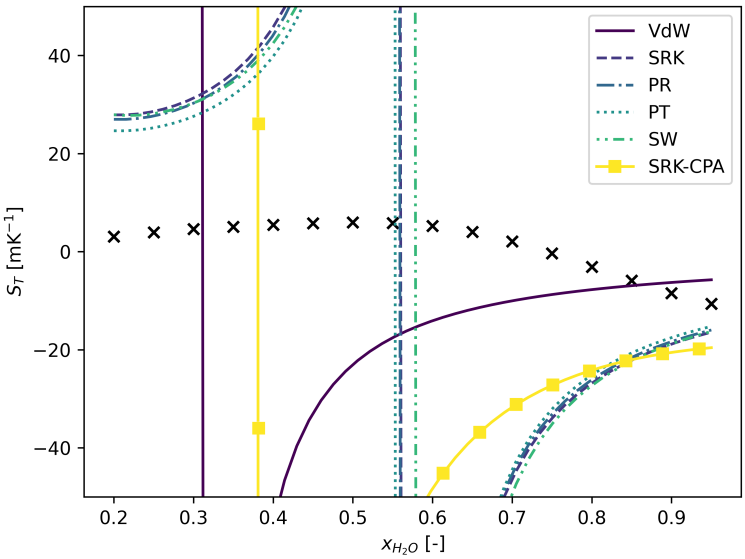
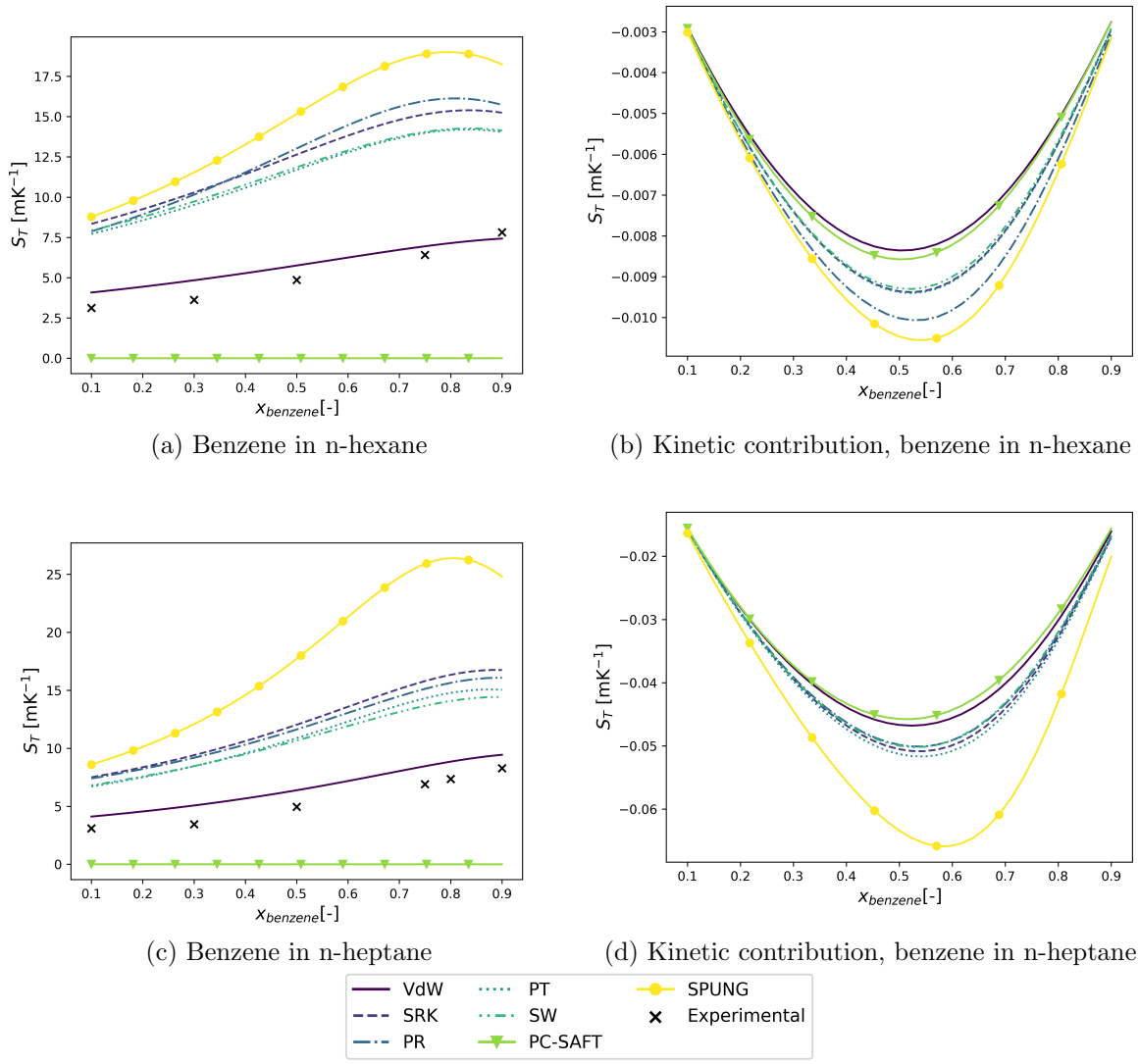


Figure 3.18: M-K89-CoV model prediction of Soret coefficient of isopropanol in water, marks are experimental data. Lines show model predictions for 1-propanol using EoS as indicated in the legend. Experimental data is found in Table C.5.

the M-K89 model. The kinetic contribution is reported as

$$S_{T,i}^{kin} = \frac{R\alpha_T^0}{x_i \frac{\partial \mu_i}{\partial x_i}}. \tag{3.1}$$

That is, the contribution to the Soret coefficient resulting from the last term in equation (1.9).



(a) Benzene in n-hexane

(b) Kinetic contribution, benzene in n-hexane

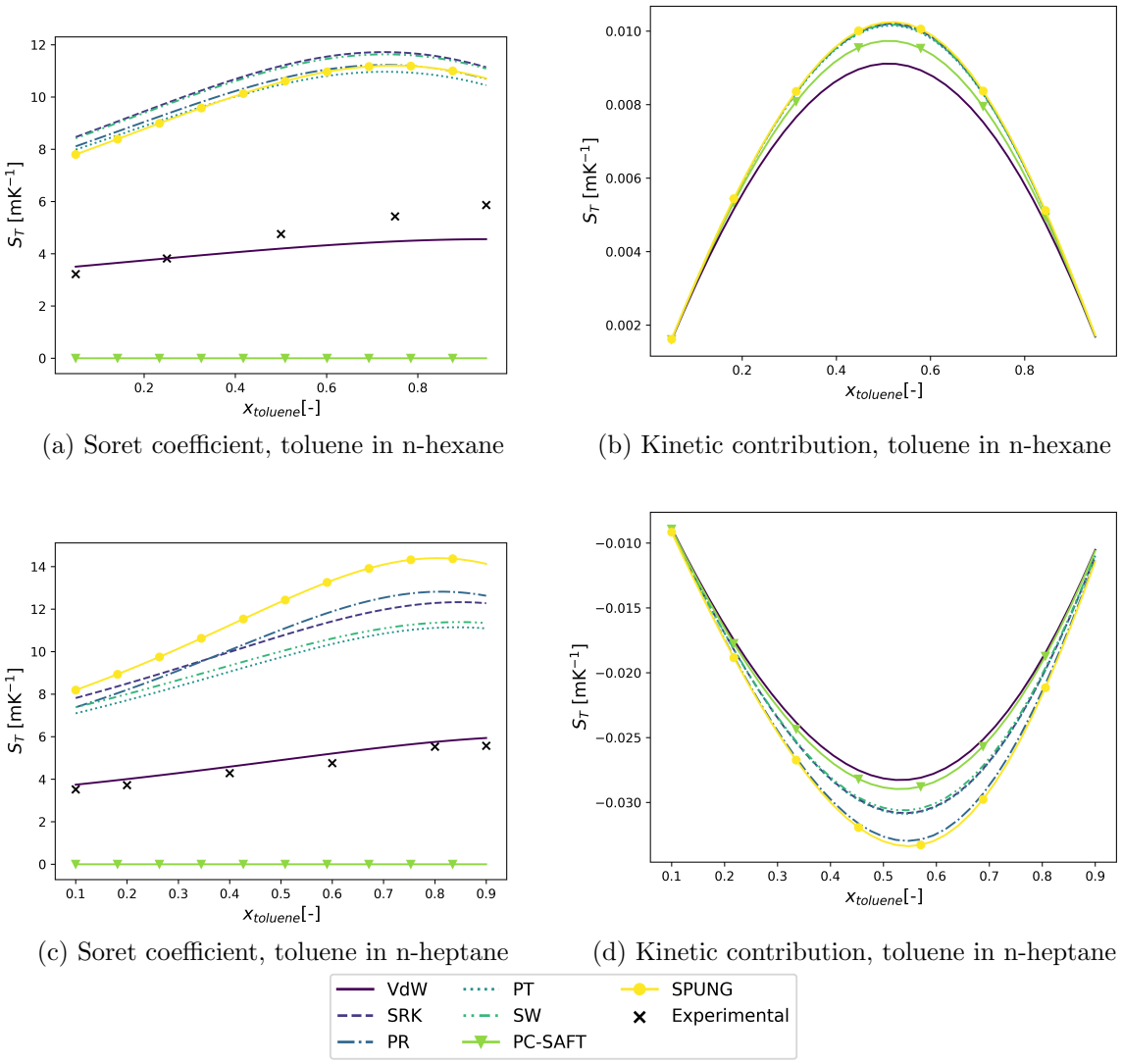
(c) Benzene in n-heptane

(d) Kinetic contribution, benzene in n-heptane

Figure 3.19: (a) and (c): K01-CoM predicted Soret coefficient of benzene in n-hexane and n-heptane respectively, marks are experimental data, lines are predicted values using EoS as indicated in the legend. (b) and (d): Kinetic contribution to the Soret coefficient of benzene in n-hexane and n-heptane respectively, computed using EoS as indicated in the legend. Experimental data found in Table C.3

It is clear from Figures 3.19 and 3.20 that the kinetic contribution serves to modulate the over-predicted curvature by the thermodynamic contribution, except in the case of the toluene/n-hexane mixture, but is too small to have a notable effect on the result.

In the n-alkane/n-alkane mixtures shown in Figures 3.21 and 3.22, the kinetic contribution is negligible in all cases. However, it is noted that the contribution serves to bring the prediction marginally closer to the experimental data when using the VdW and SPUNG equations of state, while pushing predictions by other EoS further away from the measured values.



(a) Soret coefficient, toluene in n-hexane

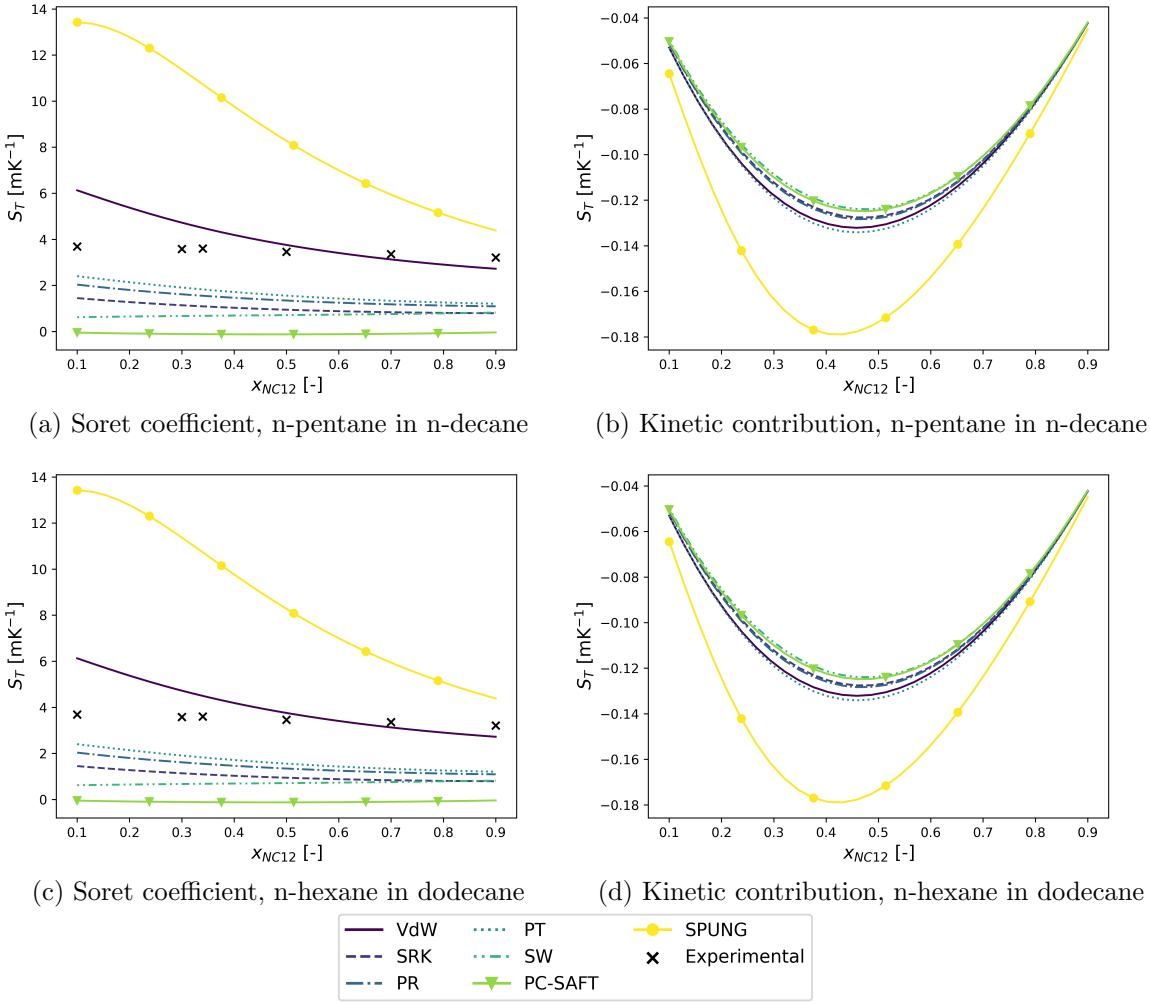
(b) Kinetic contribution, toluene in n-hexane

(c) Soret coefficient, toluene in n-heptane

(d) Kinetic contribution, toluene in n-heptane

Figure 3.20: (a) and (c): K01-CoM model prediction of Soret coefficient of toluene in n-hexane and n-heptane respectively, marks are experimental data, lines are predicted values using EoS as indicated in the legend. (b) and (d): Kinetic contribution to the Soret coefficient of toluene in n-hexane and n-heptane respectively, computed using EoS as indicated in the legend. Experimental data found in Table C.3

In the argon/methane mixture, the kinetic contribution is far larger, but still small compared to the thermodynamic contribution. As shown in Figure 3.23, including the kinetic term brings model predictions closer to the experimental values when using the VdW or SPUNG EoS.



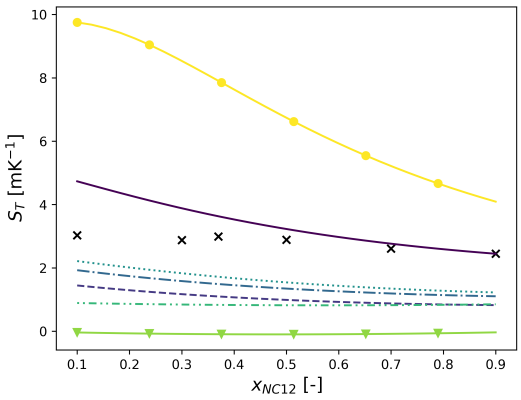
(a) Soret coefficient, n-pentane in n-decane

(b) Kinetic contribution, n-pentane in n-decane

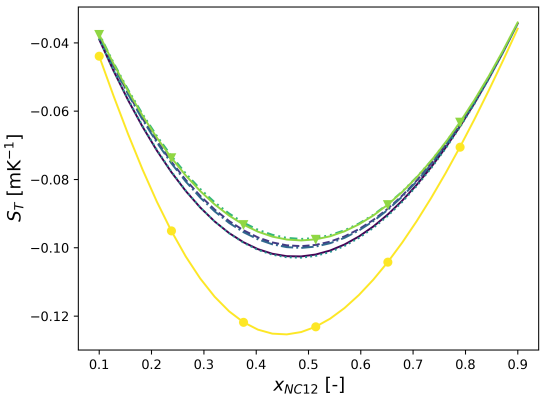
(c) Soret coefficient, n-hexane in dodecane

(d) Kinetic contribution, n-hexane in dodecane

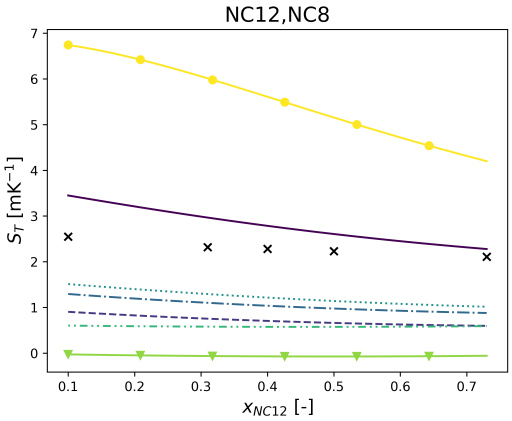
Figure 3.21: (a) and (c): K01-CoV predictions of the Soret coefficient of n-pentane in n-decane and n-hexane in n-dodecane respectively, marks show experimental data, lines show predictions using EoS as indicated in the legend. (b) and (d) show the kinetic contribution to the Soret coefficient of n-pentane in n-decane and n-hexane in n-dodecane respectively. Experimental data is found in Table C.4



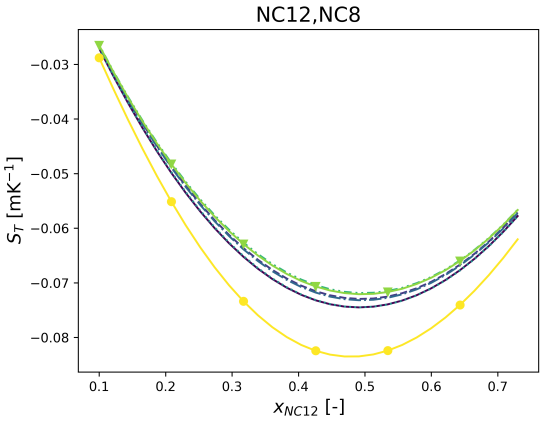
(a) Soret coefficient, n-heptane in n-dodecane



