NUMERICAL SIMULATION OF VAPOR-LIQUID EQUILIBRIA OF A WATER-ETHANOL MIXTURE

by

Michael Ikeda

B.S. in Mechanical Engineering, California Institute of Technology,

2007

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

M.S. in Mechanical Engineering

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Michael Ikeda

It was defended on

October 5th, 2010

and approved by

Laura A. Schaefer, Ph. D., Professor

Peyman Givi, Ph. D., Professor

Joseph McCarthy, Ph. D., Professor

Thesis Advisor: Laura A. Schaefer, Ph. D., Professor

NUMERICAL SIMULATION OF VAPOR-LIQUID EQUILIBRIA OF A WATER-ETHANOL MIXTURE

Michael Ikeda, M.S.

University of Pittsburgh, 2010

Vapor-liquid equilibrium studies are important to many engineering disciplines. Numerical simulations using empirical equations of state provide an excellent alternative to time consuming experimental measurement. A new methodology is developed to visualize the results from vapor-liquid equilibrium numerical studies of an aqueous alcohol binary mixture. The goal is to provide a better technique to determine the cubic equation of state, mixing rule, and combining rule combinations that will improve the predictability of the simulations, by reducing their dependence on binary interaction parameters. With an improved understanding of the various equations used in vapor-liquid equilibrium models, simulations can be more reliably used to predict data under conditions in which experimental data are unavailable or not easily obtainable. A vapor-liquid equilibrium simulation program is developed that can model fluid mixtures with assorted equation of state, mixing rule, and combining rule blends. A model's success is appraised via both convergence and performance metrics over large ranges of binary interaction pairs. It is shown that increases in equation complexity typically lead to improved correlative accuracy. However, models that converge for large numbers of pairs, and do so with good performance, are chosen as the most predictive combinations due to their ability to reproduce data even with a lack of decent binary interaction parameters. Furthermore, the relationships between the binary interaction pairs are examined. For the arithmetic and conventional combining rules, it is observed that only one experimental fitting parameter is required, for the system under consideration. Using the designed flexibility of this model, other equations and systems can be incorporated in the future, leading to the development of enhanced mixing and combining rules that are linked to specific equations of state, which increase the predictability, and consequently the usability, of the equations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION AND BACKGROUND

1.1 INTRODUCTION

Vapor-liquid equilibrium, or VLE, refers to the thermodynamic condition in which the liquid and vapor phases of a substance co-exist in a stable equilibrium state. More specifically, equilibrium can be broken down into three types: thermal, mechanical, and chemical potential. Thermal equilibrium can be expressed as a lack of net heat transfer between phases, resulting in equal temperatures of both phases ($T_{liquid} = T_{vapor}$). Mechanical equilibrium represents a balance of forces between the phases. Neglecting interfacial tension due to curved interfaces, this corresponds to equal pressures on both phases ($P_{liquid} = P_{vapor}$). Finally, chemical potential equilibrium implies that the rate of evaporation and the rate of condensation are equal. At constant temperature and pressure, this represents a minimum in the system free energy. Microscopically, there is no difference between an equilibrium and a non-equilibrium state. Molecules are colliding, evaporating, and condensing in both situations. However, on the macroscale, equilibrium signifies that there are no *net* changes occurring in the system. While it would technically take an infinite amount of time to reach an equilibrium state, VLE studies are interested in the practical equilibria that are reached in a finite time, as is common in the field of thermodynamics. Data obtained through a VLE analysis include the temperatures, pressures, and compositions at which the substances of interest exist in vapor-liquid equilibrium and specifically, the conditions at which they are saturated liquids or saturated vapors. As will be illustrated in a few brief examples below, an understanding of VLE processes is vital to a number of engineering disciplines.

A typical chemical plant, diagrammed in Figure 1, is comprised of a chemical reactor and a number of separators, which often operate on equilibrium principles [1]. These separators rely on the data obtained through VLE studies to determine the optimal operating temperatures and pressures in order to separate chemicals to varying levels of purity. This is important as the temperature and pressure necessary to separate a mixture to one level of purity can be completely different from another level. For example, at atmospheric pressure, if a 10% mixture of ethanol in water is heated to around 365 K, at the onset of boiling, the vapor produced will contain approximately 30% ethanol and 70% water. However, as the base mixture approaches around 80% ethanol and 20% water, the vapor produced will contain only slightly more ethanol than water, and the temperature required to boil the mixture will be more than 10 K lower. In fact, to purify ethanol beyond 96% it becomes necessary to use desiccants rather than distillation separators as the vapor released during boiling has the same composition as the base mixture. In other words, the vapor coming off a 96% ethanol mixture is composed of 96% ethanol as well.

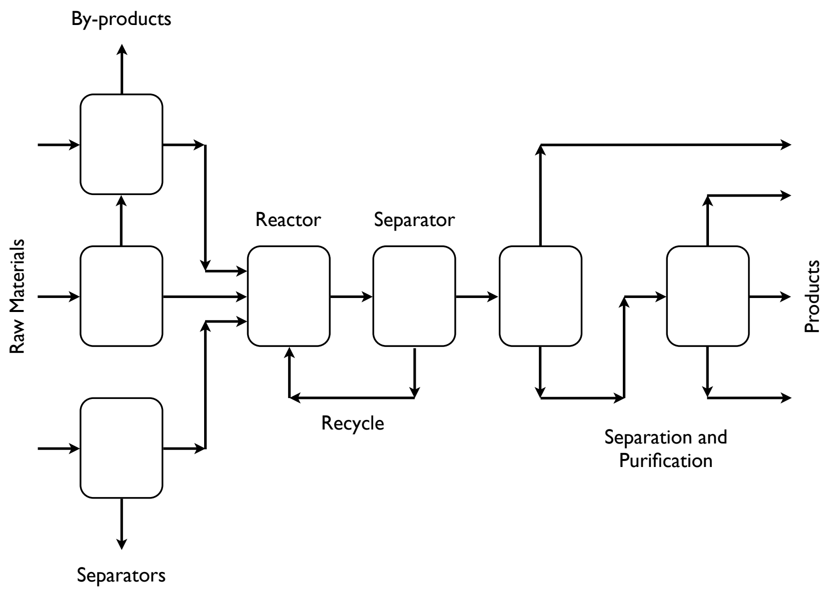


Figure 1: Typical chemical plant layout. Adapted from Wankat [1].