(b) Kinetic contribution, n-heptane in n-dodecane

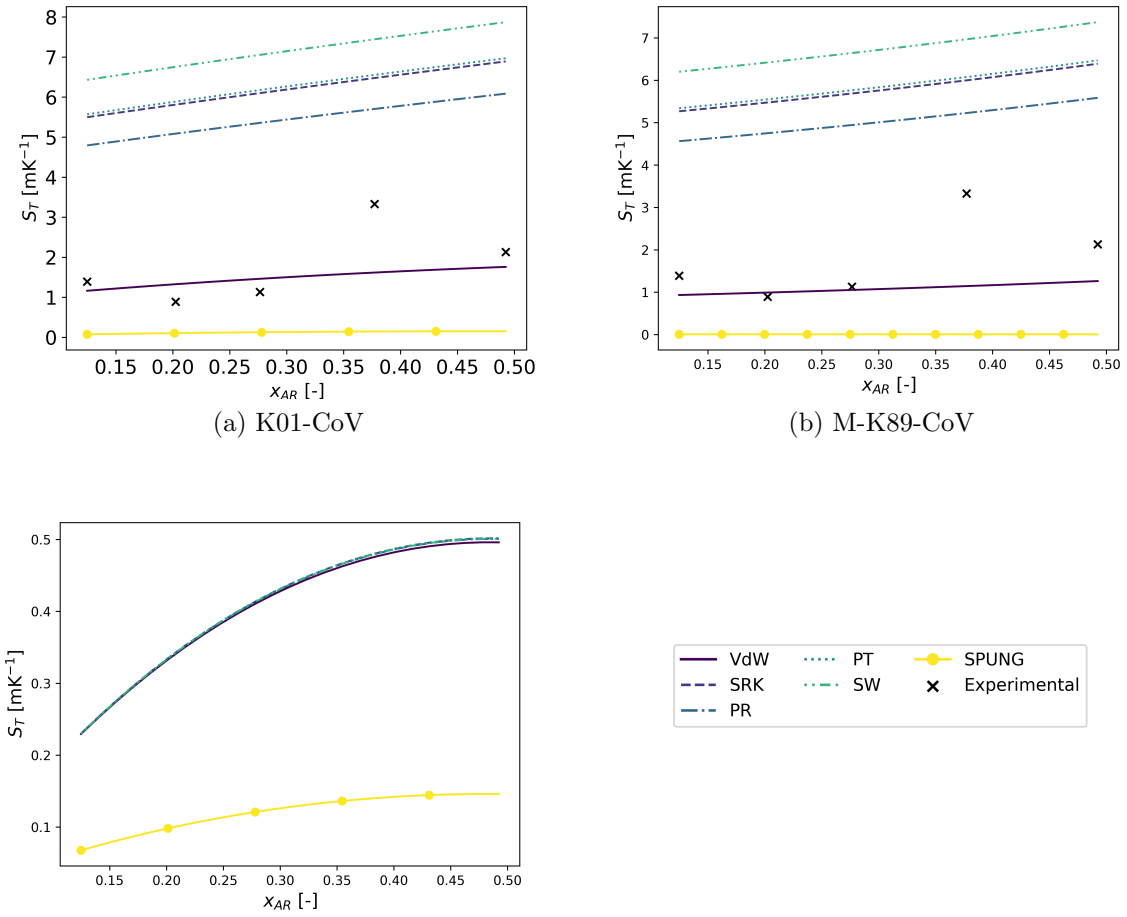


(c) Soret coefficient, n-octane in n-dodecane



(d) Kinetic contribution, n-octane in n-dodecane

Figure 3.22: (a) and (c): K01-CoV predicted Soret coefficient of n-heptane in n-dodecane and n-octane in n-dodecane respectively, marks show experimental data, lines show predictions using EoS as indicated in the legend. (b) and (d): Kinetic contribution to Soret coefficient of n-pentane in n-decane and n-hexane in n-dodecane respectively. Experimental data is found in Table C.4



(a) K01-CoV

(b) M-K89-CoV

(c) Kinetic contribution

Figure 3.23: Soret coefficient of methane in argon at 88 K, 1 atm. a) K01-CoV model model predictions. b) M-K89-CoV model predictions. c) Kinetic contribution to the Soret coefficient for the different EoS. Marks show experimental data, found in Table C.6

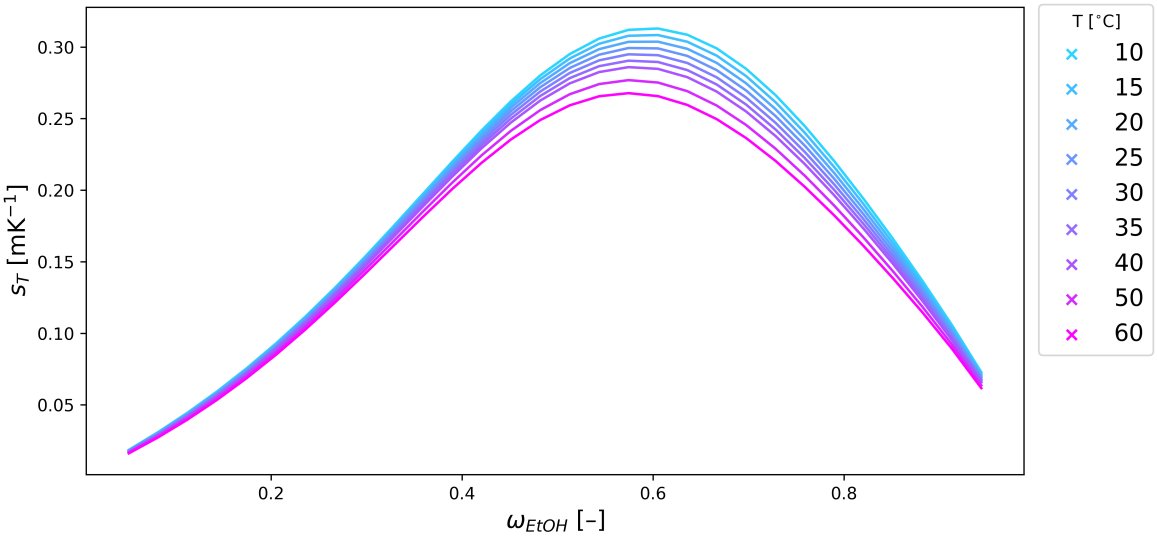


Figure 3.24: Kinetic contribution to the Soret coefficient of ethanol in water at different temperatures using the SRK-CPA EoS.

The kinetic contribution in the ethanol-water system, displayed in Figure 3.24, has an absolute value approximately two orders of magnitude smaller than the thermodynamic contribution. Therefore the the results produced by the K01 model were virtually indistinguishable from those produced by the M-K89 model.

4 Discussion

In general, the most accurate predictions of the Soret coefficient were achieved by the M-K89-CoV and K01-CoV models when supplied with the VdW equation of state. The PC-SAFT EoS consistently gave predictions very close to zero, while following the trend of the experimental data. In the only system where the kinetic contribution was non-negligible the K01-CoV model gives predictions well within experimental error when supplied with the VdW EoS.

4.1 Aromate-alkane systems

The aromate/n-alkane mixtures were modeled to reasonable precision when supplying the M-K89-CoV and K01-CoV models with the VdW EoS. When using the M-K89-CoM and K01-CoM models, the equations of state that are expected to be more accurate for hydrocarbon systems produced predictions slightly lower than the experimental data. This may be due to small convective flows being present in the systems that were measured, as the CoM models are expected to be more accurate if this is the case.[1] All equations of state predicted trends that correlate with the measured data, but are offset by various amounts. Looking at the kinetic contributions displayed in Figure 3.19, shows that the SPUNG EoS stands out, with a significantly higher absolute value for the kinetic contribution. This can only be due to a smaller predicted $\frac{\partial \mu_i}{\partial x_i}$, which again explains the large predicted Soret coefficients. An analysis of the reference EoS and reference compounds available in ThermoPack may be favourable to determine if a different combination of these would be better suited to model the aromate/alkane-systems.

Another possible explanation for the deviation of the predictions using the SRK, PR, PT and SW EoS is over-prediction of residual enthalpies, and no small scale convection being present in the system. This would also explain why the VdW EoS gave the most precise estimates when using the CoV-models, as the VdW EoS incorrectly modelled the gas phase, giving smaller residual enthalpies.

The PC-SAFT EoS was expected to model the aromate/n-alkane systems well, but consistently gave results about one order of magnitude too small, albeit following the correct trends. This may be due to default ThermoPack behaviour when parameters for one of the components is unavailable in the database, although this has not been confirmed.

Several aspects of the ethanol-water system are interesting, both the change in the sign of the Soret coefficient and the fact that an increase in temperature mitigates the effects of changing concentration, this effect is found in several alcohol-water mixtures.[31] It is clear from Figures 3.16 and 3.17 that despite the SRK-CPA EoS being known to accurately predict phase-equilibria for associating systems,[9,32] the Kempers model is unable to replicate the Soret coefficient in the system. However, it does capture the effect of increased temperature dampening the effect of changing concentration on the Soret coefficient.

4.2 Associating systems

In the ethanol-water system at the temperatures investigated, the kinetic contribution is negligible, having an absolute value approximately two orders of magnitude smaller than the calculated thermodynamic contribution. Therefore, deviations from the experimental data are most likely a result of lacking precision in the calculated partial molar enthalpies, and/or volumes.

The propanol-water system clearly displays the Kempers models' dependency on an accurate EoS. Though propanol is, according to Merck,[33] miscible in water, the SRK-CPA EoS predicts phase separation and subsequently a divergent Soret coefficient. However, it can not be concluded that that this prediction is inherently erroneous. The predicted energy of phase separation is very small, and the observed miscibility may be a result of kinetics, i.e. that the propanol-water mixture is thermodynamically unstable at certain specific concentrations at room temperature, but that slow kinetics in practice prevent phase separation.

4.3 Alkane-alkane systems

The n-alkane/n-alkane systems stand out in that the K89-CoV model gave reasonable predictions in a few cases, as shown in Figure 3.3. The kinetic contribution in these systems was calculated to be close to negligible, and both the K01-CoM and the M-K89-CoM model predicted the correct sign, trend and order of magnitude when using all EoS other than PC-SAFT. The fact the the PC-SAFT EoS consistently produces results 1-2 orders of magnitude smaller than the remaining EoS while predicting the correct trends, strengthens the belief that there is an error in its usage, rather than that the PC-SAFT EoS is poorly suited to model these systems.

Using the M-K89 and K01 models with the VdW EoS gave the estimates that most consistently fell close to experimental data, as shown in Figures 3.15 and 3.21. Again, the VdW EoS modeled the gas phase in all cases, except the n-octane/n-dodecane mixture, which is also the case in which the model deviates most from experimental data. This further strengthens the claim that the deviation from measurements when using other EoS is due to small errors in the residual partial molar enthalpy.

4.4 The liquified gas system

The methane-argon mixture was clearly best modeled by the K01-CoV model using the VdW EoS. In this mixture the VdW and SPUNG EoS modeled the gas phase, while the remaining EoS correctly modeled the liquid phase. The kinetic contribution was non-negligible and served to make predictions by the K01 model coincide better with experimental data than those by the M-K89 model. As in the case of the aromate/n-alkane mixtures, the EoS that model the liquid phase over-predict the Soret coefficient. This may be due to over-estimation of residual enthalpies computed by these equations of state.

5 Conclusion

The Kempers-89 model as implemented here misses nearly all predictions by a large factor. Due to the large improvement in predictions when using the Modified Kempers-89 model, and Kempers' statement that the standard state enthalpies do not have a large effect on the predicted Soret coefficient; it is assumed that this is due to an erroneous implementation of the ideal gas state enthalpies.

The Modified Kempers-89 model gave predictions of the Soret coefficient that were reasonable to some degree in non-associating mixtures. The predicted values vary heavily with the selected EoS, and with the selected frame of reference. However, when supplied with the VdW EoS, which modeled the gas phase instead of the liquid phase, the K89-CoV model consistently provided predictions that were in good agreement with experimental data.

The Kempers-01 model calculates a good estimate for the Soret coefficient in hydrocarbon systems, depending on the equation of state. It is a far more computationally demanding model than the M-K89, and gives only a marginal increase in precision when modeling liquid mixtures. The employed hard sphere radii were constants, and utilizing temperature dependent hard sphere radii may increase the kinetic contribution. The model predictions were in reasonable agreement with measurements, and using the K01-CoV model in combination with the VdW EoS consistently reproduced the correct trends, and were generally within 10-50% of experimental data points.

None of the models were capable of reproducing the behaviour of associating systems, despite being supplied with an EoS that is known to model these accurately.

All results are in agreement that the Kempers model depends heavily on the supplied equation of state, and the selected frame of reference. Especially the K01 model showed promising results for non-associating systems, that should garner further interest. Still, there is too little available data per now to conclude definitively in regard to what EoS should be used to model the Soret effect for different general classes of chemical compounds.

The written implementation of the models, interfaced with ThermoPack proved very efficient in testing a large number of equations of state at a large variety of conditions. If further research is to be done on the Kempers-models, a similar implementation should be used.

5.1 Future work

The implementation of Kempers' 2001-model is written so that it will work for multi-component systems if provided with a module that can compute α_T^0 for all components in a multi-component system. It also suffers heavily from the long computation time required for α_T^0. For future improvement, a module that computes α_T^0 should be implemented in a more computationally efficient language than Python.

Further, it may be of interest to test the effect of using temperature-dependent effective hard sphere radii to approximate α_T^0 instead of the Mie-potential σ-values that have

been used here. Verification of the computed α_T^0-values can be done using the implemented module for approximation from experimental data and the included, extensive data set.

Additionally, a wider series of equations of state and mixing rules should be tested to shed more light on the trends pointed out here. Preferably, a set of experimental Soret coefficients for a wider range of chemical compounds should be compiled, this may make it possible to conclude definitively regarding what equations of state are best suited for different systems. Measurements of Soret coefficients in gas-phase mixtures are also required to further verify the accuracy of the kinetic contribution to the K01 model.

The behaviour of the PC-SAFT EoS should be investigated more closely. This EoS is expected to perform well for a wide variety of mixtures, but failed to fall in agreement with experimental data regarding the magnitude of the Soret coefficient. Despite this, the trends produced by the PC-SAFT EoS fell in good agreement with the measured data, meaning a closer look is warranted.

The implementation of ideal gas state enthalpies in the Kempers-89 model is likely erroneous. A possible way to correct this is to add the enthalpies to the ThermoPack database, thereby also removing the need for manual correction of the absolute partial molar enthalpies.

References

[1] L. J. T. M. Kempers, The Journal of Chemical Physics **90**, 6541 (1989).

[2] L. J. T. M. Kempers, The Journal of Chemical Physics **115**, 6330 (2001).

[3] S. Kjelstrup, D. Bedeaux, E. Johannessen, and J. Gross, *Non-Equilibrium Thermodynamics for Engineers*, 2nd ed. (Wold Scientific Publishing Co. Pte. Ltd., 57 Shelton Street, Covent Garden, London WC2H 9HE, 2017).

[4] U. Nations, "Envision2030 goal 7: Affordable and clean energy," (2020).

[5] W. Kohler and K. Morozov, Journal of Non-Equilibrium Thermodynamics **41**, 151 (2016).

[6] L. Onsager, Phys. Rev. **37**, 405 (1931).

[7] M. Ibrahim, G. Skaugen, and I. S. Ertesvåg, Energy Procedia **51**, 353 (2014), 7th Trondheim Conference on CO2 capture, Transport and Storage (2013).

[8] Øivind Wilhelmsen, G. Skaugen, O. Jørstad, and H. Li, Energy Procedia **23**, 236 (2012), the 6th Trondheim Conference on CO2 Capture, Transport and Storage.

[9] G. M. Kontogeorgis, M. L. Michelsen, G. K. Folas, S. Derawi, N. von Solms, and E. H. Stenby, Industrial & Engineering Chemistry Research **45**, 4855 (2006).

[10] M. Ibrahim, G. Skaugen, and I. S. Ertesvåg, Energy Procedia **51**, 353 (2014), 7th Trondheim Conference on CO2 capture, Transport and Storage (2013).

[11] J. Gross and G. Sadowski, Industrial & Engineering Chemistry Research **40**, 1244 (2001), https://doi.org/10.1021/ie0003887 .

[12] W. Chapman, K. Gubbins, G. Jackson, and M. Radosz, Fluid Phase Equilibria **52**, 31 (1989).

[13] S. Benzaghou, J. P. Passarello, and P. Tobaly, Fluid Phase Equilibria **180**, 1 (2001).

[14] M. Rahman and M. Saghir, International Journal of Heat and Mass Transfer **73**, 693 (2014).

[15] Q. Galand, S. Van Vaerenbergh, W. Kohler, O. Khlybov, T. Lyubimova, A. Mialdun, I. Ryzhkov, V. Shevtsova, and T. Triller, The Journal of Chemical Physics **151**, 1 (2019).

[16] R. Haase, Zeitschrift für Physik **127**, 1 (1950).

[17] K. B. Haugen and A. Firoozabadi, The Journal of chemical physics **122** (2005), https://doi.org/10.1063/1.1829033.

[18] E. L. Dougherty and H. G. Drickamer, The Journal of Chemical Physics **23**, 295 (1955), https://doi.org/10.1063/1.1741957 .

[19] S. Chapman and T. G. Cowling, *The mathematical theory of non-uniform gases*, 2nd ed. (Cambridge university press, Bentley House, 200 Euston Road, London, N.W. 1, 1964).

[20] E. Tipton, R. Tompson, and S. Loyalka, European Journal of Mechanics - B/Fluids **28**, 353 (2009).

[21] R. Tompson, E. Tipton, and S. Loyalka, European Journal of Mechanics - B/Fluids **28**, 695 (2009).

[22] N. Vargaftik, *Tables on the Thermophysical Properties of Liquids and Gases*, 2nd ed. (Hemisphere Publishing Corporation, 1025 Vermont Ave, NW Washington DC 20005, 1975).

[23] SINTEF, "Thermopack," `https://github.com/SINTEF/thermopack/` (2020).

[24] J. Platten, M. Bou-Ali, P. Costeseque, J. Dutrieux, W. Kohler, C. Leppla, S. Wiegand, and G. Wittko, Philosophical Magazine (2010), 10.1080/01418610310001082O4.

[25] D. de Mezquia, M. Mounir Bou-Ali, J. Madariaga, and C. Santamaría, The Journal of Chemical Physics **140**, 1 (2014).

[26] D. Longree, J. C. Legros, and G. Thomas, The Journal of Physical Chemistry **84**, 3480 (1980).

[27] M. Bou-Ali, O. Ecenarro, J. Madariaga, C. Santamaría, and J. Valencia, Journal of Non-Equilibrium Thermodynamics **24**, 228 (1980).

[28] K. Zhang, M. Briggs, R. Gammon, and J. Sengers, Journal of Chemical Physics **104**, 6681 (1996).

[29] A. Königer, B. Meier, and W. Köhler, Philosophical Magazine **89**, 907 (2009).

[30] S. Wiegand, Journal of Physics: Condensed Matter **16**, R357–R379 (2004).

[31] M. Saghir, C. Jiang, and S. Derawi, The European Physical Journal E **15**, 241–247 (2004).

[32] I. Q. Matos, J. S. Varandas, and J. P. Santos, Brazilian Journal of Chemical Engineering **35**, 363 (2018).

[33] Merck, "Solvent miscibility table," (2020), https://www.sigmaaldrich.com/chemistry/solvents/solv
miscibility-table.html.

[34] A. Mialdun, V. Yasnou, S. Valentina, A. Königer, W. Köhler, D. Alonso de Mezquia, and M. Bou-Ali, The Journal of chemical physics **136**, 244512 (2012).

[35] J. Garrido, M. Cartes, and A. Mejia, The Journal of Supercritical Fluids **129**, 83 (2017).

[36] T. Lafitte, C. Avendaño, V. Papaioannou, A. Galindo, C. S. Adjiman, G. Jackson, and E. A. Müller, Molecular Physics **110**, 1189 (2012).

[37] S. A. Smith, J. T. Cripwell, and C. E. Schwarz, Journal of Chemical & Engineering Data **65**, 5778 (2020).

[38] J. Mick, M. Soroush Barhaghi, B. Jackman, K. Rushaidat, L. Schwiebert, and J. Potoff, Journal of Chemical Physics **143**, 14504 (2015).

[39] C. Miqueu, J. M. Míguez, M. M. Piñeiro, T. Lafitte, and B. Mendiboure, The Journal of Physical Chemistry B **115**, 9618 (2011).

[40] F. P. Nascimento, M. L. Paredes, A. P. D. Bernardes, and F. L. Pessoa, The Journal of Supercritical Fluids **154**, 104634 (2019).

[41] O. Talu and A. Myers, Colloids and Surfaces **188**, 83 (2001).

[42] J. T. Cripwell, S. A. M. Smith, C. E. Schwarz, and A. J. Burger, Industrial & Engineering Chemistry Research **57**, 9693 (2018), https://doi.org/10.1021/acs.iecr.8b01042 .

[43] D. Siderius, *NIST Standard Reference Simulation Website - SRD 173* (National Institute of Standards and Technology, 2017) accessed 12.12.2020.