Heat pipes, such as the one depicted in Figure 2, are often used in electronics for cooling, and rely on flow boiling to enhance heat transfer characteristics [2]. In order to optimize this heat transfer, the two-phase flow profile must be understood, which requires knowledge of the thermodynamic properties of the system at the vapor-liquid equilibrium point.

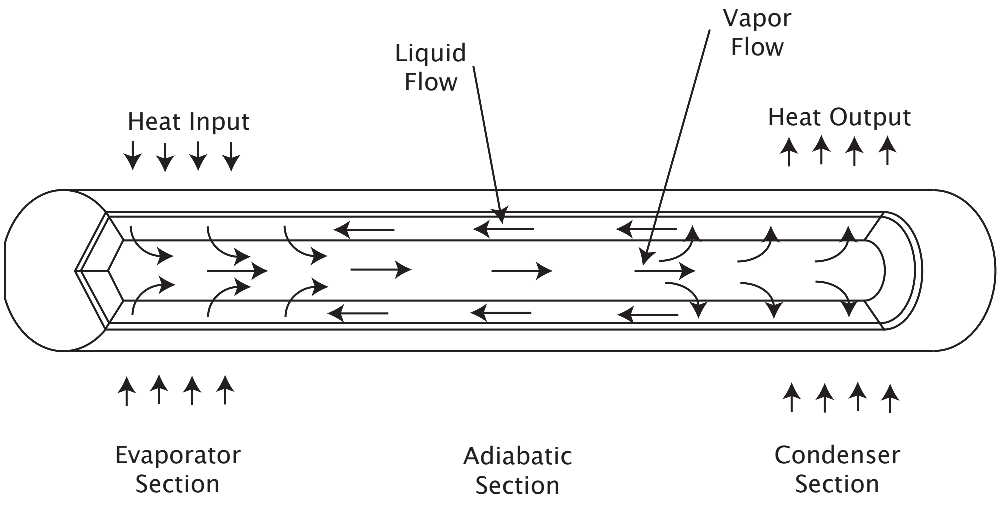


Figure 2: Diagram of a heat pipe. Adapted from M & M Metals. [2].

Absorption refrigeration systems rely on the use of two-phase flow for both heat exchange and chemical separation. An example system is shown in Figure 3 [3]. Analyzing the performance of these refrigerators requires accurate thermodynamic data at various locations throughout the system, where substances often exist in vapor-liquid equilibrium. Furthermore, the operating parameters, and even the fluids used within such systems, can be optimized using VLE data.

From the distillation of ethanol to the cooling of components in a computer, vapor-liquid equilibria data are very widely used. Traditionally, experimental studies are undertaken to determine VLE characteristics. This process can be carried out using a variety of methods, but one of the most common is the use of a static equilibrium cell, such as the one pictured in Figure 4 [4].

This apparatus is operated by placing a system with a fixed composition inside a cell. The system is then allowed to reach equilibrium under a fixed temperature or pressure. Equilibrium can be verified in a number of ways, such as checking total pressure stability or using a sampling system which checks that the phase compositions are not changing [5]. These systems are fairly simple, but after each temperature or pressure variation, the system

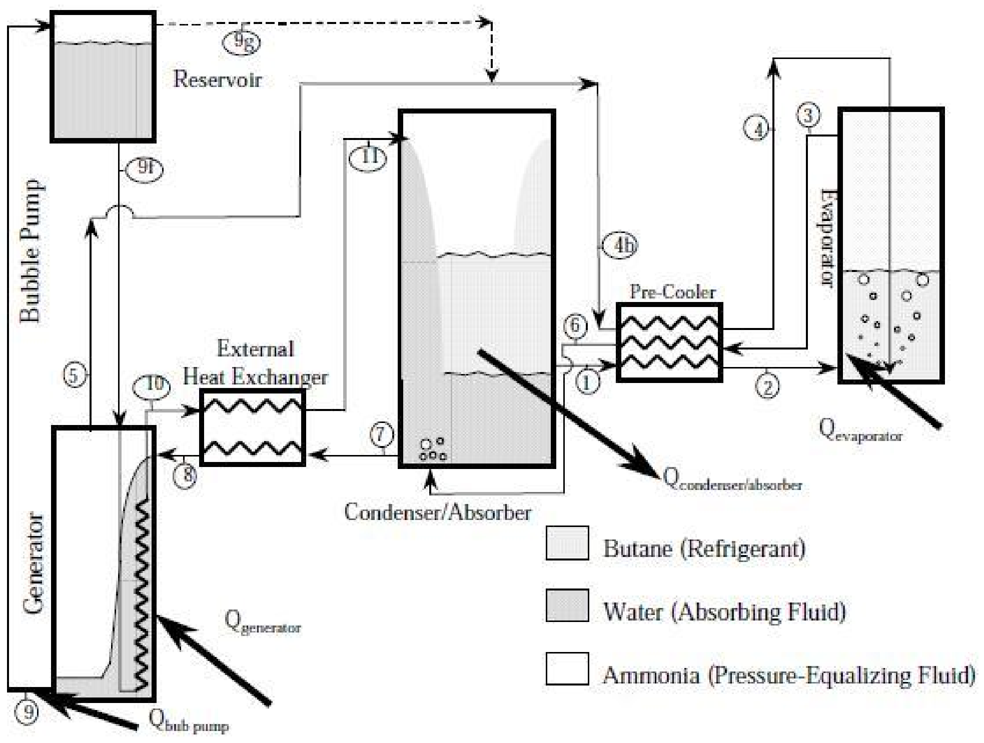


Figure 3: An example of a single-effect absorption refrigeration cycle. Taken from Schaefer [3].