A Symbols and abbreviations

Symbol	Description	Unit
c_i	Concentration of component i	$\mathrm{mol\,m^{-3}}$
D_T	Thermal diffusion coefficient	$\mathrm{m^2\,K^{-1}\,s^{-1}}$
D_i	Interdiffusion coefficient	$\mathrm{m^2\,s^{-1}}$
G	Gibbs energy	J
G_m	Molar Gibbs energy	$\mathrm{J\,mol^{-1}}$
H	Enthalpy	J
H_i	Molar enthalpy of component i	$\mathrm{J\,mol^{-1}}$
h	Enthalpy density	$\mathrm{J\,m^{-3}}$
h_i	Partial molar enthalpy of component i	$\mathrm{J\,mol^{-1}}$
$\mathbf{J}_i$	Diffusional flux of component i	$\mathrm{mol\,m^{-2}\,s^{-1}}$
$\mathbf{J}_q$	Diffusional heat flux	$\mathrm{J\,m^{-2}\,s^{-1}}$
$\mathbf{J}_q$	Diffusional, measurable heat flux	$\mathrm{J\,m^{-2}\,s^{-1}}$
$\mathbf{J}_s$	Diffusional entropy flux	$\mathrm{J\,m^{-2}\,s^{-1}\,K^{-1}}$
$L_{i,j}$	Phenomenological coefficients	–
n_i	Moles of component i	mol
n_T	Total number of moles in the system	mol
n_i	Moles of component i	mol
r_j	Reaction rate of reaction j	$\mathrm{mol\,s^{-1}\,m^{-3}}$
S	Entropy	$\mathrm{J\,K^{-1}}$
S_i	Molar entropy of component i	$\mathrm{J\,mol^{-1}\,K^{-1}}$
s	Entropy density	$\mathrm{J\,m^{-3}\,K^{-1}}$
s_T	Soret coefficient	$\mathrm{K^{-1}}$
T	Absolute temperature	K
U	Internal energy	J
u	Internal energy density	$\mathrm{J\,m^{-3}}$
$\mathbf{v}$	Fluid velocity	$\mathrm{m\,s^{-1}}$
V	Volume	$\mathrm{m^3}$
v_i	Partial molar volume of component i	$\mathrm{m^3\,mol^{-1}}$
x_i	Mole fraction of component i	–
X^0	Variable X in the ideal gas state	–
α_T	Thermal diffusion factor	–
μ_i	Chemical potential of component i	$\mathrm{J\,mol^{-1}}$
ν_i^j	Stochiometric coefficient of component i in reaction j	–
σ	Local entropy production	$\mathrm{J\,m^{-3}\,s^{-1}\,K^{-1}}$

Table A.1: Symbols used in the text, excluding Section 2.2.1.

Abbreviation	Description
EoS	Equation of State
SRK	Soave-Redlich-Kwong
PR	Peng-Robinson
PT	Patel-Teja
SW	Schmidt-Wensel
SPUNG	An extended corresponding state EoS
PC-SAFT	Perturbed Chain Statistical Associating Fluid Theory
CoV	Center of Volume
CoM	Center of Mass
K89	Kempers 1989 model
M-K89	Modified K89, see section 2.3
K01	Kempers 2001 model
KXX-CoM	KXX with CoM frame of reference
KXX-CoV	KXX with CoV frame of reference

Table A.2: Abbreviations used in the text.

B Deriving the entropy production

The system in question is has a velocity $\mathbf{v}$, diffusion of heat and mass and a net reaction taking place. Indicating the diffusive fluxes as $\mathbf{J_j}$, the local entropy production as σ, the stochiometric coefficients of the net reaction as ν_i and the net reaction rate as r, the balance equations for entropy, internal energy and mass for a stationary control volume become

$$\frac{\partial s}{\partial t} = -\nabla(\mathbf{v}s + \mathbf{J}_s) + \sigma \tag{B.1}$$

$$\frac{\partial u}{\partial t} = -\nabla(\mathbf{v}u + \mathbf{J}_q) - p\nabla\mathbf{v} \tag{B.2}$$

$$\frac{\partial c_i}{\partial t} = -\nabla(\mathbf{v}c_i + \mathbf{J}_i) + \nu_i r \tag{B.3}$$

The Gibbs equation on local form can be reordered to give

$$ds = \frac{1}{T}\left(du - \sum_i \mu_i dc_i\right) \tag{B.4}$$

Dividing by dt and inserting equations (B.2) and (B.3) yields

$$\frac{\partial s}{\partial t} = \frac{1}{T}\left[-\nabla(\mathbf{v}u + \mathbf{J}_q) - p\nabla\mathbf{v} - \sum_i \mu_i\left(-\nabla(\mathbf{v}c_i + \mathbf{J}_i) + \nu_i r\right)\right] \tag{B.5}$$

Recognising the molar Gibbs energy of reaction $\Delta_r G_m = \sum \nu_i \mu_i$, applying the relation

$$\phi \nabla \mathbf{J} = \nabla (\mathbf{J}\phi) - \mathbf{J} \nabla \phi \tag{B.6}$$

with $\phi = \left\{ \frac{1}{T}, \frac{\mu_i}{T} \right\}$ and $\mathbf{J} = \{\mathbf{J}_q, \mathbf{J}_i\}$ as well as expanding the terms $\nabla \mathbf{v} u$ and $\nabla \mathbf{v} c_i$ with the chain rule gives

$$\begin{aligned} \frac{\partial s}{\partial t} = &\left(\frac{-u - p + \sum \mu_i c_i}{T} \right) \nabla \mathbf{v} + \frac{\mathbf{v}}{T} \left(-\nabla u + \sum \mu_i \nabla c_i \right) \\ &- \nabla \left(\frac{\mathbf{J}_q - \sum \mu_i \mathbf{J}_i}{T} \right) + \mathbf{J}_q \nabla \left(\frac{1}{T} \right) - \sum \mathbf{J}_i \nabla \left(\frac{\mu_i}{T} \right) - r \frac{\Delta_r G}{T}. \end{aligned} \tag{B.7}$$

Recognizing the first term as $-s \nabla \mathbf{v}$, from the Gibbs equation and the second term as $-\mathbf{v} \nabla s$ from the Gibbs equation on local form, using the inverse chain rule and reordering gives

$$\frac{\partial s}{\partial t} = -\nabla \left(\mathbf{v} s + \frac{\mathbf{J}_q + \sum \mu_i \mathbf{J}_i}{T} \right) + \mathbf{J}_q \nabla \left(\frac{1}{T} \right) - \sum \mathbf{J}_i \nabla \left(\frac{\mu_i}{T} \right) - r \frac{\Delta_r G}{T}. \tag{B.8}$$

Comparing equations (B.1) and (B.8) yields the local entropy production

$$\sigma = \mathbf{J}_q \nabla \left(\frac{1}{T} \right) - \sum_i \mathbf{J}_i \nabla \left(\frac{\mu_i}{T} \right) - r \frac{\Delta_r G_m}{T} \tag{B.9}$$

To rewrite this in terms of measurable variables the factor $\nabla \left(\frac{\mu_i}{T} \right)$ can be expanded as

$$\begin{aligned} \nabla \left(\frac{\mu_i}{T} \right) &= \mu_i \nabla \left(\frac{1}{T} \right) + \frac{1}{T} \Bigg(\nabla \mu_{i,T} + \left(\frac{\partial \mu_i}{\partial H_i} \right)_{S_i,T} \left(\frac{\partial H_i}{\partial T} \right) \nabla T \\ &\qquad + \left(\frac{\partial \mu_i}{\partial S_i} \right)_{H_i,T} \left(\frac{\partial S_i}{\partial T} \right) \nabla T + \left(\frac{\partial \mu_i}{\partial T} \right)_{H_i,S_i} \nabla T \Bigg) \\ &= (H_i - T S_i) \nabla \left(\frac{1}{T} \right) + \frac{1}{T} \left(\nabla \mu_{i,T} + C_{p,i} \nabla T - T \left(\frac{C_{p,i}}{T} \right) \nabla T - S_i \nabla T \right) \\ &= (H_i - T S_i) \nabla \left(\frac{1}{T} \right) + \frac{1}{T} \nabla \mu_{i,T} - \frac{1}{T^2} (T S_i) \nabla T \\ &= H_i \nabla \left(\frac{1}{T} \right) + \frac{1}{T} \nabla \mu_{i,T} \end{aligned} \tag{B.10}$$

Where the second equality follows from the definition of heat capacity at constant pressure $C_{p,i} = \frac{\partial H_i}{\partial T}$ and the second law of thermodynamics. The fourth equality follows from applying the chain rule to $\nabla \left(\frac{1}{T} \right)$. Further, defining $\mathbf{J}_q = \mathbf{J}'_q + \sum H_i \mathbf{J}_i$ where $\mathbf{J}'_q$ is the

measurable, diffusional heat flux and H_i is the molar enthalpy of component i gives the local entropy production expressed in terms of measurable variables

$$\sigma = \mathbf{J}_q' \nabla \left(\frac{1}{T} \right) + \sum_i \mathbf{J}_i \left(-\frac{1}{T} \nabla \mu_{i,T} \right) + r \left(-\frac{\Delta_r G_m}{T} \right). \tag{B.11}$$

C Experimental data

C.1 Experimental Soret Coefficients

ω_{EtOH} [-] \ T [°C]	10	15	20	25	30	35	40	50	60
0.051	9.04	8.59	8.18	7.71	7.95	7.3	6.8	5.71	4.38
0.1	7.63	7.46	6.99	6.75	6.37	6.0	5.6		
0.152	6.3	5.72	5.37	5.01	4.75	4.46	4.13		
0.1983	3.82	3.75	3.66	3.41	3.22	2.98	2.87		
0.2512	1.54	1.61	1.56	1.47	1.37	1.24	1.12		
0.3016	-0.58	-0.37	-0.3	-0.26	-0.23	-0.254			
0.3538	-2.44	-2.35	-2.15	-1.51	-1.15	-0.87			
0.3987	-2.97	-3.95	-3.77	-3.55	-3.26	-2.67	-3.07		
0.4998	-6.78	-6.05	-5.53	-5.18	-4.8	-4.49	-4.23	-3.87	-3.14
0.5921	-7.21	-6.34	-5.97	-5.51	-4.99	-4.55	-4.14	-3.54	
0.8968	-2.8	-3.81	-5.14	-6.64	-1.92	-1.43			
0.9437	-2.21	-6.37	-5.28	-1.21	-0.102	-20.0			

Table C.1: Soret coefficient (mK^{-1}) of Ethanol in water[29]

$x_{Toluene}[-]$ \ T [°C]	5	15	25	35	45
0.05	3.33	3.23	3.17	3.12	3.1
0.25	4.38	4.19	3.98	3.85	3.74
0.5	5.63	5.26	4.92	4.64	4.41
0.75	6.63	6.1	5.56	5.2	4.86
0.95	7.2	6.54	5.96	5.53	5.15

Table C.2: Soret coefficient (mK^{-1}) of n-hexane in toluene[25–28]

Species \ x [-]	0.1	0.2	0.3	0.5	0.75	0.8	0.85	0.9
benzene[1]	3.09		3.45	4.96	6.91	7.35		8.28
benzene[2]	3.12		3.62	4.86	6.41			7.82
carbon tet.[1]	12.8		13.2	12.4	12.3			14.8
carbon tet.[2]	12.7	13.0	13.2	12.9	13.5		14.2	14.6

Table C.3: Soret coefficients (mK^{-1}) of several species at 298 K. The other component in the mixture is: [1]n-heptane, [2]n-hexane. Soret coefficient refers to the lighter component in the mixture.[27]

Species \ x [-]	0.1	0.31	0.4	0.5	0.7	0.9
nC12, nC8	2.55	2.32	2.28	2.23	2.11	
nC12, nC7	3.03	2.88	2.99	2.89	2.61	2.45
nC12, nC6	3.69	3.58	3.6	3.46	3.36	3.21
nC10, nC5	3.17	3.08	3.09	2.96	3.05	3.15

Table C.4: Soret coefficient (mK^{-1}) of n-alkane mixtures, mole fraction refers to the first component, Soret coefficient refers to the second component.[25]

$\omega_{H_2O}[-]$	S_T [mK^{-1}]	$\omega_{H_2O}[-]$	S_T [mK^{-1}]
0.2	3.09	0.6	5.25
0.25	3.96	0.65	4.02
0.3	4.60	0.7	2.10
0.35	5.08	0.75	-0.36
0.4	5.47	0.8	-3.10
0.45	5.81	0.85	-5.89
0.5	5.99	0.9	-8.47
0.55	5.85	0.95	-10.60

Table C.5: Soret coefficient of isopropanol in water at 298 K[34]

C.2 Mie-parameters and Standard state Enthalpies

x_{Ar}	s_T [mK^{-1}]
0.1248	1.39
0.2025	0.89
0.2765	1.13
0.3771	3.33
0.4923	2.13

Table C.6: Soret coefficient of methane in argon at 1 atm, 298 K.[26]

Component	σ [Å]	ϵ [Kk_B^{-1}]	Source
NC5	4.247	317.5	[35]
NC6	4.508	376.35	[35]
NC7	4.766	436.13	[35]
NC8	5.008	490.12	[35]
NC10	4.603	407.09	[36]
NC12	4.743	438.2	[37]
C1	3.74	161.0	[38]
C3	3.7943	221.96	[37]
BENZENE	5.292	658.17	[36]
XE	3.964	243.8	[38]
AR	3.405	122.1	[38]
NE	2.793	32.3	[38]
ETOH	3.4379	206.62	[37]
H2O	3.0856	177.6851	[39]
TOLU	4.2277	409.73	[40]
CO2	3.0465	235.73	[40]
HE	2.64	10.9	[41]
PROP1OL	3.5612	227.66	[42]

Table C.7: Mie-potential parameters used to approximate hard-sphere radii.

Component	H_m [kJ mol^{-1}]	Component	H_m [kJ mol^{-1}]
TOLU	50.1	NC5	-146.8
NC6	-167.1	NC12	-290.9
NC7	-187.8	NC8	-208.7
BENZENE	82.9	CYCLOHEX	-123.1
C1	-74.8	NC4	-125.6
ETOH	-278	C3	-104.7
AR	0	R11	-100
H2O	-285.83	PROP1OL	-256
NC10	-249.7		

Table C.8: Ideal gas Standard state enthalpies from the NIST Webbook.[43]