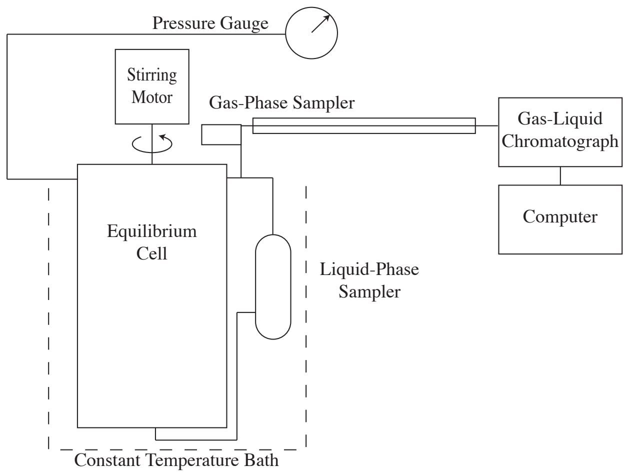


Figure 4: A typical static equilibrium cell used for experimental VLE calculations Adapted from Pawlikowski, et al. [4]

must be allowed to re-equilibrate before the phase compositions can be determined. This process must be repeated for every temperature, pressure, and composition where data are desired. As a result, complete data sets of experimental VLE curves are very time consuming to produce.

To reduce the time required to obtain VLE data, numerical methods can be used to model the data instead. As will be discussed in this work, one form of this modeling is based on the calculation of fugacity coefficients derived from equations of state. Thus, the accuracy of simulated VLE data is highly dependent on the limitations of the modeling equations that are used. This research seeks to determine the best equation of state, mixing, and combining rule combinations that increase the usability of the VLE simulation by reducing their dependence on experimental data. By developing an understanding of the relationship between all the equations used in a model, simulations can be more reliably used to predict VLE data under conditions in which experimental data is not available nor easily obtainable. Using *Fortran 90*, a VLE simulation code was developed that can model fluid mixtures with various equations. A model's success is classified by both convergence and performance metrics over large ranges of experimental fitting parameters. Models that converge for large numbers of pairs, and do so with good performance, are chosen as the most predictive combinations due to their ability to continue to reproduce data even with a lack of decent experimental data. A method to visualize the difference between correlative and predictive equations is presented with the hope that a better understanding of these principles may lead to the development of enhanced equations with increased predictive ability, which consequently, could increase the usability of those equations.

The following sections will provide an overview of the major thermodynamic concepts briefly mentioned here, including dew and bubble points, fugacities, equations of state, and mixing and combining rules.

1.2 DEW POINT AND BUBBLE POINT DATA

The primary objective of a vapor-liquid equilibrium calculation is the determination of dew and bubble points. A bubble point corresponds to the temperature, pressure, and composition of a fluid mixture at which the mixture exists as a saturated liquid. Any increase in temperature or decrease in pressure from these determined values would result in the vaporization of one or both components in the mixture, leading to a vapor-liquid mixture. A dew point, on the other hand, corresponds to the existence of a saturated vapor mixture, where a decrease in temperature or an increase in pressure would cause the condensation of liquid, resulting in a vapor-liquid mixture. Bubble point and dew point data is typically shown for a fixed temperature or a fixed pressure. The variable property acts as the dependent variable and the composition as the independent variable. Figure 5 shows a fixed temperature, pressure - composition curve and Figure 6 shows a fixed pressure, temperature - composition curve.

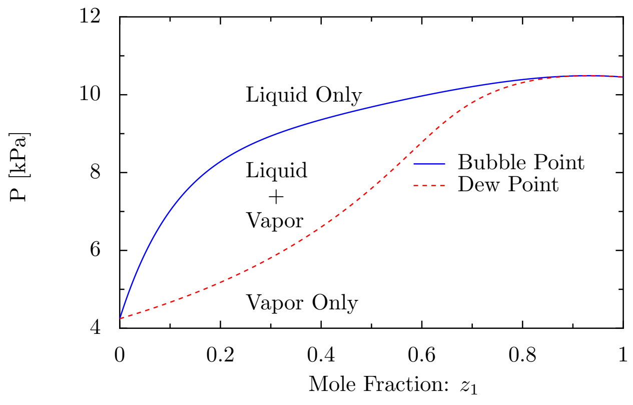


Figure 5: Pressure vs. composition of a Water-EtOH mixture at 303.15 K, showing the bubble point data as a solid blue curve and the dew point data as a dashed red curve.

It is worth noting here that the thermodynamic definition of equilibrium can be expressed in a number of equivalent forms, but the two most basic equate the Gibbs free energies, G, or the chemical potentials, μ, between various phases, $I, II, III, \ldots$, as shown in Equations (1.1) and (1.2), respectively:

$$G^{I}(T,P) = G^{II}(T,P) = G^{III}(T,P) = \cdots, \tag{1.1}$$

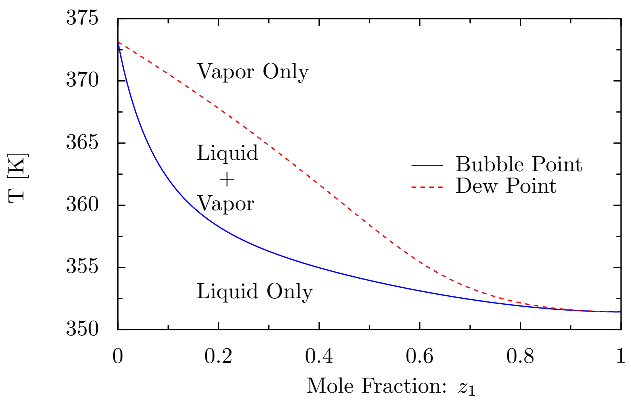


Figure 6: Temperature vs. composition of a Water-EtOH mixture at 101,325 Pa, showing the bubble point data as a solid blue curve and the dew point data as a dashed red curve.

$$\mu^{I}(T,P) = \mu^{II}(T,P) = \mu^{III}(T,P) = \cdots . \tag{1.2}$$

It is the determination of the thermodynamic properties, such that these equations are satisfied, that provides the dew and bubble point data of interest. This solution is obtained using iterative numerical methods that will be described in Chapter 4, following an in-depth presentation of the underlying mathematics.