D Implementation

D.1 Kempers' models

The implementation Kempers' models was done by constructing a parent class `Kempers`, and two children `Kempers89` and `Kempers01`. The parent holds general initialization work that is done for both models, as well as some generalized functions that facilitate quick retrieval of Soret-coefficients over a range of temperatures, compositions or pressures.

```
'''
Author: Vegard G. Jervell
Date: December 2020
Purpose: Parent class containing some general procedures to be used in both Kempers89 and
    Kempers01
Requires: numpy, ThermoPack
Note: This is a virtual class, and will not do anything exciting if initialized on its own.
'''

import numpy as np
from pycThermopack.pyctp import cubic
from models.kineticgas import KineticGas
from scipy.constants import gas_constant
import warnings

class Kempers(object):
    def __init__(self, comps, eos, x=[0.5, 0.5], temp=300, pres=1e5, phase=1):
        '''
        KempersXX parent class, contains interface for retrieving soret-coefficient for spans of
    temperatures, pressures or compositions
        and some general initialization procedures that are common for the two KempersXX' models
        :param comps (str): comma separated list of components
        :param x (1darray): list of mole fractions
        :param eos (ThermoPack): Initialized Equation of State object, initialized with components '
    comp'
        :param temp (float > 0): Temperature [K]
        :param pres (float > 0): Pressure [Pa]
        :param phase: Phase of mixture, used for calculating dmudn_TP, see thermo.thermopack for
    phase identifiers
        '''

        self.comps = comps
        self.x = np.array(x)
        self.temp = temp
        self.pres = pres
        self.phase = phase
        self.total_moles = 1 #Dummy value, because some ThermoPack methods require it. Does not
     effect output.

        eoscomp_inds = [eos.getcompindex(comp) for comp in self.comps.split(',')]
        if -1 in eoscomp_inds:
            warnings.warn('Equation of state and KempersXX must be initialized with same
    components.\n'
                          "I'm initializing using SRK with "+comps+" now to avoid crashing")
            self.eos = cubic.cubic()
```

```
40              self.eos.init(self.comps, 'SRK')
41
42          elif any(np.array(eoscomp_inds) != sorted(np.array(eoscomp_inds))):
43              eoscomps = ','.join(eos.get_comp_name(i) for i in sorted(eoscomp_inds))
44              warnings.warn('Equation of state and KempersXX must be initialized with same
    components in the same order\n'
45                            'but are initialized with ' + eoscomps + ' and ' + self.comps+'.\n'
46                            "I'm initializing using SRK with "+comps+" now to avoid crashing")
47              self.eos = cubic.cubic()
48              self.eos.init(self.comps, 'SRK')
49
50          else:
51              self.eos = eos
52
53      #Some standard set-methods (use these! they handle some important stuff!)
54      def set_min_temp(self, temp):
55          self.min_temp = temp
56          self.eos.set_tmin(temp)
57
58      def set_temp(self,temp):
59          self.temp = temp
60
61      def set_pres(self, pres):
62          self.pres = pres
63
64      def set_mole_fracs(self, x):
65          self.x = np.array(x)
66          self.reset_alpha_t0()
67
68      def set_eos(self, eos):
69          '''
70          Change equation of state
71          :param eos_key (str): new equation of state key
72          '''
73          eoscomp_inds = [eos.getcompindex(comp) for comp in self.comps.split(',')]
74          if -1 in eoscomp_inds:
75              warnings.warn('Equation of state and KempersXX must have the same components.'
76                            "I'm just not going to change anything!")
77              return 1
78
79          elif any(np.array(eoscomp_inds) != sorted(np.array(eoscomp_inds))):
80              eoscomps = ','.join(eos.get_comp_name(i) for i in sorted(eoscomp_inds))
81              warnings.warn('Equation of state and KempersXX must have the same components in the
    same order'
82                            'but are given ' + eoscomps + ' and ' + self.comps + '.'
83                            "I'm not changing anything!")
84              return 1
85          else:
86              self.eos = eos
87              return 0
88
89      def set_comps(self, comps, eos):
90          '''
91          Change components
92          :param comps: Comma separated list of components
93          :param eos: Initialized equation of state, with same components as 'comp'
```

```
94          '''
95          #Use the check in self.set_eos() to determine if input is valid, only change if it is
96          old_comps = self.comps
97          self.comps = comps
98          if self.set_eos(eos) == 0:
99              pass
100         else:
101             self.comps = old_comps
102
103     def reset_alpha_t0(self):
104         #Overridden in Kempers01
105         pass
106
107     def get_soret_cov(self, kin=False):
108         # Get soret coefficient at current settings, center of volume frame of reference
109         # Overridden in Kempers01 and Kempers89
110         pass
111
112     def get_soret_com(self, kin=False):
113         # Get soret coefficient at current settings, center of mass frame of reference
114         # Overridden in Kempers01 and Kempers89
115         pass
116
117     def get_soret_avg(self, kin=False):
118         # Get average soret coefficient computed with 'cov' and 'com' frame of reference
119         return 0.5 * (self.get_soret_cov(kin=kin) + self.get_soret_com(kin=kin))
120
121     def get_soret_comp(self, x, mode='cov', kin=False):
122         '''
123         Get soret-coefficients for a range of compositions
124         Args:
125             x : array-like
126                 mole fractions of components, if N - 1 components are given, N is calculated
    implicitly
127                 row i is composition i, column j is mole fraction of component j
128                 [[x1, x2, ..., xN], #composition 1
129                  [x1, x2, ..., xN]] #composition 2
130
131         return:
132             tuple of floats or ndarrays, matching shape of input : soret coefficients at given
    composition(s)
133         '''
134         if mode == 'cov':
135             get_soret = self.get_soret_cov
136         elif mode == 'com':
137             get_soret = self.get_soret_com
138         elif mode == 'avg':
139             get_soret = self.get_soret_avg
140         else:
141             warnings.warn("mode must be either 'cov', 'com' or 'avg', not "+str(mode)+" defaulting
    back to 'cov'")
142             get_soret = self.get_soret_cov
143
144         old_mole_fracs = [frac for frac in self.x]  # take care of the values from initialization
145
146         x = np.array(x)
```

```
147
148         if len(x.shape) == 1:
149             if x.shape[0] != len(self.x):
150                 xN = 1 - x
151                 x = np.concatenate((np.vstack(x), np.vstack(xN)), axis=1)
152             elif x.shape[0] == len(self.x):
153                 x = [x]
154
155         elif x.shape[1] == len(self.x) - 1:
156             xN = 1 - np.sum(x, axis=1)
157             x = np.concatenate((x, np.vstack(xN)), axis=1)
158         elif x.shape[1] == len(self.x):
159             pass
160         else:
161             raise IndexError('x must contain N-1 or N mole fractions')
162
163         if kin is True:
164             soret = np.empty(x.shape, float)
165             kin_contrib = np.empty(x.shape, float)
166             for i, fracs in enumerate(x):
167                 self.x = fracs
168                 self.reset_alpha_t0()
169                 soret[i], kin_contrib[i] = get_soret(kin=kin)
170
171             self.x = np.array(old_mole_fracs)  # reset values from initialization
172
173             return soret.transpose(), kin_contrib.transpose()
174         else:
175             soret = np.empty(x.shape, float)
176             for i, fracs in enumerate(x):
177                 self.x = fracs
178                 self.reset_alpha_t0()
179                 soret[i] = get_soret()
180
181             self.x = np.array(old_mole_fracs)  # reset values from initialization
182
183             return soret.transpose()
184
185     def get_soret_temp(self, temps, mode='cov', kin=False):
186         '''
187         Get soret coefficients for a range of temperatures
188         Args:
189             temps (int, float or array-like) : temperature(s) [K] to get soret-coefficients for
190
191         return:
192             tuple of floats or arrays, matching shape of input : Soret-coefficients at given
    temperature(s)
193         '''
194         if mode == 'cov':
195             get_soret = self.get_soret_cov
196         elif mode == 'com':
197             get_soret = self.get_soret_com
198         elif mode == 'avg':
199             get_soret = self.get_soret_avg
200         else:
```

```
201             warnings.warn("mode must be either 'cov', 'com' or 'avg', not "+str(mode)+" defaulting
    back to 'cov'")
202             get_soret = self.get_soret_cov
203
204         old_temp = self.temp
205
206         if type(temps) in (list, np.ndarray):
207             if kin is True:
208                 soret = np.empty((len(temps), len(self.x)))
209                 kin_contrib = np.empty((len(temps), len(self.x)))
210                 for i, T in enumerate(temps):
211                     self.temp = T
212                     soret[i], kin_contrib[i] = get_soret(kin=kin)
213             else:
214                 soret = np.empty((len(temps), len(self.x)))
215                 for i, T in enumerate(temps):
216                     self.temp = T
217                     soret[i] = get_soret()
218
219         else:
220             self.temp = temps
221             if kin is True:
222                 soret, kin_contrib = get_soret(kin=kin)
223             else:
224                 soret = get_soret()
225
226         self.temp = old_temp
227
228         if kin is True:
229             return soret.transpose(), kin_contrib.transpose()
230
231         else:
232             return soret.transpose()
233
234     def get_soret_pres(self, pressures, mode='cov'):
235         '''
236         Get soret coefficients for a range of pressures
237             Args:
238                 pressures (float or array-like) : pressure(s) [Pa] to get soret-coefficients for
239
240             return:
241                 tuple of floats or arrays, matching shape of input : Soret-coefficients at given
    pressure(s)
242         '''
243
244         if mode == 'cov':
245             get_soret = self.get_soret_cov
246         elif mode == 'com':
247             get_soret = self.get_soret_com
248         elif mode == 'avg':
249             get_soret = self.get_soret_avg
250         else:
251             warnings.warn("mode must be either 'cov', 'com' or 'avg', not "+str(mode)+" defaulting
    back to 'cov'")
252             get_soret = self.get_soret_cov
253
```

```
254         old_pres = self.pres #take care of initialization -value
255
256         if type(pressures) in (list, np.ndarray):
257
258             soret = np.empty((len(pressures), len(self.x)))
259             for i, pres in enumerate(pressures):
260                 self.pres = pres
261                 soret[i] = get_soret()
262
263         else:
264             self.pres = pressures
265             soret = get_soret()
266
267         self.pres = old_pres #reset to initialization -value
268
269         return soret
270
271     def dmudn_TP(self):
272         '''
273         Calculate chemical potential derivative with respect to number of moles at constant
    temperature and pressure
274         :return: ndarray, dmudn[i,j] = dmu_idn_j
275         '''
276
277         v, dvdn = self.eos.specific_volume(self.temp, self.pres, self.x, self.phase, dvdn=True)
278         mu, dmudn_TV = self.eos.chemical_potential_tv(self.temp, v * self.total_moles,
279                                                       self.x * self.total_moles, dmudn=True)
280         pres, dpdn = self.eos.pressure_tv(self.temp, v * self.total_moles, self.x * self.total_moles,
     dpdn=True)
281
282         return dmudn_TV - np.tensordot(dpdn, dvdn, axes=0)
283
284     def dmudx_TP(self):
285         '''
286         Calculate chemical potential derivative with respect to mole fraction of components
287         at constant temperature and pressure
288         :return: ndarray, dmudx[i,j] = dmu_idn_j
289         '''
290         dmudn = self.dmudn_TP()
291
292         M1 = (np.tensordot(np.ones(len(self.x)), -1 / self.x, axes=0) +
293               (np.identity(len(self.x)) * (1 / self.x + 1 / (1 - self.x)))) * self.total_moles
294
295         return np.dot(dmudn, M1)
```

```
1 '''
2 Author: Vegard G. Jervell
3 Date: December 2020
4 Purpose: Implementation of L. J. T. M. Kempers' 1989 model for prediction of Soret-coefficients in
      multicomponent
5         liquid mixtures, derived in "A thermodynamic theory of the Soret effect in a multicomponent
      liquid",
6         Journal of Chemical Physics, 1989.
7         doi : http://dx.doi.org/10.1063/1.456321
8 Requires: numpy, scipy, pandas, ThermoPack
9 Note: pandas is used to get ideal-gas state enthalpies from the file 'ideal_gas_enthalpies.xlsx'. It
```

```
    should
    be possible to aquire these from ThermoPack, thereby removing the need for a separate data-file of
    standard-state enthalpies.
'''

from pycThermopack.pyctp import cubic, cpa, lee_kesler
from models.kempers import Kempers
import numpy as np
from scipy.optimize import root
import pandas as pd
import warnings
import os

class Kempers89(Kempers):
    def __init__(self, comps, eos, x=[0.5, 0.5], temp=300, pres=1e5, phase=1):
        '''
        :param comps (str): Comma separated list of components, as per ThermoPack convention
        :param x (array-like): list of component mole fractions
        :param eos (ThermoPack): Initialized equation of state, with same components as Kempers
        :param temp (float): Temperature [K]
        :param pres (float): Pressure [Pa]
        :param phase (int): Phase identifier to be used by ThermoPack
        '''
        super().__init__(comps, eos, x=x, temp=temp, pres=pres, phase=phase)

        if self.eos.guess_phase(self.temp, self.pres, self.x) != 1:
            warnings.warn('This model may work poorly for phases other than liquid,'
                          ' ThermoPack predicted phase is '+
                          eos.get_phase_type(eos.guess_phase(self.temp, self.pres, self.x)))

    def get_standard_enthalpies(self):
        '''
        Get ideal-gas standard state enthalpies from 'ideal_gas_enthalpies.xlsx'
        :return: 1darray: standard state enthalpies of components
        '''
        file_path = os.path.dirname(__file__)
        data = pd.read_excel(file_path+'/ideal_gas_enthalpies.xlsx')
        return np.array([data.loc[data['Comp'] == comp].values.flatten()[1] for comp in self.comps.
    split(',')])

    def get_soret_cov(self, kin=False):
        '''
        Get soret coefficients at current settings
        :return: 1darray: soret coefficients of components
        '''

        if kin is True:
            raise ValueError('Kempers89 has no kinetic contribution!')

        self.eos.set_tmin(1)
        V, v = self.eos.specific_volume(self.temp, self.pres, self.x, self.phase, dvdn=True)
        H, h = self.eos.enthalpy(self.temp, self.pres, self.x, self.phase, dhdn=True)
        H0, h0 = self.eos.enthalpy(self.temp, 1e-5, self.x, 2, dhdn=True)
        H_molar_ideal = self.get_standard_enthalpies()

```

```
        h = h - h0 + H_molar_ideal
        dmudx = self.dmudx_TP()

        N = len(self.x)

        def eq_set(alpha):
            dxdT = - (alpha * self.x * (1 - self.x))
            eqs = np.zeros(N)
            for i in range(0, N - 1):
                eqs[i] = -((h[i] / v[i]) - (h[N - 1] / v[N - 1])) \
                         + sum([((dmudx[i, j] / v[i]) - (dmudx[N - 1, j] / v[N - 1])) * dxdT[j] for j
    in range(0, N - 1)])

            eqs[N - 1] = sum(alpha)

            return eqs

        alpha = root(eq_set, [1 for i in range(N)]).x
        soret = alpha / self.temp

        return soret

    def get_soret_com(self, kin=False):
        '''
        Get soret coefficients at current settings
        :return: 1darray: soret coefficients of components
        '''

        if kin is True:
            raise ValueError('Kempers89 has no kinetic contribution!')

        M = self.mole_weights = np.array([self.eos.compmoleweight(self.eos.getcompindex(comp))
                                          for comp in self.comps.split(',')])
        _, h = self.eos.enthalpy(self.temp, self.pres, self.x, self.phase, dhdn=True)
        _, h0 = self.eos.enthalpy(self.temp, 1e-5, self.x, 2, dhdn=True)
        H_molar_ideal = self.get_standard_enthalpies()

        h = h - h0 + H_molar_ideal
        dmudx = self.dmudx_TP()

        N = len(self.x)

        def eq_set(alpha):
            dxdT = - (alpha * self.x * (1 - self.x))
            eqs = np.zeros(N)
            for i in range(0, N - 1):
                eqs[i] = -((h[i] / M[i]) - (h[N - 1] / M[N - 1])) \
                         + sum([((dmudx[i, j] / M[i]) - (dmudx[N - 1, j] / M[N - 1])) * dxdT[j] for
    j in range(0, N - 1)])

            eqs[N - 1] = sum(alpha)

            return eqs

        alpha = root(eq_set, [1 for i in range(N)]).x
        soret = alpha / self.temp
```

```
117
118         return soret
```

```
1  '''
2  Author: Vegard G. Jervell
3  Date: December 2020
4  Purpose: Implementation of L. J. T. M. Kempers' 2001 model for prediction of Soret coefficients in multicomponent
5          mixtures, derived in "A comprehensive thermodynamic theory of the Soret effect in a multicomponent gas,
6          liquid, or solid", 2001, Journal of Chemical Physics
7          doi : http://dx.doi.org/10.1063/1.1398315
8  Requires : numpy, scipy, ThermoPack, KineticGas
9  Note : KineticGas is only 2-component compatible, replacing the KineticGas module with a module capable of predicting
10         thermal diffusion coefficients for multicomponent ideal-gas mixtures will make this module multicomponent-compatible.
11 '''
12
13 import numpy as np
14 from scipy.constants import gas_constant
15 from pycThermopack.pyctp import cubic
16 from pycThermopack.pyctp import cpa
17 from models.kempers import Kempers
18 from scipy.optimize import root
19 from models.alpha_t0_empirical import Alpha_T0_empirical
20 from models.kineticgas import KineticGas
21
22 class Kempers01(Kempers):
23     def __init__(self, comps, eos, x=[0.5, 0.5], temp=300, pres=1e5, phase=1,
24                  alpha_t0_empirical=False, alpha_t0_method='wheights', alpha_t0_N = 7):
25         '''
26         :param comps (str): comma separated list of components
27                 :param eos (ThermoPack): Initialized equation of state, with same components as Kempers
28         :param x (1darray): list of mole fractions
29         :param temp (float > 0): Temperature [K]
30         :param pres (float > 0): Pressure [Pa]
31         :param phase: Phase of mixture, used for calculating dmudn_TP, see thermo.thermopack for phase identifiers
32         :param alpha_t0_empirical (bool): True if using empirical alpha_T0 values
33         :param alpha_t0_method (str): what method to use for empirical alpha_T0 values (see Alpha_T0_empirical)
34         :param alpha_t0_N (int > 0): Order of approximation when using analytical alpha_T0 values
35         '''
36         super().__init__(comps, eos, x=x, temp=temp, pres=pres, phase=phase)
37
38         self.alpha_T0_empirical = alpha_t0_empirical
39         if alpha_t0_empirical:
40             self.alpha_T0_method = alpha_t0_method
41             self.kinetic_gas = Alpha_T0_empirical(comps, method=alpha_t0_method)
42         else:
43             self.alpha_T0_N = alpha_t0_N
44             self.kinetic_gas = KineticGas(comps, self.eos, mole_fracs=x, N=alpha_t0_N)
45
46     def reset_alpha_t0(self):
```

```
47          #reset alpha_t0_getter with current mole fractions
48          if self.alpha_T0_empirical:
49              self.kinetic_gas = Alpha_T0_empirical(self.comps, mole_fracs=self.x, method=self.
    alpha_T0_method)
50          else:
51              self.kinetic_gas = KineticGas(self.comps, self.eos, mole_fracs=self.x, N=self.
    alpha_T0_N)
52
53      def get_soret_cov(self, kin=False):
54          '''
55          Get soret coefficients at current settings, center of volume frame of reference
56          :return: (ndarray) soret coefficients
57          '''
58
59          R = gas_constant
60          v, dvdn = self.eos.specific_volume(self.temp, self.pres, self.x, self.phase, dvdn=True)
61          h, dhdn = self.eos.enthalpy(self.temp, self.pres, self.x, self.phase, dhdn=True)
62          h0, dh0dn = self.eos.enthalpy(self.temp, 1e-5, self.x, 2, dhdn=True)
63
64          dmudx = self.dmudx_TP()
65          alpha_T0 = self.kinetic_gas.alpha_T0(self.temp)
66
67          #using alpha_T0 as initial guess for root solver
68          initial_guess = alpha_T0 * self.temp
69
70          N = len(self.x)
71          #Defining the set of equations
72          def eq_set(alpha):
73              eqs = np.zeros(N)
74              for i in range(N-1):
75                  eqs[i] = ((dhdn[-1] - dh0dn[-1])/dvdn[-1]) - ((dhdn[i] - dh0dn[i])/dvdn[i])\
76                           + R * self.temp * ((alpha_T0[i] * (1 - self.x[i]) / dvdn[i])
77                                              - (alpha_T0[-1] * (1 - self.x[-1])/dvdn[-1]))\
78                           - sum((dmudx[i, j]/dvdn[i] - dmudx[-1, j]/dvdn[-1]) * self.x[j] * (1 - self.
    x[j]) * alpha[j]
79                                 for j in range(N - 1))
80
81              eqs[N-1] = sum(self.x * (1 - self.x) * alpha)
82              return eqs
83
84          #Solve the set of equations, warn if non-convergent
85          solved = root(eq_set, initial_guess )
86          if solved.success is False:
87              print('Solution did not converge for composition :', self.x, ', Temperature :', self.
    temp)
88          alpha = solved.x
89
90          soret = alpha / self.temp
91
92          if kin is True:
93              kin_contrib = alpha_T0 * R /(self.x * dmudx.diagonal())
94              return soret, kin_contrib
95          else:
96              return soret
97
98      def get_soret_com(self, kin=False):
```

```
        '''
        Get soret coefficients at current settings, center of mass frame of reference
        :return: (ndarray) soret coefficients
        '''

        R = gas_constant
        M = self.mole_weights = np.array([self.eos.compmoleweight(self.eos.getcompindex(comp))
                                          for comp in self.comps.split(',')])
        h, dhdn = self.eos.enthalpy(self.temp, self.pres, self.x, self.phase, dhdn=True)
        h0, dh0dn = self.eos.enthalpy(self.temp, 1e-5, self.x, 2, dhdn=True)

        dmudx = self.dmudx_TP()
        alpha_T0 = self.kinetic_gas.alpha_T0(self.temp)

        # using alpha_T0 as initial guess for root solver
        initial_guess = alpha_T0 * self.temp

        N = len(self.x)

        # Defining the set of equations
        def eq_set(alpha):
            eqs = np.zeros(N)
            for i in range(N - 1):
                eqs[i] = ((dhdn[-1] - dh0dn[-1]) / M[-1]) - ((dhdn[i] - dh0dn[i]) / M[i]) \
                         + R * self.temp * ((alpha_T0[i] * (1 - self.x[i]) / M[i])
                                            - (alpha_T0[-1] * (1 - self.x[-1]) / M[-1])) \
                         - sum(
                    (dmudx[i, j] / M[i] - dmudx[-1, j] / M[-1]) * self.x[j] * (1 - self.x[j]) *
alpha[j]
                    for j in range(N - 1))

            eqs[N - 1] = sum(self.x * (1 - self.x) * alpha)
            return eqs

        # Solve the set of equations, warn if non-convergent
        solved = root(eq_set, initial_guess)
        if solved.success is False:
            print('Solution did not converge for composition :', self.x, ', Temperature :', self.
temp)
        alpha = solved.x

        soret = alpha / self.temp

        if kin is True:
            kin_contrib = alpha_T0 * R /(self.x * dmudx.diagonal())
            return soret, kin_contrib
        else:
            return soret
```

D.2 Prediction of the thermal diffusion factor

The methods of calculate or approximate α_T^0 are implemented as the classes Alpha_T0_Empirical and KineticGas. The latter also includes methods for calculating the interdiffusion coefficient and the thermal diffusion coefficient in the ideal gas state.

```
'''
Author: Vegard G. Jervell
Date: December 2020
Purpose: Implementation of the Chapman-Enskog solutions to the Boltzmann equations for a binary system
        as proposed by Tompson, Tipton and Loyalka, in the paper
        "Chapman–Enskog solutions to arbitrary order in Sonine polynomials III:
        Diffusion, thermal diffusion, and thermal conductivity in a binary, rigid-sphere, gas mixture"
        doi : https://doi.org/10.1016/j.euromechflu.2008.12.002

Requires : numpy, scipy, matplotlib, pandas
'''

import numpy as np
import scipy.linalg as lin
import scipy.constants as const
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.gridspec as gs
import pandas as pd
import os

#fac = np.math.factorial

def fac(n):
    if n in (0,1):
        return 1
    else:
        val = 1
        for i in range(2,n+1):
            val *= i
        return val

def summation(start, stop, func, args=None):
    if args is not None:
        return sum(func(i, args) for i in range(start, stop + 1))
    else:
        return sum(func(i) for i in range(start, stop + 1))

def delta(i, j):
    if i == j:
        return 1
    else:
        return 0

def w(l, r):
    return 0.25 * (2 - ((1 / (l + 1)) * (1 + (-1) ** l))) * np.math.factorial(r + 1)

class KineticGas():

    def __init__(self, comps, eos, mole_fracs=[0.5,0.5], N=1):
        '''
        :param comps (str): Comma-separated list of components, following Thermopack-convention
        :param eos (thermopack): An initialized equation of state, only used to get mole weights
        :param mole_fracs (array-like): list of mole fractions
```

```
        :param N (int > 0): Order of approximation.
                            Be aware that N > 10 can be detrimental to runtime. This should be implemented in C++ or Fortran.
        '''

        #Packing out variables in an n-component compatible way
        complist = comps.split(',')
        self.mole_weights = np.array([eos.compmoleweight(eos.getcompindex(comp)) for comp in complist])

        self.m0 = np.sum(self.mole_weights)
        self.M = self.mole_weights/self.m0

        self.sigmaij = self.get_hard_sphere_radius(comps)
        self.sigma = np.diag(self.sigmaij)

        #This part is only binary-compatible
        self.M1, self.M2 = self.M
        self.x1, self.x2 = mole_fracs
        self.m1, self.m2 = self.mole_weights
        self.sigma1, self.sigma2 = self.sigma
        self.sigma12 = self.sigmaij[0, 1]

        #Calculate the (temperature independent) soret-coefficient at the ideal gas state
        self.T = 100 #Set T to a dummy value, because intermediate expressions depend on T, even though final result does not.
        self.N = N

        pq_range = np.arange(-N, N + 1, 1)
        self.A_matr = np.empty((2 * N + 1, 2 * N + 1), float)
        for i, p in enumerate(pq_range):
            for j, q in zip([j for j in range(i, len(pq_range))], pq_range[i:]):
                self.A_matr[i, j] = self.a(int(p), int(q)) # a(p,q) gives NaN or inf if type(p) == np.int64 or type(q) == np.int64... dont ask why
                self.A_matr[j, i] = self.A_matr[i, j]

        delta_0 = (3 / 2) * np.sqrt(const.Boltzmann * self.T / np.sum(self.mole_weights))
        b = np.zeros(2 * N + 1)
        b[int((len(b) - 1) / 2)] = delta_0
        d = lin.solve(self.A_matr, b)

        d_1, d0, d1 = d[int(((len(d) + 1) / 2) - 2): int(((len(d) + 1) / 2) + 1)]
        self.soret = -(5 / (2 * d0)) * ((self.x1 * d1 / np.sqrt(self.M1)) + (self.x2 * d_1 / np.sqrt(self.M2)))

        self.d_1, self.d0, self.d1 = d_1, d0, d1

    def interdiffusion(self, T):
        delta_0 = (3 / 2) * np.sqrt(const.Boltzmann * self.T / np.sum(self.mole_weights))
        b = np.zeros(len(self.A_matr))
        b[int((len(b) - 1) / 2)] = delta_0
        d = lin.solve(self.A_matr, b)
        d_1, d0, d1 = d[int(((len(d) + 1) / 2) - 2): int(((len(d) + 1) / 2) + 1)]

        return 0.5 * self.x1 * self.x2 * np.sqrt(2 * const.Boltzmann * T / self.m0) * d0