1.3 FUGACITY

Fugacity is a concept that often causes confusion. This is due to its entirely mathematical definition, that is often introduced without a physical interpretation. However, a better understanding of what the fugacity is can be obtained by examining its derivation. To begin, the Gibbs-Duhem equation for a pure substance, as shown in Equation (1.3), relates the chemical potential to the thermodynamic properties of the system:

$$d\mu = -S_m dT + V_m dP, \tag{1.3}$$

where S_m and V_m are the molar entropy and volume, respectively. Note that molar quantities are assumed throughout this derivation unless specifically noted. The subscript, m, for the molar volume will be maintained to avoid confusion as the total volume, labeled V_t, will be used in later discussions. Following the derivation provided by Richet, the Gibbs-Duhem equation can be integrated, first isothermally and then isobarically, from (T_0, P_0) to (T, P) so that [6]:

$$\mu - \mu\left(T_0, P_0\right) = -\int_{T_0}^{T} S_m dT + \int_{P_0}^{P} V_m dP. \tag{1.4}$$

For simplicity, the standard chemical potential, μ^0, can then be defined as the chemical potential of a substance at its standard state with a fixed pressure of $P_0 = 1\ bar$, so that:

$$\mu^\circ = \mu\left(T_0, P_0\right) - \int_{T_0}^{T} S_m dT. \tag{1.5}$$

Using this definition, Equation (1.4) can be rewritten as:

$$\mu = \mu^\circ + \int_{P_0}^{P} V_m dP. \tag{1.6}$$

For an ideal gas $V_m = \frac{RT}{P}$, which when substituted into Equation (1.6), leads to:

$$\mu = \mu^\circ + RT \ln\left(\frac{P}{P_0}\right), \tag{1.7}$$

after integration. Note that this expression can be used interchangibly with the equation of state as the definition of an ideal gas. In order to maintain the form of Equation (1.7), the fugacity, f, was introduced on a purely mathematical basis such that Equation (1.8) holds *exactly* for a real substance:

$$\mu = \mu^\circ + RT \ln\left(\frac{f}{f_0}\right). \tag{1.8}$$

Finally, comparing this relation to the corresponding form for an ideal gas shown in Equation (1.7), it becomes clear that the fugacity is just a theoretical pressure that, at a given temperature, is required to make a non-ideal gas satisfy an equation for the chemical potential of an ideal gas [7]. In other words, the fugacity represents the theoretical pressure of a system, where the real gas would take on the properties that an ideal gas has at the actual pressure of the system.

A few important attributes of the fugacity arise from its mathematical underpinning. First, it must obey the following limit:

$$\lim_{P \to 0} f = P. \tag{1.9}$$

This ensures that Equation (1.8) will reduce to Equation (1.7) when the pressure goes to zero, the theoretical condition where all substances behave as ideal gases.

Furthermore, an interesting problem arises when the fugacity is introduced with the integral form of the chemical potential. This is obtained by combining Equations (1.6) and (1.8), yielding:

$$RT \ln\left(\frac{f}{f_0}\right) = \int_{P_0}^{P} V_m dP. \tag{1.10}$$

Now, if P_0 is allowed to go to zero, the volume of the gas will go to infinity, making it impossible to calculate the absolute fugacity of a substance. To deal with this issue, the fugacity coefficient, ϕ, is introduced, such that by Equation (1.9), ϕ goes to 1 as P goes to 0 and,

$$\phi = \frac{f}{P}. \tag{1.11}$$

From Equation (1.8), it is clear that if the chemical potentials of two phases are equal, the fugacities will also be equal, and, similarly, the fugacity coefficients will be equal. Thus, the thermodynamic relationship that defines equilibrium between phases $I, II, III, \ldots$ in Equation (1.2) can be rewritten in terms of fugacity coefficients as:

$$\phi^{I}\left(T,P\right) = \phi^{II}\left(T,P\right) = \phi^{III}\left(T,P\right) = \cdots \tag{1.12}$$

This is the primary condition of equilibrium used in the numerical routine presented in this work. As such, an explicit formulation of the fugacity coefficient, in terms of easily measurable substance properties, is desired. Following the derivation by Walas, this expression is developed by first considering Equation (1.10), which is transformed into a volume integral for convenience [8]:

$$\begin{aligned} RT \ln\left(\frac{f}{f_0}\right) &= \int_{P_0}^{P} V_m dP \\ &= \int_{P_0 V_0}^{P V_m} d\left(P V_m\right) - \int_{V_0}^{V_m} P dV_m \\ &= P V_m - P_0 V_0 - \int_{V_0}^{V_m} P dV_m. \end{aligned} \tag{1.13}$$

Now, an expression for the fugacity coefficient can be derived using Equation (1.11) and the fact that, by the definition of the fugacity, as P goes to 0, $f \to P$ and $f_0 \to P_0$:

$$\begin{aligned} RT\ln\phi &= RT\ln\frac{f}{P} \\ &= RT\ln\left(\frac{f}{f_0}\frac{f_0}{P}\right) \\ &= RT\ln\frac{f}{f_0} - RT\ln\frac{P}{P_0}. \end{aligned} \tag{1.14}$$

At this point, the fugacity coefficient can be further developed in terms of integration over pressure or volume using either Equation (1.10) or Equation (1.13), respectively. Both of these forms are shown as they are useful for different aspects of this analysis. First, substituting Equation (1.10) into Equation (1.14) yields:

$$\begin{aligned} RT\ln\phi &= \int_{P_0}^{P} V_m dP - RT\ln\frac{P}{P_0} \\ &= \int_{P_0}^{P}\left(V_m - \frac{RT}{P}\right)dP. \end{aligned} \tag{1.15}$$

Using the definition of compressibility,

$$Z = \frac{PV_m}{RT}, \tag{1.16}$$

and choosing $P_0 = 0$, this can be rewritten as:

$$\ln\phi = \int_0^P \frac{Z-1}{P}dP. \tag{1.17}$$

From this form, it becomes evident that the fugacity coefficient is just a representation of the deviation a substance takes from an ideal gas. When the substance of interest behaves as an ideal gas, either because $Z = 1$ or as $P \to 0$, the requirement that $f \to P$, and therefore $\phi \to 1$, holds.