```

```
106     def thermal_diffusion(self,T):
107         delta_0 = (3 / 2) * np.sqrt(const.Boltzmann * self.T / np.sum(self.mole_weights))
108         b = np.zeros(len(self.A_matr))
109         b[int((len(b) - 1) / 2)] = delta_0
110         d = lin.solve(self.A_matr, b)
111         d_1, d0, d1 = d[int(((len(d) + 1) / 2) - 2): int(((len(d) + 1) / 2) + 1)]
112 
113         return -(5/4) * self.x1 * self.x2 * np.sqrt(2 * const.Boltzmann * T / self.m0) * \
114                 ((self.x1 * d1 / np.sqrt(self.M1)) + (self.x2 * d_1/np.sqrt(self.M2)))
115 
116     def alpha_T0(self, T):
117         delta_0 = (3 / 2) * np.sqrt(const.Boltzmann * self.T / np.sum(self.mole_weights))
118         b = np.zeros(len(self.A_matr))
119         b[int((len(b) - 1) / 2)] = delta_0
120         d = lin.solve(self.A_matr, b)
121         d_1, d0, d1 = d[int(((len(d) + 1) / 2) - 2): int(((len(d) + 1) / 2) + 1)]
122 
123         alpha_T0 = -(5 / (2 * d0)) * ((self.x1 * d1 / np.sqrt(self.M1)) + (self.x2 * d_1 / np.sqrt(
    self.M2)))
124 
125         return np.array([alpha_T0, -alpha_T0])
126 
127     def get_hard_sphere_radius(self, comps):
128         '''
129         Get hard-sphere diameters, assumed to be equal to Mie-potential sigma parameter. Gets Mie
    -parameters from the file mie.xlsx.
130 
131         :param comps: Comma seperated list of components
132         :return: N x N matrix of hard sphere diameters, where sigma_ij = 0.5 * (sigma_i + sigma_j),
133                 such that the diagonal is the radius of each component, and off-diagonals are the
    average diameter of
134                 component i and j.
135         '''
136         df = pd.read_excel(os.path.dirname(__file__)+'/mie.xlsx')
137         sigma_i = np.array([df.loc[df['comp'] == comp]['sigma'].iloc[0] for comp in comps.split(',')
    ])
138 
139         sigma_ij = 0.5 * np.sum(np.meshgrid(sigma_i, np.vstack(sigma_i)), axis=0)
140         return sigma_ij
141 
142     def omega(self, ij, l, r):
143         if ij in (1, 2):
144             return self.sigma[ij - 1] ** 2 * np.sqrt((np.pi * const.Boltzmann * self.T) / self.
    mole_weights[ij - 1]) * w(l, r)
145 
146         elif ij in (12, 21):
147             return 0.5 * self.sigma12 ** 2 * np.sqrt(2 * np.pi * const.Boltzmann * self.T / (self.m0
     * self.M1 * self.M2)) * w(l, r)
148         else:
149             raise ValueError('(' + str(ij) + ', ' + str(l) + ', ' + str(r) + ') are non-valid
    arguments for omega.')
150 
151     def A(self, p, q, r, l):
152         def inner(i):
153             return ((8 ** i * fac(p + q - 2 * i) * (-1) ** l * (-1) ** (r + i) * fac(r + 1) * fac(
154                 2 * (p + q + 2 - i)) * 4 ** r) /
```

```
                    (fac(p - i) * fac(q - i) * fac(l) * fac(i + 1 - l) * fac(r - i) * fac(p + q + 1
- i - r) * fac(
                        2 * r + 2)
                    * fac(p + q + 2 - i) * 4 ** (p + q + 1))) * ((i + 1 - l) * (p + q + 1 - i - r)
- l * (r - i))

        return summation(l - 1, min(p, q, r, p + q + 1 - r), inner)

    def A_prime(self, p, q, r, l):
        F = (self.M1 ** 2 + self.M2 ** 2) / (2 * self.M1 * self.M2)
        G = (self.M1 - self.M2) / self.M2

        def inner(w, args):
            i, k = args
            return ((8 ** i * fac(p + q - 2 * i - w) * (-1) ** (r + i) * fac(r + 1) * fac(
                2 * (p + q + 2 - i - w)) * 2 ** (2 * r) * F ** (i - k) * G ** w) /
                    (fac(p - i - w) * fac(q - i - w) * fac(r - i) * fac(p + q + 1 - i - r - w) * fac
(2 * r + 2) * fac(
                        p + q + 2 - i - w)
                    * 4 ** (p + q + 1) * fac(k) * fac(i - k) * fac(w))) * (
                            2 ** (2 * w - 1) * self.M1 ** i * self.M2 ** (p + q - i - w)) * 2 * (
                        self.M1 * (p + q + 1 - i - r - w) * delta(k, l) - self.M2 * (r - i) *
delta(k, l - 1))

        def sum_w(k, i):
            return summation(0, min(p, q, p + q + 1 - r) - i, inner, args=(i, k))

        def sum_k(i):
            return summation(l - 1, min(l, i), sum_w, args=i)

        return summation(l - 1, min(p, q, r, p + q + 1 - r), sum_k)

    def A_tripleprime(self, p, q, r, l):
        if l % 2 != 0:
            return 0

        def inner(i):
            return ((8 ** i * fac(p + q - (2 * i)) * 2 * (-1) ** (r + i) * fac(r + 1) * fac(
                2 * (p + q + 2 - i)) * 2 ** (2 * r)) /
                    (fac(p - i) * fac(q - i) * fac(l) * fac(i + 1 - l) * fac(r - i) * fac(p + q + 1
- i - r) * fac(
                        2 * r + 2)
                    * fac(p + q + 2 - i) * 4 ** (p + q + 1))) * (((i + 1 - l) * (p + q + 1 - i - r)
) - l * (r - i))

        return 0.5 ** (p + q + 1) * summation(l - 1, min(p, q, r, p + q + 1 - r), inner)

    def H_ij(self, p, q, ij):
        M1, M2 = self.M1, self.M2

        if ij == 21:  # swap indices
            M1, M2 = M2, M1

        def inner(r, l):
            return self.A(p, q, r, l) * self.omega(12, l, r)

```

```
205         def sum_r(l):
206             return summation(l, p + q + 2 - l, inner, args=l)
207 
208         val = 8 * M2 ** (p + 0.5) * M1 ** (q + 0.5) * summation(1, min(p, q) + 1, sum_r)
209 
210         return val
211 
212     def H_i(self, p, q, ij):
213 
214         if ij == 21:  # swap indices
215             self.M1, self.M2 = self.M2, self.M1
216 
217         def inner(r, l):
218             return self.A_prime(p, q, r, l) * self.omega(12, l, r)
219 
220         def sum_r(l):
221             return summation(l, p + q + 2 - l, inner, args=l)
222 
223         val = 8 * summation(1, min(p, q) + 1, sum_r)
224 
225         if ij == 21:  # swap back
226             self.M1, self.M2 = self.M2, self.M1
227 
228         return val
229 
230     def H_simple(self, p, q, i):
231         def inner(r, l):
232             return self.A_tripleprime(p, q, r, l) * self.omega(i, l, r)
233 
234         def sum_r(l):
235             return summation(l, p + q + 2 - l, inner, args=l)
236 
237         return 8 * summation(2, min(p, q) + 1, sum_r)
238 
239     def a(self, p, q):
240         if p == 0 or q == 0:
241             if p > 0:
242                 return self.M1 ** 0.5 * self.x1 * self.x2 * self.H_i(p, q, 12)
243             elif p < 0:
244                 return - self.M2 ** 0.5 * self.x1 * self.x2 * self.H_i(-p, q, 21)
245             elif q > 0:
246                 return self.M1 ** 0.5 * self.x1 * self.x2 * self.H_i(p, q, 12)
247             elif q < 0:
248                 return - self.M2 ** 0.5 * self.x1 * self.x2 * self.H_i(p, -q, 21)
249             else:  # p == 0 and q == 0
250                 return self.M1 * self.x1 * self.x2 * self.H_i(p, q, 12)
251 
252         elif p > 0 and q > 0:
253             return self.x1 ** 2 * (self.H_simple(p, q, 1)) + self.x1 * self.x2 * self.H_i(p, q, 12)
254 
255         elif p > 0 and q < 0:
256             return self.x1 * self.x2 * self.H_ij(p, -q, 12)
257 
258         elif p < 0 and q > 0:
259             return self.x1 * self.x2 * self.H_ij(-p, q, 21)
260 
```

```
261         else:   # p < 0 and q < 0
262             return self.x2 ** 2 * self.H_simple(-p, -q, 2) + self.x1 * self.x2 * self.H_i(-p, -q,
    21)
263
264     def get_alpha_T0(self, T):
265         return np.array([self.soret * T, - self.soret * T])
266
267     def plot_test(self, N, compare=True, save=False):
268         '''
269         Plot soret coefficient from kinetic gas theory for N'th order approximation with different
    m2/m1 and sigma2/sigma1 ratios
270         :param N (int): Order of approximation
271         :param compare (bool): Use same limits as Tipton, Tompson and Loyalka
272         :param save (str): Filename to save figure as (defaults to False)
273         '''
274         cmap = cm.get_cmap('viridis')
275
276         x_list = np.linspace(0.001, 0.999, 50)
277
278         s_list = np.zeros((10, 50))
279
280         sigma_list = [2, 1, 0.5]
281         self.sigma2 = 1
282
283         if compare is True:
284             m_list = np.array([1, 2, 3, 4, 5, 8, 10])
285             fig = plt.figure(figsize=(10, 5))
286             grid = gs.GridSpec(ncols=4, nrows=1, figure=fig, wspace=0.5, width_ratios=[1,1,1,0])
287
288             axs = [None for i in range(3)]
289             for i in range(3):
290                 axs[i] = fig.add_subplot(grid[i])
291
292             lim_list = [(0.05, -0.11), (0.05, -0.15), (0, -0.2)]
293
294             plt.sca(axs[0])
295             plt.hlines(0, 0, 1, colors='black', alpha=0.5)
296
297         else:
298             m_list = np.arange(0.35, 0.55, 0.05)
299             fig = plt.figure(figsize = (10,5))
300             grid = gs.GridSpec(ncols=3, nrows=1, figure=fig, wspace=0)
301
302             axs = [None for i in range(3)]
303             axs[0] = fig.add_subplot(grid[0])
304
305             for i in (1,2):
306                 axs[i] = fig.add_subplot(grid[i], sharey=axs[0])
307                 plt.setp(axs[i].get_yticklabels(), visible=False)
308
309         for i in range(3):
310             print('Making plot', i+1)
311             plt.sca(axs[i])
312             self.sigma1 = sigma_list[i]
313             self.sigma = np.array([self.sigma1, self.sigma2])
314             self.sigma12 = 0.5 * (self.sigma1 + self.sigma2)
```

```
315
316             for s_line , m in enumerate(m_list):
317                 print('m =', m)
318                 self.mole_weights = np.array([1, m])
319
320                 self.m0 = sum(self.mole_weights)
321                 self.m1, self.m2 = self.mole_weights
322                 self.M = self.mole_weights / np.sum(self.mole_weights)
323                 self.M1, self.M2 = self.M
324
325                 for k, x in enumerate(x_list):
326                     self.mole_fracs = np.array([x, 1 - x])
327                     self.x1, self.x2 = self.mole_fracs
328
329                     pq_range = np.arange(-N, N + 1, 1)
330                     self.A_matr = np.empty((2 * N + 1, 2 * N + 1), float)
331                     for ind, p in enumerate(pq_range):
332                         for j, q in zip([j for j in range(ind, len(pq_range))], pq_range[ind:]):
333                             self.A_matr[ind, j] = self.a(int(p), int(q))
334                             self.A_matr[j, ind] = self.A_matr[ind, j]
335
336                     delta_0 = (3 / 2) * np.sqrt(const.Boltzmann * self.T / self.m0)
337                     b = np.zeros(2 * N + 1)
338                     b[int((len(b) - 1) / 2)] = delta_0
339                     d = lin.solve(self.A_matr, b)
340
341                     d_1, d0, d1 = d[int(((len(d) + 1) / 2) - 2) : int(((len(d) + 1) / 2) + 1)]
342                     self.soret = -(5 / (2 * d0)) * ((self.x1 * d1 / np.sqrt(self.M1)) + (self.x2 *
    d_1 / np.sqrt(self.M2)))
343                     s_list[s_line, k] = self.soret
344
345             for m, s in zip(m_list, s_list):
346                 plt.plot(x_list, s, label = round(m,1), color = cmap(m / max(m_list)))
347
348             if compare is True:
349                 print(lim_list[i])
350                 plt.ylim(lim_list[i][1], lim_list[i][0])
351
352             plt.xlim(0,1)
353             plt.title(r'$\frac{\sigma_2}{\sigma_1} = $'+str(round(self.sigma2/self.sigma1, 2)),
    fontsize=14)
354
355         legend = plt.legend(title=r'$\frac{m_2}{m_1}$', bbox_to_anchor=[1, 1.025])
356         plt.setp(legend.get_title(), fontsize=14)
357         #plt.suptitle(r'Calculated $k_T$ values for some theoretical mixtures')
358
359         if save:
360             plt.savefig(save, dpi=600)
361         #plt.show()
```

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import matplotlib.cm as cm
4 from mpl_toolkits.mplot3d import Axes3D
5 import numpy as np
6 import warnings
```

```
7  import os
8  from scipy.interpolate import CloughTocher2DInterpolator as interpol
9
10 class Alpha_T0_empirical:
11     def __init__(self, comps, mole_fracs = [0.5, 0.5], method='wheights'):
12         '''
13         Callable object, composition and components are set upon initialization
14         __call__(temp): returns alpha_T0 for component 1 at given composition and temperature,
       approximated from experimental data
15
16         :param comps: 'comp1,comp2'
17         :param mole_fracs: mole fractions of components
18         :param method: 'wheights' (default), 'ct': CloughTocher, 'plane': linear interpolation
19         '''
20
21         self.comp1, self.comp2 = comps.split(',')
22         self.comps = comps
23
24         df = pd.read_excel(os.path.dirname(__file__) + '/alpha_t0.xlsx')
25
26         if self.comp2 in df.loc[df['Comp1'] == self.comp1]['Comp2']:
27             pass
28         elif self.comp1 in df.loc[df['Comp2'] == self.comp2]['Comp1']:
29             self.comp1, self.comp2 = self.comp2, self.comp1
30             self.flip = True
31         else:
32             raise ValueError('Components '+comps+' not found in database.')
33
34         self.data = df.loc[(df['Comp1'] == self.comp1) & (df['Comp2'] == self.comp2)]
35
36         self.c_list = np.array(self.data['c_avg'].tolist())
37         self.T_list = np.array(self.data['T_avg'].tolist())
38         self.alpha_list = np.array(self.data['Alpha_T_0'].tolist())
39
40         self.cT_ranges = (min(self.c_list), max(self.c_list), min(self.T_list), max(self.T_list))
41         self.num_points = len(self.c_list)
42
43         #Calculate span of T and c for the avalible data set
44         self.c_span = max(self.c_list) - min(self.c_list)
45         self.T_span = max(self.T_list) - min(self.T_list)
46
47         #To avoid deviding by zero
48         if self.c_span == 0:
49             self.c_span = 1
50         if self.T_span == 0:
51             self.T_span = 1
52
53         self.c = mole_fracs[0]
54
55         if method == 'wheights':
56             self.get_alpha_T0 = self.get_alpha_T0_wheights
57         elif method == 'plane':
58             self.get_alpha_T0 = self.get_alpha_T0_plane
59         elif method == 'ct':
60             self.get_alpha_T0 = self.get_alpha_T0_CloughTocher
61         else:
```

```
62            raise ValueError("method must be either 'wheights', 'plane' or 'ct'")
63
64    def get_alpha_T0_wheights(self, T, plot=False):
65        '''
66        :param comps (str) : comp1,comp2 following Thermopack convention
67        :param c (float) : Volumetric concentration at which to get kinetic_gas
68        :param T (float) : Temperature (K) at which to get kinetic_gas
69        :param plot (bool or '2d' or '3d') : display fitted data
70        :return float : kinetic_gas at the specified conditions, approximated from experimental
    data
71                        using the custom weighting scheme
72        '''
73        self.plot = plot
74        c = self.c * 100
75
76        #Check if c,T is inside the range of experimental data, warn if not
77        if any(np.array([c < self.cT_ranges[0], c > self.cT_ranges[1], T < self.cT_ranges[2], T >
    self.cT_ranges[3]])):
78            warnings.warn('\nc = '+str(round(c,0))+' T = '+str(round(T,0))+' is outside range '
79                          '\nc : ('+str(round(self.cT_ranges[0],0))+', '+str(round(self.cT_ranges
    [1],0))+'), T : '
80                          + str(round(self.cT_ranges[2], 0)) + ', '+str(round(self.cT_ranges[3],0))+
    ')')
81
82        #Distance from required point to each experimental data point
83        dist = ((self.c_list - c)/self.c_span)**2 + ((self.T_list - T)/self.T_span)**2
84
85        #Create list of indices sorted by increasing distance from required point
86        #Mask all but the first element
87        dist_ind = np.array([i for i, _ in sorted(enumerate(dist), key=lambda pair: pair[1])])
88        dist_ind_mask = np.array([False for i in dist_ind])
89        dist_ind_mask[0] = True
90
91        #Build list of c and T points that box in the required point
92        close_c = np.array([self.c_list[dist_ind[0]]])
93        close_T = np.array([self.T_list[dist_ind[0]]])
94        i = 1
95        while not self.check_inside(close_c, close_T, c, T) and i < self.num_points:
96            new_c = self.c_list[dist_ind[i]]
97            new_T = self.T_list[dist_ind[i]]
98            valid_point = self.check_valid(new_c/self.c_span, new_T/self.T_span,
99                                           close_c/self.c_span, close_T/self.T_span, c/self.c_span,
    T/self.T_span)
100           while i < self.num_points - 1 and not valid_point:
101               i += 1
102               new_c = self.c_list[dist_ind[i]]
103               new_T = self.T_list[dist_ind[i]]
104               if self.check_valid(new_c, new_T, close_c, close_T, c, T):
105                   valid_point = True
106                   break
107
108           if valid_point:
109               dist_ind_mask[i] = True
110
111           close_c = self.c_list[dist_ind[dist_ind_mask]]
112           close_T = self.T_list[dist_ind[dist_ind_mask]]
```