Now, returning to Equation (1.14) and using Equation (1.13), the fugacity coefficient can also be expressed as follows:

$$RT\ln\phi = PV_m - P_0V_0 - \int_{V_0}^{V_m} PdV_m - RT\ln\frac{P}{P_0}. \tag{1.18}$$

By adding and subtracting $\frac{RT}{V_m}$ inside the integrand, this can be rewritten as:

$$RT \ln \phi = PV_m - P_0V_0 - \int_{V_0}^{V_m} \left(P - \frac{RT}{V_m}\right) dV_m - \int_{V_0}^{V_m} \left(\frac{RT}{V_m}\right) dV_m - RT \ln \frac{P}{P_0}. \tag{1.19}$$

Integrating:

$$\begin{aligned} RT \ln \phi &= PV_m - P_0V_0 - \int_{V_0}^{V_m} \left(P - \frac{RT}{V_m}\right) dV_m - RT \ln \frac{V_m}{V_0} - RT \ln \frac{P}{P_0} \\ &= PV_m - P_0V_0 - \int_{V_0}^{V_m} \left(P - \frac{RT}{V_m}\right) dV_m - RT \ln \left(\frac{V_m P}{RT} \frac{RT}{V_0 P_0}\right) \\ &= PV_m - P_0V_0 - \int_{V_0}^{V_m} \left(P - \frac{RT}{V_m}\right) dV_m - RT \ln \frac{V_m P}{RT} - RT \ln \frac{RT}{V_0 P_0} \\ &= PV_m - P_0V_0 - \int_{V_0}^{V_m} \left(P - \frac{RT}{V_m}\right) dV_m - RT \ln Z - RT \ln Z_0. \end{aligned} \tag{1.20}$$

If the reference state is taken such that $P \to 0$, it can be assumed that $V_m \to \infty$, $Z_0 \to 1$, and the ideal gas law holds such that $P_0V_0 \to RT$, leading to:

$$RT \ln \phi = PV_m - RT - \int_{\infty}^{V_m} \left(P - \frac{RT}{V_m}\right) dV_m - RT \ln Z. \tag{1.21}$$

Finally, dividing through by RT and reorganizing leads to the final formulation of the fugacity coefficient of a pure substance, shown in Equation (1.22):

$$\ln \phi = \frac{1}{RT} \int_{V_m}^{\infty} \left(P - \frac{RT}{V_m}\right) dV_m - \ln Z + Z - 1. \tag{1.22}$$

It can be seen now that a relationship between the pressure, volume, and temperature is required to determine the fugacity coefficient. This relationship, also referred to as an equation of state, is described in the following section.

1.4 EQUATIONS OF STATE

An equation of state is a fundamental thermodynamic correlation which relates thermodynamic properties, and fully defines a system. A common form of an equation of state includes the absolute temperature, the pressure, and the molar volume. The most basic equation of state is the ideal gas law, referenced before and shown in Equation (1.23):

$$PV_m = RT, \tag{1.23}$$

where P is the pressure, V_m is the molar volume, which is equivalent to the total volume divided by the number of moles ($\frac{V_t}{n}$), R is the molar universal gas constant, and T is the temperature. This equation can also be represented using the compressibility Z, defined previously as:

$$Z = \frac{PV_m}{RT}, \tag{1.24}$$

which, for an ideal gas, leads to:

$$Z = 1. \tag{1.25}$$

This form of the ideal gas law highlights the primary downside of this simple equation. Based on the assumption that all molecules are incompressible hard-spheres, the ideal gas law predicts a constant compressibility of 1 for all substances, regardless of composition and molecular interactions. This inherent limitation has led to the development of equations of state that seek to include interaction contributions and therefore achieve more realistic results.

There are two main categories of equations of state that are used for modeling, those with a theoretical basis and those formulated using empirical data. As shown in Figure 7, the empirically-based equations are primarily derived from the van der Waals equation, (VDW), while the theoretical forms are based on various molecular and statistical theories [9]. Each of these groups then branches into a number of different forms of equations, based on the specific empirical methods or theories used to develop each one.

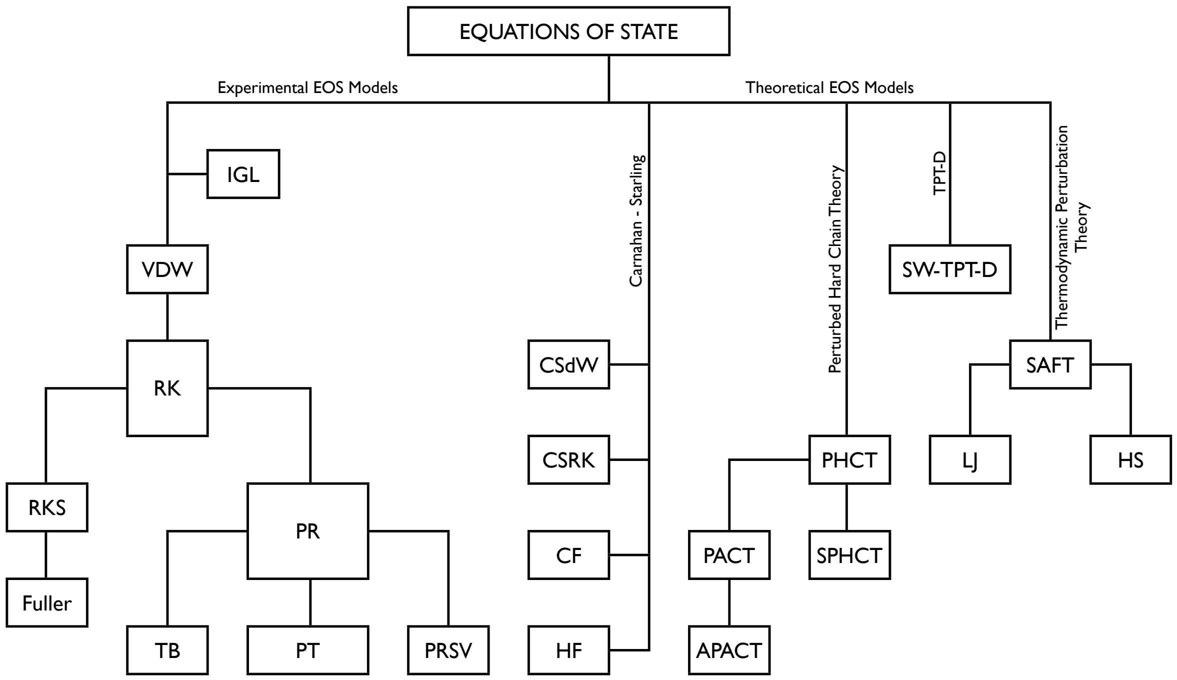


Figure 7: A chart displaying the genealogy of various equations of state, represented by their common abbreviations. Adapted from Wei & Sadus [9]