```
113             i += 1
114
115         #Get corresponding kinetic_gas to the experimental points that box in the required point
116         close_alpha = self.alpha_list[dist_ind[dist_ind_mask]]
117
118         #Get distance to each of the data points
119         r = dist[dist_ind[dist_ind_mask]]
120
121         #Compute wheights for a wheigted average. Closer data points count more.
122         tot = sum([np.prod(r[:i]) * np.prod(r[i + 1:]) for i in range(len(r))])
123         wheights = [np.prod(r[:i]) * np.prod(r[i + 1:]) / tot for i in range(len(r))]
124
125         #Compute wheigted average of experimental points
126         fit_alpha = sum(close_alpha * wheights)
127
128         #Compute R2, warn if high, plot if desired
129         R2 = sum((close_c - c) ** 2 + (close_T - T) ** 2 + (close_alpha - fit_alpha) ** 2) / len(
    close_alpha)
130         if plot:
131             self.plot_fit(dist, close_c, close_T, close_alpha, c, T, fit_alpha, R2)
132
133         if R2 > 500 or i == self.num_points:
134             warnings.warn('\nAlpha_T0 at c = '+str(round(c,1))+' T = '+str(round(T,0))
135                           +' for '+self.comps+' may be a bad approximation (R^2 = '+str(round(
    R2,0))+')')
136
137         return np.array([fit_alpha, -fit_alpha])
138
139     def get_alpha_T0_CloughTocher(self, T, plot=False):
140         '''
141         :param comps (str): comp1,comp2 following Thermopack convention
142         :param c (float): Volumetric concentration at which to get kinetic_gas
143         :param T (float): Temperature (K) at which to get kinetic_gas
144         :param plot (bool or '2d' or '3d'): display fitted data
145         :return float: kinetic_gas at the specified conditions, approximated from experimental
    data
146                        by CloughTocher-interpolation
147         '''
148         self.plot = plot
149         c = self.c * 100
150
151         # Check if c,T is inside the range of experimental data, warn if not
152         if any(np.array([c < self.cT_ranges[0], c > self.cT_ranges[1], T < self.cT_ranges[2], T >
    self.cT_ranges[3]])):
153             warnings.warn('\nc = ' + str(round(c, 0)) + ' T = ' + str(round(T, 0)) + ' is outside
    range '
154                                                                 '\nc: (' + str(
155                 round(self.cT_ranges[0], 0)) + ', ' + str(round(self.cT_ranges[1], 0)) + '), T: '
156                           + str(round(self.cT_ranges[2], 0)) + ', ' + str(round(self.cT_ranges[3], 0)
    ) + ')')
157
158         # Compute interpolation weights
159         fit_alpha = interpol(np.array([self.c_list, self.T_list]).transpose(), self.alpha_list)([c,T
    ])
160
161         return np.array([fit_alpha, -fit_alpha])
```

```
162
163     def get_alpha_T0_plane(self, T, plot=False):
164         '''
165         :param comps (str) : comp1,comp2 following Thermopack convention
166         :param c (float) : Volumetric concentration at which to get kinetic_gas
167         :param T (float) : Temperature (K) at which to get kinetic_gas
168         :param plot (bool or '2d' or '3d') : display fitted data
169         :return float : kinetic_gas at the specified conditions, approximated from experimental
    data
170                         by fitting a plane to the closest three data points that enclose the desired
    point
171         '''
172         self.plot = plot
173         c = self.c * 100
174
175         #Check if c,T is inside the range of experimental data, warn if not
176         if any(np.array([c < self.cT_ranges[0], c > self.cT_ranges[1], T < self.cT_ranges[2], T >
    self.cT_ranges[3]])):
177             warnings.warn('\nc = '+str(round(c,0))+' T = '+str(round(T,0))+' is outside range '
178                           '\nc : ('+str(round(self.cT_ranges[0],0))+', '+str(round(self.cT_ranges
    [1],0))+'), T : '
179                           + str(round(self.cT_ranges[2], 0)) + ', '+str(round(self.cT_ranges[3],0))+
    ')')
180
181         #Distance from required point to each experimental data point
182         dist = ((self.c_list - c)/self.c_span)**2 + ((self.T_list - T)/self.T_span)**2
183
184         #Create list of indices sorted by increasing distance from required point
185         #Mask all but the first element
186         dist_ind = np.array([i for i, _ in sorted(enumerate(dist), key=lambda pair: pair[1])])
187
188         dist_ind_mask = np.array([False for i in dist_ind])
189
190         n = 0
191         close_c = np.array([])
192         while len(close_c) < 3 and n < self.num_points:
193             dist_ind_mask = np.array([False for i in dist_ind])
194             dist_ind_mask[n] = True
195             close_c = np.array([self.c_list[dist_ind[n]]])
196             close_T = np.array([self.T_list[dist_ind[n]]])
197
198             i = SkipCounter(0,n-1)
199             new_c = self.c_list[dist_ind[i]]
200             new_T = self.T_list[dist_ind[i]]
201             valid_point = self.check_valid(new_c/self.c_span, new_T/self.T_span,
202                                            close_c/self.c_span, close_T/self.T_span, c/self.c_span,
    T/self.T_span)
203             while i < self.num_points - 1 and not valid_point:
204                 i += 1
205                 new_c = self.c_list[dist_ind[i]]
206                 new_T = self.T_list[dist_ind[i]]
207                 if self.check_valid(new_c, new_T, close_c, close_T, c, T):
208                     valid_point = True
209                     break
210
211             if valid_point:
```

```
                dist_ind_mask[i] = True

            close_c = self.c_list[dist_ind[dist_ind_mask]]
            close_T = self.T_list[dist_ind[dist_ind_mask]]

            while i < self.num_points - 1 and n != self.num_points:
                i += 1
                new_c = self.c_list[dist_ind[i]]
                new_T = self.T_list[dist_ind[i]]
                box_c = np.concatenate((close_c, [new_c]))
                box_T = np.concatenate((close_T, [new_T]))
                if self.check_inside(box_c, box_T, c, T):
                    dist_ind_mask[i] = True
                    break

            close_c = self.c_list[dist_ind[dist_ind_mask]]
            close_T = self.T_list[dist_ind[dist_ind_mask]]

            n += 1
        #Get corresponding kinetic_gas to the experimental points that box in the required point
        close_alpha = self.alpha_list[dist_ind[dist_ind_mask]]

        if len(close_c) == 3:
            close_points = np.array([close_c, close_T, close_alpha]).transpose()
            normal_vec = np.cross(close_points[1] - close_points[0], close_points[2] - close_points[0])
            fit_alpha = (np.dot(normal_vec,close_points[0]) - normal_vec[0]*c - normal_vec[1] * T) / normal_vec[2]
        else:
            #Get distance to each of the data points
            r = dist[dist_ind[dist_ind_mask]]

            #Compute wheights for a wheigted average. Closer data points count more.
            tot = sum([np.prod(r[:i]) * np.prod(r[i + 1:]) for i in range(len(r))])
            wheights = [np.prod(r[:i]) * np.prod(r[i + 1:]) / tot for i in range(len(r))]

            #Compute wheigted average of experimental points
            fit_alpha = sum(close_alpha * wheights)

        #Compute R2, warn if high, plot if desired
        R2 = sum((close_c - c) ** 2 + (close_T - T) ** 2 + (close_alpha - fit_alpha) ** 2) / len(close_alpha)
        if plot:
            self.plot_fit(dist, close_c, close_T, close_alpha, c, T, fit_alpha, R2)

        if R2 > 100:
            warnings.warn('\nAlpha_T0 at c = '+str(round(c,1))+' T = '+str(round(T,0))
                          +' for '+self.comps+' may be a bad fit (R^2 = '+str(round(R2,0))+')')
        return np.array([fit_alpha, -fit_alpha])

    def check_inside(self, x, y, c, T):
        '''
        :param x (ndarray): list of x-points
        :param y (ndarray): list of y-points
        :param c (float) : x-point
        :param T (float) : y-point
```

```
265
266         :return bool : Is the point (c,T) inside the polygon spanned by (x1,y1),(x2,y2) ,...
267         '''
268         if len(x) < 3:
269             return False
270
271         vecs = np.array([x - c, y - T]).transpose()
272         vecs = np.concatenate((vecs, np.vstack([0 for i in vecs])), axis=1)
273         indices = [i for i in range(len(vecs))]
274         for i in indices[1:]:
275             if np.cross(vecs[0], vecs[i])[-1] > 0:
276                 for j in indices[1:i] + indices[i+1:]:
277                     if np.cross(vecs[0], vecs[j])[-1] < 0 and np.cross(vecs[i], vecs[j])[-1] > 0:
278                         return True
279             else:
280                 for j in indices[1:i] + indices[i + 1:]:
281                     if np.cross(vecs[0], vecs[j])[-1] > 0 and np.cross(vecs[i], vecs[j])[-1] < 0:
282                         return True
283         return False
284
285     def check_valid(self, new_x, new_y, x_list, y_list, c, T):
286         '''
287         :param new_x (float) : x-point
288         :param new_y (float) : y-point
289         :param x_list (ndarray) : list of x-points
290         :param y_list (ndarray) : list of y-points
291         :param c (float) : x-value at point that is being approximated
292         :param T (float) : y-value at point that is being approximated
293
294         :return bool : If the point (new_x,new_y) will improve the approximated value at (c, T)
295         '''
296
297         vecs = np.array([c - x_list, T - y_list]).transpose()
298         new_point_vecs = np.array([new_x - x_list, new_y - y_list]).transpose()
299         for vec, new_point_vec in zip(vecs, new_point_vecs):
300             if np.dot(new_point_vec, vec) < 0:
301                 return False
302         return True
303
304     def plot_fit(self, dist, close_c, close_T, close_alpha, c, T, fit_alpha, R2):
305         '''
306             Plots the fitted data, good for debugging and to view errors.
307             Run __call__ with plot='2d' or plot='3d' to activate
308             NB: Does NOT display the plot. plt.show() must be called after function call to display
309         '''
310
311         self.cmap = cm.get_cmap('plasma')
312         max_alpha = max(self.alpha_list)
313
314         if self.plot == '2d' or self.plot is True:
315             plt.scatter(self.data['c_avg'], self.data['T_avg'], color = [self.cmap(self.alpha_list[i
    ]/max_alpha) for i in range(len(dist))])
316             plt.scatter(close_c, close_T, color = 'green', marker='x', label = 'Interpolation points'
    )
317             plt.scatter(c,T, color = 'black', label = 'Point to approximate')
318             plt.xlabel('c [vol%]')
```

```
319             plt.ylabel('T [K]')
320             print('plots/selected/'+str(round(c,2))+'_'+str(round(T,0)))
321             plt.savefig('plots/selected/'+str(c)[:2]+'_'+str(T)[:3])
322             plt.close()
323             #plt.show()
324
325         if self.plot == '3d' or self.plot is True:
326             fig = plt.figure()
327             ax = fig.add_subplot(111, projection='3d')
328
329             ax.scatter(self.data['c_avg'], self.data['T_avg'], self.data['Alpha_T_0'], color = 'blue',
    marker='x')
330             ax.scatter(close_c, close_T, close_alpha, color = 'red')
331             ax.scatter(c,T, fit_alpha, color = 'black')
332
333             ax.set_xlabel('c [vol%]')
334             ax.set_ylabel('T [K]')
335             ax.set_zlabel(r'$\alpha_T^\circ$ [-]')
336
337             plt.title(r'$R^2 = $'+str(round(R2,0)))
338             plt.show()
339
340         if self.plot not in ('2d','3d',True):
341             print("Argument 'plot' can be:\n"
342                   "'2d': plot 2d selection of data points\n"
343                   "'3d': plot 3d fit and selection of data points\n"
344                   "True: plot both the above")
345
346     def plot_points(self, ax):
347         '''
348         scatter experimental data points in 3d
349         :param ax: A_matr matplotlib.Axes3D instance on which to plot
350         '''
351         ax.scatter(self.data['c_avg'], self.data['T_avg'], self.data['Alpha_T_0'], color='red',
    marker='x', s = 40)
352
353     def plot_mesh(self, dim_1d = False):
354         '''
355         Plot the data-set selected upon initialization and the interpolation with the method
    selected upon initialization.
356         :param dim_1d: Set to True if data is 1d
357         '''
358         warnings.filterwarnings('ignore')
359         min_c, max_c, min_T, max_T = self.cT_ranges
360         c_list_1d = np.linspace(min_c, max_c, 25) * 0.01
361         T_list_1d = np.linspace(min_T, max_T, 25)
362
363         c_list, T_list = np.meshgrid(c_list_1d, T_list_1d)
364         fig = plt.figure()
365         ax = fig.add_subplot(111, projection='3d')
366         self.plot_points(ax)
367
368         if dim_1d:
369             alpha_vals_1d = np.array([self.get_alpha_T0(c_list_1d[i], T_list_1d[i])[0] for i in
    range(len(c_list_1d))])
370             ax.plot(c_list_1d * 100, T_list_1d, alpha_vals_1d, color='black')
```

```
371
372         else:
373             alpha_vals = np.array([[self.get_alpha_T0(c_list[j], i], T_list[j, i])[0] for i in range(
    len(c_list))]
374                                   for j in range(len(T_list))])
375
376             if self.get_alpha_T0 == self.get_alpha_T0_CloughTocher:
377                 print(alpha_vals.shape)
378                 alpha_vals = alpha_vals.transpose()[0].transpose()
379
380             ax.plot_wireframe(c_list * 100, T_list, alpha_vals, color = 'black', alpha=0.5)
381
382
383         plt.title(r'Approximated $\alpha_T^\circ$ values for '+self.comps)
384         ax.set_xlabel('c [%vol]')
385         ax.set_ylabel('T [K]')
386         ax.set_zlabel(r'$\alpha_T^\circ$ [-]')
387
388 class SkipCounter:
389     #Helper-class for get_alpha_t0_plane()
390     #increasing iterator that starts at 'val' and skips the value 'skip'
391
392     def __init__(self, val, skip):
393         if val == skip:
394             self.val = skip + 1
395         else:
396             self.val = val
397         self.skip = skip
398
399     def __lADD__(self, other):
400         if self.val + other == self.skip:
401             self.val += other + 1
402         else:
403             self.val += other
404         return self
405
406     __add__ = __lADD__
407
408     def __lt__(self, other):
409         return self.val < other
410
411     def __index__(self):
412         return self.val
413
414 def test_wheights():
415     '''
416         Testing procedure for different wheighted averages based on distance from point to different
    experimental points
417     '''
418     R2_list = []
419     min_r = []
420     miss1_list = []
421     miss2_list = []
422     miss3_list = []
423
424     for x_p in np.linspace(0.01, 0.1, 5):
```

```
425         for y_p in np.linspace(0.01, 0.1, 5):
426             x = np.array([0.01, 0.1, 0.01, 0.1])
427             y = np.array([0.01, 0.1, 0.1, 0.01])
428 
429             r = (x - x_p)**2 + (y - y_p)**2
430             f = np.exp(-np.sqrt(r))
431             w1 = f / sum(f)
432             tot = sum([np.prod(r[:i]) * np.prod(r[i+1:]) for i in range(len(r))])
433             w2 = [np.prod(r[:i]) * np.prod(r[i+1:])/tot for i in range(len(r))]
434 
435             f = x ** 2 + y
436 
437             R2_list.append(sum(r)/len(r))
438             min_r.append(min(r))
439             miss1_list.append(sum(f*w1) - (x_p**2 + y_p))
440             miss2_list.append(sum(f*w2) - (x_p**2 + y_p))
441             miss3_list.append(sum(f)/len(f) - (x_p**2 + y_p))
442 
443     plt.scatter(R2_list, abs(np.array(miss1_list)), color = 'b')
444     plt.scatter(R2_list, abs(np.array(miss2_list)), color = 'r')
445     #plt.scatter(R2_list, abs(np.array(miss3_list)), color = 'g')
446     plt.show()
447 
448     plt.scatter(min_r, abs(np.array(miss1_list)), color = 'b')
449     plt.scatter(min_r, abs(np.array(miss2_list)), color = 'r')
450     #plt.scatter(min_r, abs(np.array(miss3_list)), color = 'g')
451     plt.show()
452 
453     fig = plt.figure()
454     ax = fig.add_subplot(111, projection='3d')
455     ax.scatter(R2_list, min_r, abs(np.array(miss1_list)), color = 'b')
456     ax.scatter(R2_list, min_r, abs(np.array(miss2_list)), color = 'r')
457     #ax.scatter(R2_list, min_r, abs(np.array(miss3_list)), color = 'g')
458     ax.set_xlabel('R2')
459     ax.set_ylabel('r_min')
460 
461     plt.show()
462 
463 class DB_Builder:
464     # Build compact database of components and the data range for which they have experimental
        data points in alpha_t0.xlsx
465     # initialize with the main database-file
466     # run build_db(outfile.txt), with the desired outfile name
467 
468     def __init__(self, filename):
469         self.filename = filename
470 
471     def get_cT_range(self, comp1, comp2):
472         df = pd.read_excel(self.filename)
473         data = df.loc[(df['Comp1'] == comp1) & (df['Comp2'] == comp2)]
474 
475         min_c = min(data['c_avg'])
476         max_c = max(data['c_avg'])
477 
478         min_T = min(data['T_avg'])
479         max_T = max(data['T_avg'])
```

```
480
481         return min_c, max_c, min_T, max_T
482
483     def build_db(self, outfile):
484         df = pd.read_excel(self.filename)
485         comp1 = df['Comp1']
486         comp2 = df['Comp2']
487
488         comp_tuples = set((c1, c2) for c1,c2 in zip(comp1,comp2))
489         comp_dict = {}
490         for c1, c2 in comp_tuples:
491             if c1 in comp_dict.keys():
492                 comp_dict[c1]['Mix'].append(c2)
493             else:
494                 comp_dict[c1] = {'Mix' : [c2]}
495
496         for c1 in comp_dict.keys():
497             comp_dict[c1]['min_c'] = np.zeros(len(comp_dict[c1]['Mix']))
498             comp_dict[c1]['max_c'] = np.zeros(len(comp_dict[c1]['Mix']))
499             comp_dict[c1]['min_T'] = np.zeros(len(comp_dict[c1]['Mix']))
500             comp_dict[c1]['max_T'] = np.zeros(len(comp_dict[c1]['Mix']))
501
502             for i, c2 in enumerate(comp_dict[c1]['Mix']):
503                 min_c, max_c, min_T, max_T = self.get_cT_range(c1,c2)
504                 comp_dict[c1]['min_c'][i] = min_c
505                 comp_dict[c1]['max_c'][i] = max_c
506                 comp_dict[c1]['min_T'][i] = min_T
507                 comp_dict[c1]['max_T'][i] = max_T
508
509
510         with open(outfile, 'w') as file:
511             file.write('Comp1, Comp2, min_c, max_c, min_T, max_T\n')
512             for c1 in comp_dict.keys():
513                 for i, c2 in enumerate(comp_dict[c1]['Mix']):
514                     file.write(c1 + ', ' + c2 + ', '+
515                                str(round(comp_dict[c1]['min_c'][i],0))+', '+str(round(comp_dict[c1]['
    max_c'][i],0))+', '
516                                +str(comp_dict[c1]['min_T'][i])+', '+str(comp_dict[c1]['max_T'][i])+'
    \n')
517                 file.write('\n')
```

D.3 Plotting procedures and example-usage

All plots displaying the calculated Soret coefficients were generated by various methods in the class `DataPlotter`. The implementation of this class also serves as example-usage of the `Kempers89` and `Kempers01` classes.