1.4.1 Theoretical Equations of State

Theoretical equations are based on statistical thermodynamics, which draws from an understanding of molecular dynamics. While potentially more accurate over larger ranges of conditions, these equations are very difficult to solve and are therefore extremely computationally expensive.

The first theoretical equation of state was the virial equation of state. When it was originally proposed in 1885 by Thiesen, its justification was entirely empirical. However, later work showed that the form of the equation was consistent with statistical mechanics. If the density form of the ideal gas law is considered [10–12]:

$$\frac{P}{\rho_n RT} = 1, \tag{1.26}$$

where $R = N_A k_B$, $\rho_n = N/(N_A V)$, and N_A is Avogadro's number, the virial equation is the natural next step, as it is simply a Maclaurin series expanded around $\rho_n = 0$, as shown in Equation (1.27):

$$\frac{P}{\rho_n RT} = 1 + B\rho_n + C\rho_n^2 + D\rho_n^3 + \cdots . \tag{1.27}$$

The coefficients in the equation are dependent on temperature and composition and can be directly related to intermolecular potential energy functions. For example, the square-well potential function, shown in Figure 8, has a fairly simply mathematical form:

$$\Gamma = \begin{cases} \infty & \text{if } r \leq \sigma, \\ -\epsilon & \text{if } \sigma < r \leq R\sigma, \\ 0 & \text{if } r > R\sigma, \end{cases} \tag{1.28}$$

where ϵ is the depth of the energy well (minimum potential energy), σ is the molecular diameter, R is the reduced well width, and r is the distance between two molecules. This function can be used to write the second virial coefficient, B, as [13, 14]:

$$B = b_0 R^3 \left(1 - \frac{R^3 - 1}{R^3} \exp \frac{\epsilon}{RT}\right). \tag{1.29}$$

However, determining the higher order coefficients becomes exceedingly difficult, and unfortunately, the equation only converges for low densities, making it unusable for liquid calculations or temperatures and pressures near the critical point [12].

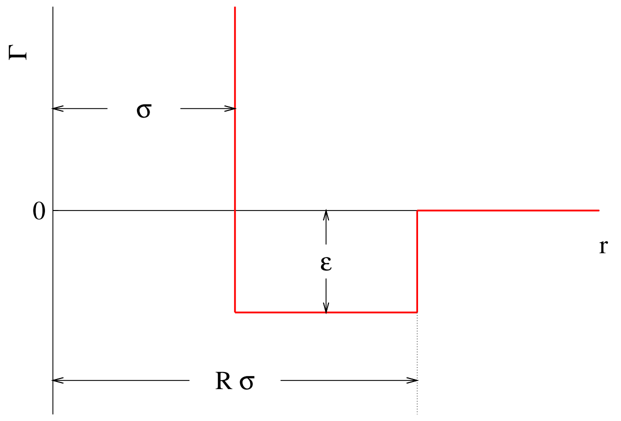


Figure 8: The square-well molecular potential function.

Although the development of new theoretical equations has become very complicated, some insight into their formation can be gained by considering the statistical thermodynamics used as their basis. This approach begins with the fact that the energy of every molecule of a substance can be divided into various forms. An ideal gas, for example, is characterized by non-interacting hard-spheres. As a result, the only energy that is considered for an ideal gas is the translational energy allowed by the individual molecules that comprise it. For a real substance, however, the rotational, vibrational, and potential energies of each molecule must also be considered [8, 15]. The amalgamation of these energies is formulated using the molecular partition function, which is defined as:

$$Q = \sum_i g_i \exp\left(\frac{-\varepsilon_i}{k_B T}\right), \tag{1.30}$$

where ε_i is the quantized translational, rotational, vibrational, or potential energy, g_i is the number of quantized states with that energy, and k_B is the Boltzmann constant. Using this function, many thermodynamic properties can be expressed. The internal energy, entropy, Gibbs free energy, and chemical potential are shown in Equations (1.31) - (1.34), respectively [8, 15]:

$$\begin{aligned} U &= \sum_i N_i \varepsilon_i \\ &= RT^2 \left(\frac{\partial \ln Q}{\partial T}\right)_V , \end{aligned} \tag{1.31}$$

$$S = R\left[T\left(\frac{\partial \ln Q}{\partial T}\right)_V + \ln Q\right], \tag{1.32}$$

$$G = RT\left[V\left(\frac{\partial \ln Q}{\partial V}\right)_T - \ln Q\right], \tag{1.33}$$

$$\mu_i = -RT\left(\frac{\partial \ln Q}{\partial n_i}\right)_{T,V,n_j} . \tag{1.34}$$

Theoretical equations of state seek to express the interactions between molecules for various substances and mixtures by defining the different energies associated with different molecules. It is important to note that, as more detail is used to describe these molecular interactions, more complexity enters the equations. Excellent examples of theoretical equations are the Associated Perturbed Anisotropic Chain Theory (APACT) and the Statistical Associating Fluid Theory (SAFT) equations of state, described briefly here.