```
1 '''
2 Author: Vegard G. Jervell
3 Purpose: Plot simulated data produced by the KempersXX models and demostrate usage of the models
4 Requires: Matplotlib, Pandas, Numpy, ThermoPack
5 Notes: Assumes that this file is placed in a direcory named 'plotting', that data files are
6         in a directory named 'data' in the project root directory, and that the models are
7         in a directory named 'models', also in the project root directory.
8 '''
```

```
9
10 from models.kempers01 import Kempers01
11 from models.kempers89 import Kempers89
12 from models.modKempers89_ import Mod_Kempers89
13 from pycThermopack.pyctp import cubic, pcsaft, extended_csp, cpa
14 import numpy as np
15 import pandas as pd
16 import matplotlib.pyplot as plt
17 import matplotlib.cm as cm
18 import os
19
20 root_path = os.path.dirname(os.path.abspath(__file__)).strip('plotting')
21 data_path = root_path + 'data/'
22
23 #Defining styles globaly for consistency
24 lines_dict = {'VdW' : '-', 'SRK' : '--', 'PR' : '-.', 'PT' : ':',
25               'SW' : (0, (1, 2, 1, 2, 4, 2)), 'PC-SAFT' : '-', 'SPUNG' : '-', 'CPA' : '-'}
26
27 marker_dict = {'VdW' : '', 'SRK' : '', 'PR' : '', 'PT' : '', 'SW' : '',
28                'PC-SAFT' : 'v', 'SPUNG' : 'o', 'CPA' : 's'}
29
30 eos_cmap = cm.get_cmap('viridis')
31 color_dict = {'VdW' : eos_cmap(0), 'SRK' : eos_cmap(1/6), 'PR' : eos_cmap(2/6),
32                'PT' : eos_cmap(3/6), 'SW' : eos_cmap(4/6),
33               'PC-SAFT' : eos_cmap(5/6), 'SPUNG' : eos_cmap(6/6), 'CPA' : eos_cmap(6/6)}
34
35 class DataPlotter:
36     def __init__(self, model, mode='cov', points=50, no_h0=False, pred=True, exp=True):
37         self.points = points
38         self.mode = mode
39         self.model_name = model
40         self.pred = pred
41         self.exp = exp
42         if model == 'K89':
43             self.plots_path = root_path + 'plots/kempers89/' + mode + '/'
44             self.KempersXX = Kempers89
45         elif model == 'M-K89':
46             self.plots_path = root_path + 'plots/kempers89/' + 'no_h0' + '/' + mode + '/'
47             self.KempersXX = Mod_Kempers89
48         elif model == 'K01':
49             self.plots_path = root_path + 'plots/kempers01/' + mode + '/'
50             self.KempersXX = Kempers01
51         else:
52             raise ValueError("'model' must be either 'K89', 'M-K89' or 'Kempers01'")
53
54     def etoh_h2o_T(self, eos, eos_name, cpa_flag=False, ylim=None):
55         save_path = self.plots_path + 'etoh_h2o/ethanol_water_temp_' + eos_name
56         if cpa_flag is True:
57             save_path += '_CPA'
58             eos_name += ' CPA'
59         # Get data
60         data = pd.read_csv(data_path + 'ethanol_water.csv', na_values='NaN')
61         temp_list = [int(x) for x in data.columns[1:]]
62         cons_list = data['c']
63
64         comps = 'ETOH,H2O'
```

```
65          temp_ax = np.linspace(min(temp_list), max(temp_list), self.points) + 273
66
67          # Because experimental data are given with weight fraction ethanol
68          # So must convert to mole fractions to use model
69          M_h2o = 18.02
70          M_etoh = 46.07
71
72          # set up plot
73          fig = plt.figure(figsize=(10, 5))
74          ax = plt.subplot(111)
75          ax.set_position([0.08, 0.11, 0.75, 0.77])
76          cmap = cm.get_cmap('viridis')
77
78          # Matrix containing all experimental data points
79          ST_matr = np.array([[x for x in data[str(T)]] for T in temp_list]).transpose()
80
81          # Do actual plotting
82          max_c = max(cons_list)
83          for ST, c in zip(ST_matr, cons_list):
84              x_etoh = (c / M_etoh) / ((c / M_etoh) + ((1 - c) / M_h2o))
85              x_h2o = 1 - x_etoh
86              model = self.KempersXX(comps, eos, x=[x_etoh, x_h2o])
87              plt.plot(temp_ax - 273, model.get_soret_temp(temp_ax, mode=self.mode)[0] * 1e3, color=
    cmap(c / max_c))
88              plt.scatter(temp_list, ST, color=cmap(c / max_c), label=str(c), marker='x')
89
90          plt.xlabel(r'T [$^{\circ}$C]', fontsize=14)
91          plt.ylabel(r'$s_T$ [mK$^{-1}$]', fontsize=14)
92
93          if ylim is not None:
94              plt.ylim(ylim[0], ylim[1])
95              save_path += '_ylim'
96
97          legend = plt.legend(title=r'$\omega_{EtOH}$ [-]', bbox_to_anchor=(1.01, 1.015), loc='upper
    left')
98          plt.setp(legend.get_title(), fontsize=14)
99
100         plt.sca(ax)
101         plt.title(eos_name, fontsize=14)
102
103         plt.savefig(save_path, dpi=600)
104         print('Saved', save_path + '.png')
105         plt.close(fig)
106
107     def etoh_h2o_c(self, eos, eos_name, cpa_flag=False, ylim=None):
108         save_path = self.plots_path + 'etoh_h2o/ethanol_water_cons_' + eos_name
109         if cpa_flag is True:
110             save_path += '_CPA'
111             eos_name += ' CPA'
112         # Get data
113         data = pd.read_csv(data_path + 'ethanol_water.csv', na_values='NaN')
114         temp_list = [int(x) for x in data.columns[1:]]
115         cons_list = data['c']
116
117         comps = 'ETOH,H2O'
118
```

```
119         M_h2o = 18.02
120         M_etoh = 46.07
121 
122         w_etoh_ax = np.linspace(min(cons_list), max(cons_list), self.points)
123         w_h20_ax = 1 - w_etoh_ax
124 
125         x_etoh_ax = (w_etoh_ax / M_etoh) / ((w_etoh_ax / M_etoh) + (w_h20_ax / M_h2o))
126 
127         # Set up plot
128         cmap = cm.get_cmap('cool')
129         max_T = max(temp_list)
130 
131         fig = plt.figure(figsize=(10, 5))
132         ax = plt.subplot(111)
133         ax.set_position([0.1, 0.11, 0.75, 0.77])
134 
135         for T in temp_list:
136             print(T)
137             model = self.KempersXX(comps, eos, temp=T + 273)
138             if self.pred is True:
139                 plt.plot(w_etoh_ax, model.get_soret_comp(x_etoh_ax, mode=self.mode)[1] * 1e3, color
    =cmap((T) / (max_T)))
140 
141             if self.exp is True:
142                 plt.scatter(cons_list, data[str(T)], color=cmap((T) / (max_T)), label=str(T),
    marker='x')
143 
144         plt.xlabel(r'$\omega_{{EtOH}}$ [-]', fontsize=14)
145         plt.ylabel(r'$s_T$ [mK$^{-1}$]', fontsize=14)
146 
147         if ylim is not None:
148             plt.ylim(ylim[0], ylim[1])
149             save_path += '_ylim'
150 
151         legend = plt.legend(title='T [$^{\circ}$C]', bbox_to_anchor=(1.015, 1.015), loc='upper left',
     fontsize=14)
152         plt.setp(legend.get_title(), fontsize=14)
153 
154         plt.title(eos_name)
155         plt.savefig(save_path, dpi=600)
156         print('Saved', save_path + '.png')
157         plt.close(fig)
158 
159     def toluene_n_hexane_c(self, eos, eos_name):
160         plotname = 'toluene_n_hexane_cons'
161         save_path = self.plots_path + 'toluene_hexane/' + plotname + '_' + eos_name
162         # Read in data
163         data = pd.read_excel(data_path + 'Toluene_n_hexane.xlsx')
164         temp_list = [float(x) for x in data.columns[1:]]
165         cons_list = data['x_toluene:']
166 
167         # Set up model
168         comps = 'TOLU,NC6'
169         tolu_ax = np.linspace(min(cons_list), max(cons_list), self.points)
170 
171         # Set up plot
```

```
172         cmap = cm.get_cmap('cool')
173
174         max_T = max(temp_list)
175
176         fig = plt.figure()
177
178         # Do plotting
179         for i, T in enumerate(temp_list):
180             model = self.KempersXX(comps, eos, temp=T + 273)
181             soret = model.get_soret_comp(tolu_ax, mode=self.mode)[0] * 1e3
182             plt.plot(tolu_ax, soret, color=cmap(T / max_T))
183             plt.scatter(cons_list, data[T], color=cmap(T / max_T), label=str(T), marker='x')
184
185         plt.xlabel(r'$x_{{Toluene}}$ [-]', fontsize=14)
186         plt.ylabel(r'$s_T$ [mK$^{-1}$]', fontsize=14)
187
188         plt.legend(title='T [$^{\circ}$C]', bbox_to_anchor=(1.015, 1.015), loc='upper left')
189         plt.title('Soret cofficient of toluene in n-hexane, computed with\n'
190                   +self.model_name+' using '+eos_name+' equation of state')
191
192         plt.savefig(save_path, dpi=600)
193         print('Saved ', save_path + '.png')
194         plt.close(fig)
195
196     def toluene_n_hexane_T(self, eos, eos_name):
197         # Getting data
198         plotname = 'toluene_n_hexane_temp'
199         save_path = self.plots_path + 'toluene_hexane/' + plotname + '_' + eos_name
200
201         data = pd.read_excel(data_path + 'Toluene_n_hexane.xlsx')
202
203         temp_list = [float(x) for x in data.columns[1:]]
204         cons_list = data['x_toluene:'].to_list()
205
206         comps = 'TOLU,NC6'
207         temps = np.linspace(min(temp_list), max(temp_list), self.points) + 273
208
209         # Setting up plot
210         cmap = cm.get_cmap('viridis')
211         max_c = max(cons_list)
212         fig = plt.figure()
213
214         for c in cons_list:
215             model = self.KempersXX(comps, eos, x=np.array([c, 1 - c]))
216             soret = model.get_soret_temp(temps, mode=self.mode)[0] * 1e3
217             plt.plot(temps - 273, soret, color=cmap((c) / (max_c)))
218             plt.scatter(temp_list, data.loc[data['x_toluene:'] == c].values.flatten()[1:]
219                         , color=cmap((c) / (max_c)), label=str(c), marker='x')
220
221         plt.xlabel(r'$T$ [$^{\circ}$C]', fontsize=14)
222         plt.ylabel(r'$s_T$ [mK$^{-1}$]', fontsize=14)
223
224         plt.legend(title='$x_{Toluene}$ [-]', bbox_to_anchor=(1.015, 1.015), loc='upper left')
225         plt.title('Soret cofficient of toluene in n-hexane computed with\n'+self.model_name
226                   + ' using '+eos_name+' equation of state')
227         plt.savefig(save_path, dpi=600)
```

```
228         print('Saved ', save_path + '.png')
229         plt.close(fig)
230 
231     def data_298(self, filename, comps, compname):
232         data = pd.read_excel(data_path + '298_files/' + filename + '_298.xlsx')
233         x_data = data['c']
234         ST_data = data['ST']
235 
236         x_axis = np.linspace(min(x_data), max(x_data), self.points)
237 
238         eos_list = ['VdW', 'SRK', 'PR', 'PT', 'SW', 'PC-SAFT', 'SPUNG']
239 
240         eos = cubic.cubic()
241         eos.init(comps, 'VdW')
242         model = self.KempersXX(comps, eos, temp=298, pres=1e5)
243 
244         fig, ax = plt.subplots()
245         if self.pred is True:
246             for i, eos_key in enumerate(eos_list[:-2]):
247                 eos = cubic.cubic()
248                 eos.init(comps, eos_key)
249                 model.set_eos(eos)
250                 plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[0] * 1e3,
251                          linestyle=lines_dict[eos_key], label=eos_key, color=color_dict[eos_key])
252 
253             eos = pcsaft.pcsaft()
254             eos.init(comps)
255             model = self.KempersXX(comps, eos, temp=298)
256             plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[0] * 1e3,
257                      linestyle=lines_dict['PC-SAFT'], marker=marker_dict['PC-SAFT'],
258                      label=eos_list[-2], color=color_dict['PC-SAFT'], markevery=5)
259 
260             eos = extended_csp.ext_csp()
261             eos.init(comps, 'SRK', 'Classic', 'vdW', 'NIST_MEOS', 'C3')
262             model = self.KempersXX(comps, eos, temp=298)
263             plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[0] * 1e3,
264                      linestyle=lines_dict['SPUNG'], marker=marker_dict['SPUNG'],
265                      label=eos_list[-1], color=color_dict['SPUNG'], markevery=5)
266 
267         if self.exp is True:
268             plt.scatter(x_data, ST_data, marker='x', color='black', label='Experimental')
269 
270         plt.xlabel('$x_{' + compname + '}$[-]', fontsize=14)
271         ax.set_ylabel('$S_T$ [mK$^{-1}$]', fontsize=14)
272         plt.title('Soret coefficient of '+compname+' in '+filename.split('_')[-1]+
273                   ' at 298K using '+self.model_name)
274 
275         save_path = self.plots_path + '298_files/' + filename + '_298'
276 
277         if self.pred and self.exp:
278             pass
279         elif self.pred:
280             save_path += '_pred'
281         elif self.exp:
282             save_path += '_exp'
283 
```

```
284         plt.savefig(save_path, dpi=600)
285         print('Saved', save_path + '.png')
286         plt.close(fig)
287 
288         fig = plt.figure()
289         for key in eos_list:
290             plt.plot(0, 0, linestyle=lines_dict[key], marker=marker_dict[key],
291                      color=color_dict[key], label=key)
292         plt.scatter(0, 0, color='black', marker='x', label='Experimental')
293         plt.legend(ncol=3)
294         plt.savefig(self.plots_path + '298_files/legend.png', dpi=600)
295         print('Saved', self.plots_path + '298_files/legend.png')
296         plt.close(fig)
297 
298     def plot_298(self):
299         solute_names = ['benzene', 'toluene']
300         solvent_names = ['_hexane', '_heptane']
301         solvent_codes = [',NC6', ',NC7']
302 
303         comp_codes = ['BENZENE', 'TOLU']
304         compnames = ['benzene', 'toluene']
305 
306         for i in range(2):
307             for j in range(2):
308                 self.data_298(solute_names[i] + solvent_names[j], comp_codes[i] + solvent_codes[j],
    compnames[i])
309 
310     def n_alkanes(self):
311         mixtures = ['NC10,NC5', 'NC12,NC6', 'NC12,NC7', 'NC12,NC8']
312 
313         eos_list = ['VdW', 'SRK', 'PR', 'PT', 'SW', 'PC-SAFT', 'SPUNG']
314 
315         for comps in mixtures:
316             comp1, comp2 = comps.split(',')
317 
318             save_path = self.plots_path + 'n_alkanes/'
319             save_path += comp1 + '_' + comp2
320 
321             data = pd.read_excel(data_path + 'n-alkanes/' + comp1 + '_' + comp2 + '_298K.xlsx')
322             x1_list = data['c']
323             soret_list = data['ST']
324 
325             x_axis = np.linspace(min(x1_list), max(x1_list), self.points)
326 
327             fig, ax = plt.subplots()
328             if self.pred:
329                 for i, eos_key in enumerate(eos_list[:-2]):
330                     print(eos_key, comps)
331                     eos = cubic.cubic()
332                     eos.init(comps, eos_key)
333                     model = self.KempersXX(comps, eos, temp=298)
334                     plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[1] * 1e3,
335                              linestyle=lines_dict[eos_key], label=eos_key,
336                              color=color_dict[eos_key])
337 
338                 eos = pcsaft.pcsaft()
```

```
                eos.init(comps)
                model = self.KempersXX(comps, eos, temp=298)
                plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[1] * 1e3,
                         linestyle=lines_dict['PC-SAFT'], marker=marker_dict['PC-SAFT'],
                         label=eos_list[-2], color=color_dict['PC-SAFT'], markevery=5)

                eos = extended_csp.ext_csp()
                eos.init(comps, 'SRK', 'Classic', 'vdW', 'NIST_MEOS', 'C3')
                model = self.KempersXX(comps, eos, temp=298)
                plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[1] * 1e3,
                         linestyle=lines_dict['SPUNG'], marker=marker_dict['SPUNG'],
                         label=eos_list[-1], color=color_dict['SPUNG'], markevery=5)

            if self.exp:
                plt.scatter(x1_list, soret_list, marker='x', color='black', label='Experimental')

            if self.pred and self.exp:
                pass
            elif self.pred:
                save_path += '_pred'
            elif self.exp:
                save_path += '_exp'
                plt.yticks(fontsize=15)
                plt.xticks(fontsize=15)

            plt.xlabel(r'$x_{' + comp1 + '}$ [-]', fontsize=14)
            plt.ylabel(r'$S_T$ [mK$^{-1}$]', fontsize=14)
            plt.title(comps, fontsize=16)
            plt.savefig(save_path, dpi=600)
            print('Saved', save_path + '.png')
            plt.close(fig)

            fig = plt.figure()
            for key in eos_list:
                plt.plot(0, 0, linestyle=lines_dict[key], marker=marker_dict[key],
                         color=color_dict[key], label=key)
            plt.scatter(0, 0, color='black', marker='x', label='Experimental')
            plt.legend(ncol=3)
            plt.savefig(self.plots_path + 'n_alkanes/legend.png', dpi=600)
            print('Saved', self.plots_path + 'n_alkanes/legend.png')
            plt.close(fig)

    def cold_gases(self):
        comps = 'AR,C1'
        T = 88

        save_path = self.plots_path + 'cold_gases/AR_C1_' + str(T) + 'K'
        data = pd.read_excel(data_path + 'cold_gases/AR_C1_' + str(T) + 'K.xlsx')
        x_list = data['c']
        soret_list = data['ST']

        x_axis = np.linspace(min(x_list), max(x_list), self.points)

        eos_list = ['VdW', 'SRK', 'PR', 'PT', 'SW']

        fig = plt.figure()
```

```
395
396         if self.pred is True:
397             for i, eos_key in enumerate(eos_list):
398                 print(eos_key)
399                 eos = cubic.cubic()
400                 eos.init(comps, eos_key)
401                 model = self.KempersXX(comps, eos, temp=T)
402                 plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[1] * 1e3,
403                          color=color_dict[eos_key], linestyle=lines_dict[eos_key], label=eos)
404
405             eos = extended_csp.ext_csp()
406             eos.init(comps, 'SRK', 'Classic', 'vdW', 'NIST_MEOS', 'C3')
407             model = self.KempersXX(comps, eos, temp=298)
408             plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[1] * 1e3,
409                      linestyle=lines_dict['SPUNG'], marker=marker_dict['SPUNG'],
410                      label=eos_list[-1], color=color_dict['SPUNG'], markevery=5)
411
412         if self.exp is True:
413             plt.scatter(x_list, soret_list, marker='x', color='black', label='Experimental')
414
415         plt.legend()
416         plt.xlabel(r'$x_{AR}$ [-]', fontsize=14)
417         plt.ylabel(r'$S_T$ [mK$^{-1}$]', fontsize=14)
418         plt.title('Soret coefficient of '+comps+' at '+str(T)+'K, computed with '+self.
    model_name)
419
420         plt.savefig(save_path, dpi=600)
421         print('Saved', save_path + '.png')
422         plt.close(fig)
423
424         fig = plt.figure()
425         for key in eos_list:
426             plt.plot(0,0, linestyle=lines_dict[key], marker=marker_dict[key],
427                      color=color_dict[key], label=key)
428         plt.plot(0, 0, linestyle=lines_dict['SPUNG'], marker=marker_dict['SPUNG'],
429                  color=color_dict['SPUNG'], label='SPUNG')
430         plt.scatter(0,0, marker='x', color='black', label='Experimental')
431         plt.legend(ncol=3)
432         plt.savefig(self.plots_path+'cold_gases/legend.png', dpi=600)
433         plt.close(fig)
434
435     def propanol_h2o(self):
436         save_path = self.plots_path + 'propanol_h2o'
437
438         data = pd.read_excel(data_path + 'Isopropanol.xlsx')
439         x_list = data['c']
440         soret_list = data['ST']
441
442         x_axis = np.linspace(min(x_list), max(x_list), self.points)
443
444         comps = 'PROP1OL,H2O'
445
446         fig = plt.figure()
447
448         eos_keys = ['VdW', 'SRK', 'PR', 'PT', 'SW']
449         for key in eos_keys:
```