The APACT equation, developed by Ikonomou and Donohue, is written as the sum of compressibility factors that account for isotropic repulsive and attractive interactions that are independent of association, and anisotropic interactions that arise from dipole and quadrupole moments, as well as hydrogen bonding. The APACT equation is shown in its most general form in Equation (1.35) [9, 16, 17]. Each of its terms has been further developed by Ikonomou, Donohue, Economou, and Vilmalchand [9, 16, 18–22].

$$Z = 1 + Z^{rep} + Z^{att} + Z^{assoc} \tag{1.35}$$

The SAFT equation, based on the sum of four Helmholtz energy terms that account for hard-sphere repulsive forces, dispersion forces, chain formation, and association, was proposed by Chapman et al. and developed by Huang and Radosz. Its general form is shown in Equation (1.36) [9, 23–25]:

$$\frac{A}{NkT} = \frac{A^{ideal}}{NkT} + \frac{A^{seg}}{NkT} + \frac{A^{chain}}{NkT} + \frac{A^{assoc}}{NkT}. \tag{1.36}$$

More detail concerning the individual terms of these equations can be found in the references given above. Additionally, an excellent overview of these equations, as well their many modifications, is given by Wei and Sadus [9].

1.4.2 Empirical Equations of State

Empirical equations, compared to theoretical ones, are relatively easy to solve numerically, typically requiring simple iterative and root-finding procedures. Unfortunately, empirical equations are not generally applicable for all fluids, in all conditions. That said, many cubic equations of state have been developed by fitting equation parameters to experimental data that are capable of reproducing pure fluid properties over a significantly large number of fluids in a variety of conditions. In fact, at low pressures, empirical equations can provide more accurate data reproduction than theoretical equations [9]. The vast majority of empirical equations of state are based on modifications to the van der Waals equation. Originally proposed in 1873, and shown in Equation (1.37), the van der Waals equation was the first equation of state with the ability to calculate the simultaneous occurrence of both liquid and vapor phases in a state of equilibrium [9].

$$Z = Z^{rep} + Z^{att} \tag{1.37}$$

It is interesting to note that, superficially, this equation is similar to the theoretical APACT equation shown in Equation 1.35. However, instead of segregating the association independent repulsive and attractive terms and the association term, as in the APACT equation, the van der Waals equation only includes one repulsive and one attractive term. Z^{rep} and Z^{att} take the following forms for the vdW equation of state:

$$Z^{rep} = \frac{V_m}{V_m - b}, \tag{1.38}$$

$$Z^{att} = -\frac{a}{RTV_m}. \tag{1.39}$$

In these relations, the b parameter represents the covolume, defined such that if the molecules were hard-spheres with a diameter σ, b would be equal to $\frac{2}{3}\pi N\sigma^3$. The a parameter, on the other hand, represents the attractive forces between molecules [9]. Thus, the van der Waals equation is comprised of two terms, the first is responsible for the repulsion between molecules due largely to their spatial requirements, and the second represents the attraction between the molecules. It is important to remember though, that these are inferences applied to what are, in reality, curve-fitting parameters whose values have been related to physical properties. Their forms are derived by considering the constraints imposed by the shape of the critical isotherm of a pure substance, shown in Figure 9 [26]. This requires that the critical isotherm have a horizontal inflection point at the critical state, implying:

$$\left(\frac{\partial P}{\partial V}\right)_{T;cr} = 0, \tag{1.40}$$

$$\left(\frac{\partial^2 P}{\partial V^2}\right)_{T;cr} = 0. \tag{1.41}$$

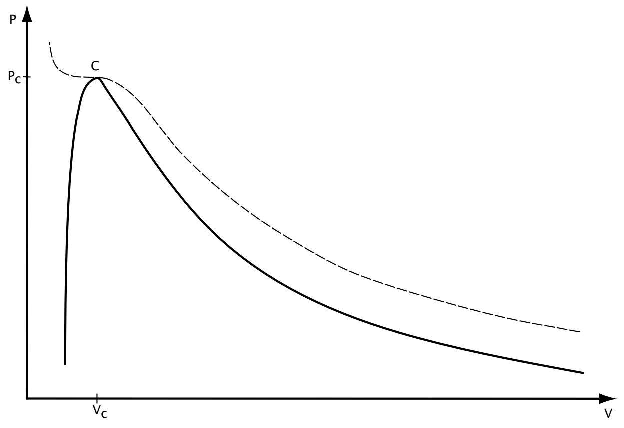


Figure 9: Critical isotherm of a pure substance. Adapted from *Schaum's Thermodynamics Outline* by Abbott & Van Ness [26].

More detail concerning the practical calculation of the a and b terms for mixtures will be given in Chapter 2, but it is these parameters that incorporate information about the specific substances being modeled. For example, in order to calculate the parameters for a pure substance for the van der Waals equation of state, Equations (1.40) and (1.41) are applied to the equation of state at the critical point and the following expressions are obtained:

$$a = \frac{27}{64}\frac{(RT_c)^2}{P_c}, \tag{1.42}$$

$$b = \frac{1}{8}\frac{RT_c}{P_c}, \tag{1.43}$$

where T_c and P_c are the critical temperature and critical pressure of the substance, respectively.

While the van der Waals equation is capable of predicting the coexistence of the liquid and vapor phases, it is plagued by inaccuracy. For example, regardless of the chosen fluid, the van der Waals equation calculates the same critical compressibility factor ($Z_c = \frac{P_c V_{m,c}}{RT_c}$) of 0.375. This result, while much better than the $Z = 1$ calculated by the ideal gas equation, is still erroneous. Consequently, a great deal of work has gone into modifying the functional form of the equation, with the majority of time being spent on the attractive term. This includes attempts to incorporate varying degrees of volumetric and temperature dependence, as well as further dependence on the covolume parameter, and even the introduction of additional fitting parameters. While a, b, and other parameters remain calculable from the critical properties of fluids, the functional form of these expressions has also undergone a good deal of manipulation and actual values can vary greatly. The result from the last 140 years of work, much of which as been carried out in the last 60, is an agglomeration of equations, all with variations on the basic form proposed by van der Waals. Each equation falls into a different category based on which fluids it can successfully model and at which temperature and pressure ranges. Some of the most common modifications are shown in Table 1 [9]. The various parameters shown in these equations have been added to increase the flexibility of the equations. For example, the α parameter was introduced first in the Redlich-Kwong equation of state to add a temperature dependence to the attractive term in the following manner:

$$Z = \frac{V_m}{V_m - b} - \frac{a\alpha}{RT\,(V_m + b)} \tag{1.44}$$

$$= \frac{V_m}{V_m - b} - \frac{a}{RT^{1.5}\,(V_m + b)}. \tag{1.45}$$

This concept was taken a step further by Soave, with his introduction of a more general expression for α, including both a temperature dependence and additional fluid properties, such that:

$$a\alpha = 0.4274\left(\frac{R^2T_c^2}{P_c^2}\right)\left(1+m\left[1-\left(\frac{T}{T_c}\right)^{0.5}\right]\right)^2, \tag{1.46}$$

with:

$$m = 0.480 + 1.57\omega - 0.176\omega^2, \tag{1.47}$$

where ω is the acentric factor of the fluid.