```
            eos = cubic.cubic()
            eos.init(comps, key)
            model = self.KempersXX(comps, eos)
            plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[0] * 1e3,
                     linestyle=lines_dict[key], color=color_dict[key], label=key)

        eos = cpa.cpa()
        eos.init(comps, 'SRK')
        model = self.KempersXX(comps, eos)
        plt.plot(x_axis, model.get_soret_comp(x_axis, mode=self.mode)[0] * 1e3,
                 linestyle=lines_dict['CPA'], color=color_dict['CPA'],
                 marker=marker_dict['CPA'], label='SRK-CPA', markevery=3)

        plt.scatter(x_list, soret_list, marker='x', color='black')

        plt.ylim(-50, 50)
        plt.xlabel(r'$x_{H_2O}$ [-]')
        plt.ylabel(r'$S_T$ [mK$^{-1}$]')

        plt.legend()

        plt.savefig(save_path, dpi=600)
        print('Saved', save_path + '.png')
        plt.close(fig)

        fig = plt.figure()
        for key in eos_keys:
            plt.plot(0,0, linestyle=lines_dict[key], marker=marker_dict[key],
                     color=color_dict[key], label=key)
        plt.scatter(0,0, marker='x', color='black', label='Experimental')
        plt.legend(ncol=3)
        plt.savefig(self.plots_path+'/propanol_legend.png')
        plt.close(fig)
```

E Thermopack

ThermoPack is a SINTEF-developed open-source library for thermodynamic calculations written primarily in Fortran90 and C, with an interface to Python.[23] The library contains implementations of an extensive list of equations of state, and a database containing parameters necessary to initialize the equations of state for a wide array of chemical compounds. The following is a documentation of the most important features of ThermoPack used in this report.

E.1 Equations of State

Cubic		Non-Cubic	
Name	Key	Name	Key
Van der Waal	VdW	Lee Kesler	LK
Soave Redlich Kwong	SRK	SAFT-VR Mie	SAFT_VR_MIE
Peng Robinson	PR	PC-SAFT	PC_SAFT
Schmidt-Wensel	SW	Benedict-Webb-Rubin	Mwbr19
Patel Teja	PT	Modified Benedict-Webb-Rubin	Mwbr32
		Barker-Henderson	BH_pert
		NIST-like multiparameter EoS	Nist
		NIST-like for ideal fluid mixtures	Nist_mix

Table E.1: Equations of state implemented in ThermoPack and the corresponding keys used for initialization.

E.2 Mixing Rules and Phase keys

Name	Key	Phase	Key
Van der Waals	vdW	Two-phase	0
Wong Sandler	WS	Liquid	1
Huron Vidal	HV	Vapor	2
Huron Vidal	HV2	Minimum Gibbs	3
Reid	Reid	Single	4
NRTL	NRTL	Solid	5
UNIFAC	UNIFAC	Fake	6

Table E.2: Mixing rules and phases availible in thermopack, with the corresponding keys used to identify them.

E.3 Components

List of chemical compounds in the ThermoPack database, and the corresponding keys used when initializing an equation of state. The data-files are found in the `fluids` directory, the database can be modified at need by adding or modifying the fluid-files, running `addon/pyUtils/complist.py`, copying the generated `compdatadb.f90` to the `src` directory and recompiling ThermoPack with `make optim` from the top-level directory. The module `gen_compdb.py` generates a file `comp_db.txt` that is useful for searching for components and identifying the ID used to initialize them.

ID	Formula	Name	CAS-number	Mole wheight
BUT1OL	C4H10O	1-butanol	111-27-3	74.1216
HEX1OL	C6H14O	1-hexanol	111-27-3	102.1748
PENT1OL	C5H12O	1-pentanol	71-41-0	88.1482
PROP1OL	C3H8O	1-propanol	71-23-8	60.095
13BD	C4H6	1,3-butadiene	106-99-0	54.092
2MHX	C7H16	2-methylhexane	591-76-4	100.205
3MP	C6H14	3-methylpentane	96-14-0	86.178
ACETONE	C3H6O	Acetone	67-64-1	58.08
ACETYLEN	C2H2	Acetylen	74-86-2	26.038
ALLENE	C3H4	Propadiene	7173-51-5	40.065
NH3	NH3	Ammonia	7664-41-7	17.031
AR	AR	Argon	7440-37-1	39.948
BENZENE	C6H6	Benzene	71-43-2	78.114
CO2	CO2	Carbon dioxide	124-38-9	44.01
CO	CO	Carbon monoxide	630-08-0	28.01
CL-	Cl-	Cloride-ion	16887-00-6	35.453
ClF3Si	ClF3Si	Chlorotrifluorosilane	14049-36-6	120.5
CYCLOHEX	C6H12	Cyclohexane	110-82-7	84.161
C3-1	C3H6	Cyclopropane	75-19-4	42.081
D2	D2	Deuterium	7782-39-0	4.0282
S434	C12H26O	Di-n-hexyl ether	112-58-3	186.339
N2O4	N2O4	Dinitrogen tetroxide	10544-72-6	92.011
E-H2	H2	Equilibrium-hydrogen	1333-74-0	2.01594
C2	C2H6	Ethane	74-84-0	30.07
ETOH	ETOH	Ethanol	64-17-5	46.0684
EBZN	C8H10	Ethylbenzene	100-41-4	106.167
C2-1	C2H4	Etylene	74-85-1	28.054
HE	HE	Helium-4	7440-59-7	4.003
H2	H2	Hydrogen	1333-74-0	2.016
H2S	H2S	Hydrogen sulfide	7783-06-4	34.08
IC4	C4H10	Isobutane	75-28-5	58.124
IC5	C5H12	Isopentane	78-78-4	72.151
LJF	LJF	Lennard-jones-fluid		1.0

ID	Formula	Name	CAS-number	Mole wheight
MXYL	C8H10	M-xylene	108-38-3	106.167
C1	CH4	Methane	74-82-8	16.0425
MEOH	CH4O	Methanol	67-56-1	32.042
MTC5	C6H12	Methylcyclopentane	96-37-7	84.162
MEG	C2H6O2	Ethylene glycol	107-21-1	62.07
NC4	C4H10	N-butane	106-97-8	58.124
NC10	C10H22	N-decane	124-18-5	142.286
NC22	C22H46	N-docosane	629-97-0	310.61
NC12	C12H26	N-dodecane	112-40-3	170.34
NC20	C20H42	N-eicosane	112-95-8	282.55
NC21	C21H44	N-heneicosane	629-94-7	296.58
NC17	C17H36	N-heptadecane	629-78-7	240.47
NC7	C7H16	N-heptane	142-82-5	100.205
NC16	C16H34	N-hexadecane	544-76-3	226.446
NC6	C6H14	N-hexane	110-54-3	86.178
NC19	C19H40	N-nonadecane	629-92-5	268.53
NC9	C9H20	N-nonane	111-84-2	128.259
NC18	C18H38	N-octadecane	593-45-3	254.5
NC8	C8H18	N-octane	111-65-9	114.232
NC25	C25H52	N-pentacosane	629-99-2	352.69
NC15	C15H32	N-pentadecane	629-62-9	212.42
NC5	C5H12	N-pentan	109-66-0	72.151
NC24	C24H50	N-tetracosane	646-31-1	338.66
NC14	C14H30	N-tetradecane	629-59-4	198.39
NC23	C23H48	N-tricosane	638-67-5	324.63
NC13	C13H28	N-tridecane	629-50-5	184.37
NC11	C11H24	N-undecane	1120-21-4	156.312
NE	NE	Neon	7440-01-9	20.183
NO	NO	Nitric oxide	10102-43-9	30.0061
N2	N2	Nitrogen	7727-37-9	28.013
N2O	N2O	Nitrous oxide	10024-97-2	44.013
N-H2	H2	N-hydrogen	1333-74-0	2.01594
O-H2	H2	Ortho-hydrogen	1333-74-0	2.01594
OXYL	C8H10	O-xylene	95-47-6	106.167
O2	O2	Oxygen	7782-44-7	31.999
P-H2	H2	Para-hydrogen	1333-74-0	2.01594
PXYL	C8H10	P-xylene	106-42-3	106.167
C3	C3H8	Propane	74-98-6	44.097
PRLN	C3H6	Propylene	115-07-1	42.081
R11	CCL3F	Trichlorofluoromethane	75-69-4	137.368
R1114	C2F4	Tetrafluoroethylene	116-14-3	100.016

ID	Formula	Name	CAS-number	Mole wheight
R1132a	C2H2F2	1,1-difluoroethylene	75-38-7	64.035
R114	C2CL2F4	1,2-dichlorotetrafluoroethane	76-14-2	170.922
R115	C2CLF5	Chloropentafluoroethane	76-15-3	154.467
R116	C2F6	Hexafluoroethane	76-16-4	138.012
R12	CCL2F2	Dichlorodifluoromethane	75-71-8	120.914
R1234yf	CF3CF=CH2	2,3,3,3-tetrafluoropropene	754-12-1	114.041593
R1234ze	CHF=CHCF3-(t	Trans-1,3,3,3-tetrafluoropropene	29118-24-9	114.0416
R124	C2HCLF4	2-chloro-1,1,1,2-tetrafluoroethane	2837-89-0	136.475
R124a	C2HCLF4	1-chloro-1,1,2,2-tetrafluoroethane	354-25-6	136.475
R125	C2HF5	Pentafluoroethane	354-33-6	120.03
R13	CCLF3	Chlorotrifluoromethane	75-72-9	104.459
R134a	C2H2F4	1,1,1,2-tetrafluoroethane	811-97-2	102.03
R14	CF4	Carbon tetrafluoride	75-73-0	88.005
R142b	C2H3ClF2	1-chloro-1,1-difluoroethane	75-68-3	100.496
R143a	C2H3F3	1,1,1-trifluoroethane	420-46-2	84.041
R152a	C2H4F2	1,1-difluoroethane	75-37-6	66.051
R21	CHCL2F	Dichlorofluoromethane	75-43-4	102.923
R218	C3F8	Octafluoropropane	76-19-7	188.017
R22	CHCLF2	Chlorodifluoromethane	75-45-6	86.469
R23	CHF3	Trifluoromethane	75-46-7	70.013
R32	CH2F2	Difluoromethane	75-10-5	52.023
R41	CH3F	Methyl fluoride	593-53-3	34.033
NA+	Na+	Sodium-ion	17341-25-2	22.99
F6S	F6S	Sulfur hexafluoride	2551-62-4	146.054
SO2	SO2	Sulfur dioxide	7446-09-5	64.065
F4N2	F4N2	Tetrafluorohydrazine	10036-47-2	104.016
TOLU	C7H8	Toluene	108-88-3	92.141
F3NO	F3N0	Trifluoroamineoxide	13847-65-9	87.001
H2O	H2O	Water	7732-18	18.015
XE	XE	Xenon	7440-63-3	131.3

E.4 Class thermopack

Methods in the thermopack-class are inherited by all equation of state-classes. The methods present here are the primary methods for interfacing with an equation of state.

E.4.1 Phase-properties

specific_volume(self, temp, press, x, phase, dvdt=None, dvdp=None, dvdn=None)

Calculate single-phase specific volume. Note that the order of the output match the default order of input for the differentials. Note further that dvdt, dvdp and dvdn only are flags to enable calculation.
Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Molar composition

phase (int):

Calcualte root for specified phase

dvdt (logical, optional):

Calculate volume differentials with respect to temperature while pressure and composition are held constant. Defaults to None.

dvdp (logical, optional):

Calculate volume differentials with respect to pressure while temperature and composition are held constant. Defaults to None.

dvdn (logical, optional):

Calculate volume differentials with respect to mol numbers while pressure and temperature are held constant. Defaults to None.

Returns:

float: Specific volume (m3/mol), and optionally differentials

enthalpy(self, temp, press, x, phase, dhdt=None, dhdp=None, dhdn=None)

Calculate specific single-phase enthalpy. Note that the order of the output match the default order of input for the differentials. Note further that dhdt, dhdp and dhdn only are flags to enable calculation.

Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Molar composition

phase (int):

Calcualte root for specified phase

dhdt (logical, optional):

Calculate enthalpy differentials with respect to temperature while pressure and composition are held constant. Defaults to None.

dhdp (logical, optional):

Calculate enthalpy differentials with respect to pressure while temperature and composition are held constant. Defaults to None.

dhdn (logical, optional):

Calculate enthalpy differentials with respect to mole numbers while pressure and temperature are held constant. Defaults to None.

Returns:

float: Specific enthalpy (J/mol), and optionally differentials

entropy(self,temp,press,x,phase,dsdt=None,dsdp=None,dsdn=None)

Calculate specific single-phase entropy. Note that the order of the output match the default order of input for the differentials. Note further that dsdt,

dhsp and dsdn only are flags to enable calculation.

Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Molar composition

phase (int):

Calcualte root for specified phase

dsdt (logical, optional):

Calculate entropy differentials with respect to temperature while pressure and composition are held constant. Defaults to None.

dsdp (logical, optional):

Calculate entropy differentials with respect to pressure while temperature and composition are held constant. Defaults to None.

dsdn (logical, optional):

Calculate entropy differentials with respect to mol numbers while pressure and temperature are held constant. Defaults to None.

Returns:

float: Specific entropy (J/mol/K), and optionally differentials

idealenthalpysingle(self,temp,press,j,dhdt=None,dhdp=None)

Calculate specific ideal enthalpy. Note that the order of the output match the default order of input for the differentials. Note further that dhdt, and dhdp only are flags to enable calculation.

Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Molar composition

phase (int):

Calcualte root for specified phase

dhdt (logical, optional):

Calculate ideal enthalpy differentials with respect to temperature while pressure and composition are held constant. Defaults to None.

dhdp (logical, optional):

Calculate ideal enthalpy differentials with respect to pressure while temperature and composition are held constant. Defaults to None.

Returns:

float: Specific ideal enthalpy (J/mol), and optionally differentials

thermo(self,temp,press,x,phase, **kwargs)

Calculate logarithm of fugacity coefficient given composition, temperature and pressure. Note that the order of the output match the default order of input for the differentials. Note further that dlnfugdt, dlnfugdp, dlnfugdn and ophase only are flags to enable calculation.

Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Molar composition (.)

phase (int):

Calcualte root for specified phase

dlnfugdt (logical, optional):

Calculate fugacity coefficient differentials with respect to temperature while pressure and composition are held constant. Defaults to None.

dlnfugdp (logical, optional):

Calculate fugacity coefficient differentials with respect to pressure while temperature and composition are held constant. Defaults to None.

dlnfugdn (logical, optional):

Calculate fugacity coefficient differentials with respect to mol numbers while pressure and temperature are held constant. Defaults to None.

ophase (int, optional):

Phase flag. Only set when phase=MINGIBBSPH.

v (float, optional):

Specific volume (m3/mol)

Returns:

ndarray: fugacity coefficient (-), and optionally differentials

zfac(self, temp, press, x, phase, dzdt=None, dzdp=None, dzdn=None)

Calculate single-phase compressibility. Note that the order of the output match the default order of input for the differentials. Note further that dzdt, dzdp and dzdn only are flags to enable calculation.

Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Molar composition

phase (int):

Calcualte root for specified phase

dzdt (logical, optional):

Calculate compressibility differentials with respect to temperature while pressure and composition are held constant. Defaults to None.

dzdp (logical, optional):

Calculate compressibility differentials with respect to pressure while temperature and composition are held constant. Defaults to None.

dzdn (logical, optional):

Calculate compressibility differentials with respect to mol numbers while pressure and temperature are held constant. Defaults to None.

Returns:

float: Compressibility (-), and optionally differentials

speed_of_sound(self, temp, press, x, y, z, betaV, betaL, phase)

Calculate speed of sound for single phase or two phase mixture assuming mechanical, thermal and chemical equilibrium.

Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

x (array-like):

Liquid molar composition

y (array-like):

Gas molar composition

z (array-like):

Overall molar composition

betaV (float):

Molar gas phase fraction

betaL (float):

Molar liquid phase fraction

phase (int):

Calcualte root for specified phase

Returns:

float: Speed of sound (m/s)

E.4.2 Flash-interfaces

set_ph_tolerance(self, tol)

Set tolerance of isobaric-isentalpic (PH) flash
Parameters:

tol (float):

Tolerance

two_phase_psflash(self,press,z,entropy,temp=None)

Do isentropic-isobaric (SP) flash
Parameters:

press (float):

Pressure (Pa)

z (array-like):

Overall molar composition

entropy (float):

Specific entropy (J/mol/K)

temp (float, optional):

Initial guess for temperature (K)

Returns:

temp (float): Temperature (K)
x (ndarray): Liquid molar composition
y (ndarray): Gas molar composition
betaV (float): Molar gas phase fraction
betaL (float): Molar liquid phase fraction
phase (int): Phase identifier (iTWOPH/iLIQPH/iVAPPH)

two_phase_tpflash(self,temp,press,z)

Do isothermal-isobaric (TP) flash
Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

z (array-like):

Overall molar composition

Returns:

x (ndarray): Liquid molar composition
y (ndarray): Gas molar composition
betaV (float): Molar gas phase fraction
betaL (float): Molar liquid phase fraction
phase (int): Phase identifier (iTWOPH/iLIQPH/iVAPPH)

two_phase_phflash(self,press,z,enthalpy,temp=None)

Do isenthalpic-isobaric (HP) flash
Parameters:

press (float):

Pressure (Pa)

z (array-like):

Overall molar composition

enthalpy (float):

Specific enthalpy (J/mol)

temp (float, optional):

Initial guess for temperature (K)

Returns:

temp (float): Temperature (K)
x (ndarray): Liquid molar composition
y (ndarray): Gas molar composition
betaV (float): Molar gas phase fraction
betaL (float): Molar liquid phase fraction
phase (int): Phase identifier (iTWOPH/iLIQPH/iVAPPH)

two_phase_uvflash(self,z,specific_energy,specific_volume,temp=None,press=None)

Do isoenergetic-isochoric (UV) flash
Parameters:

press (float):

Pressure (Pa)

z (array-like):

Overall molar composition

specific_energy (float):

Specific energy (J/mol)

specific_volume (float):

Specific volume (m3/mol)

temp (float, optional):

Initial guess for temperature (K)

press (float, optional):

Initial guess for pressure (Pa)

Returns:

temp (float): Temperature (K)
press (float): Pressure (Pa)
x (ndarray): Liquid molar composition
y (ndarray): Gas molar composition
betaV (float): Molar gas phase fraction
betaL (float): Molar liquid phase fraction
phase (int): Phase identifier (iTWOPH/iLIQPH/iVAPPH)

guess_phase(self, temp, press, z)

If only one root exsist for the equation of state the phase type can be determined from either the psedo-critical volume or a volume ratio to the co-volume
Parameters:

temp (float):

Temperature (K)

press (float):

Pressure (Pa)

Returns:

int: Phase int (VAPPH or LIQPH)

E.4.3 TV-property interfaces

pressure_tv(self, temp, volume, n, dpdt=None, dpdv=None, dpdn=None)

Calculate pressure given temperature, volume and mol numbers.
Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dpdt (No type, optional):

Flag to activate calculation. Defaults to None.

dpdv (No type, optional):

Flag to activate calculation. Defaults to None.

dpdn (No type, optional):

Flag to activate calculation. Defaults to None.

Returns:

float: Pressure (Pa)

internal_energy_tv(self, temp, volume, n, dedt=None, dedv=None)

Calculate internal energy given temperature, volume and mol numbers.
Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dedt (logical, optional):

Flag to activate calculation. Defaults to None.

dedv (logical, optional):

Flag to activate calculation. Defaults to None.

Returns:

float: Energy (J), optionally energy differentials

enthalpy_tv(self, temp, volume, n, dhdt=None, dhdv=None, dhdn=None)

Calculate enthalpy given temperature, volume and mol numbers.
Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dhdt (logical, optional):

Flag to activate calculation. Defaults to None.

dhdv (logical, optional):

Flag to activate calculation. Defaults to None.

dhdn (logical, optional):

Flag to activate calculation. Defaults to None.

Returns:

float: Enthalpy (J), optionally enthalpy differentials

helmholtz_tv(self, temp, volume, n, dadt=None, dadv=None)

Calculate Helmholtz energy given temperature, volume and mol numbers.

Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dadt (logical, optional):

Flag to activate calculation. Defaults to None.

dadv (logical, optional):

Flag to activate calculation. Defaults to None.

Returns:

float: Helmholtz energy (J) Optionally energy differentials

chemical_potential_tv(self, temp, volume, n, dmudt=None, dmudv=None, dmudn=None)

Calculate chemical potential given temperature, volume and mol numbers.
Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dmudt (logical, optional):

Flag to activate calculation. Defaults to None.

dmudv (logical, optional):

Flag to activate calculation. Defaults to None.

dmudn (logical, optional):

Flag to activate calculation. Defaults to None.

Returns:

float: Chemical potential (J/mol), optionally chemical potential differentials

fugacity_tv(self, temp, volume, n, dlnphidt=None, dlnphidv=None, dlnphidn=None)

Calculate natural logarithm of fugacity given temperature, volume and mol numbers.
Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dlnphidt (No type, optional):

Flag to activate calculation. Defaults to None.

dlnphidv (No type, optional):

Flag to activate calculation. Defaults to None.

dlnphidn (No type, optional):

Flag to activate calculation. Defaults to None.

Returns:

ndarray: Natural logarithm of fugacity, optionally differentials

entropy_tv(self, temp, volume, n, dsdt=None, dsdv=None, dsdn=None)

Calculate entropy given temperature, volume and mol numbers.

Parameters:

temp (float):

Temperature (K)

volume (float):

Volume (m3)

n (array-like):

Mol numbers (mol)

dsdt (No type, optional):

Flag to activate calculation. Defaults to None.

dsdv (No type, optional):

Flag to activate calculation. Defaults to None.

dsdn (No type, optional):

Flag to activate calculation. Defaults to None.

Returns:

float: Entropy (J/K)