The c parameter in the Patel-Teja equation of state and the c and d parameters of the Trebble-Bishnoi equation, further increase the ability of the equations to reproduce real fluid behavior. In a two-parameter equation of state, such as the van der Waals equation, the critical compressibility and b parameter are constant for all substances. Introducing a third parameter allows the variability of one of those values, and a fourth makes both variable, potentially allowing more accurate modeling of a wider range of substances [27–29].

Some equations of state are better for pure fluid property prediction, while others excel at fluid mixtures with specific interaction properties. It must be remembered however, that all of these equations, no matter how intricate, are still empirical relations. In the end, fitting parameters are required in order to correlate the expressions to experimental data sets. Often, without decent parameters, the equations fail not only to predict data that is accurate, but can even calculate behavior that is completely unphysical. This highlights an important aspect of empirical equations of state: they are by nature correlative, not predictive. It is necessary to have decent *a priori* knowledge of empirical work before the calculation of data is possible. This is not to say these equations are not extremely useful. It has been found that by determining fitting parameters using a relatively small range of data, much larger ranges can be predicted accurately, using the same parameters. Similarly, by experimentally determining parameters for one fluid pair, it is often possible to use similar parameters for fluids with similar properties and interactions. However, it is often difficult to decide which equations should be used at which times, and to what extent existing parameters can be used to predictively calculate missing data. It is therefore important, if the implementation of these equations is desired, to be aware of the true limitations of their usability. Not only will this allow a better insight of the working ranges of the equations, but it will also assist in the selection of equations based on the availability of good empirical data. This need provides the primary motivation for this work.

The following chapter will discuss modifications to the fugacity coefficient equation and the equations of state that were presented above, which are necessary for their application to the calculation of the vapor-liquid equilibrium of mixtures. This will be followed by an overview of the procedure used to determine vapor-liquid equilibria data using these equations. Finally, the behavior of the equations will be analyzed and an enhanced method for the visualization of performance, from both a correlative and a predictive perspective, will be introduced.

Table 1: Modifications to the Attractive Term of the van der Waals Equation [9]

Equation	Attractive Term ($-Z^{att}$)
Redlich-Kwong (1949)	$\frac{a\alpha}{RT(V_m+b)}$
Redlich-Kwong-Soave (1972)	$\frac{a\alpha}{RT(V_m+b)}$
Peng-Robinson (1976)	$\frac{a\alpha V_m}{RT[V_m(V_m+b)+b(V_m-b)]}$
Patel-Teja (1982)	$\frac{a\alpha V_m}{RT[V_m(V_m+b)+c(V_m-b)]}$
Peng-Robinson-Stryjek-Vera (1986)	$\frac{a\alpha V_m}{RT\left(V_m^2+2bV_m-b^2\right)}$
Trebble-Bishnoi (1987)	$\frac{a\alpha V_m}{RT\left[V_m^2+(b+c)V_m-\left(bc+d^2\right)\right]}$

2.0 APPLICATION OF EQUATIONS TO MIXTURES

The previous chapter provided an overview of fugacity and equations of state. However, little was said about the manner in which these equations can be applied to the calculation of mixture properties. The following sections will describe the methods used to accomplish this procedure. Recall that Equation (1.22), which is repeated here for convenience, gives the expression for the fugacity coefficient of a pure substance:

$$\ln \phi = \frac{1}{RT} \int_{V_m}^{\infty} \left(P - \frac{RT}{V_m} \right) dV_m - \ln Z + Z - 1. \tag{2.1}$$

From Hu, et al., the relationship between the partial and the pure fugacity coefficients can be written as [30]:

$$\ln \hat{\phi}_i = \left(\frac{\partial \left(n \ln \phi \right)}{\partial n_i} \right)_{T,P,n_{j \neq i},V_t}. \tag{2.2}$$

Combining Equations (2.1) and (2.2) gives:

$$\ln \hat{\phi}_i = \frac{\partial}{\partial n_i} \left[nZ - n - n \ln Z \right] - \frac{1}{RT} \left(\frac{\partial}{\partial n_i} \left[\int_{\infty}^{V_m} \left(nP - \frac{nRT}{V_m} \right) d \left(V_m \right) \right] \right), \tag{2.3}$$

where the constants of the partial derivatives are assumed but omitted for simplicity. Next, using the definition that:

$$n = \sum_i n_i, \tag{2.4}$$

such that:

$$\frac{\partial n}{\partial n_i} = 1, \tag{2.5}$$

and $V_m = V_t/n$, each term can be evaluated as follows:

$$\frac{\partial \left(nZ \right)}{\partial n_i} = \frac{\partial}{\partial n_i} \left[n \frac{PV_m}{RT} \right] = \frac{\partial}{\partial n_i} \left[\frac{PV_t}{RT} \right] = 0, \tag{2.6}$$

$$
\begin{aligned}
\frac{\partial\left(n \ln Z\right)}{\partial n_i} &= \frac{\partial n}{\partial n_i}\ln Z + n\frac{\partial \ln Z}{\partial n_i} \\
&= \ln Z + \frac{n}{Z}\frac{\partial Z}{\partial n_i} \\
&= \ln Z + \frac{n^2 RT}{PV_t}\frac{\partial}{\partial n_i}\left[\frac{PV_t}{nRT}\right] \\
&= \ln Z + \frac{n^2 RT}{PV_t}\frac{PV_t}{RT}\left(-\frac{1}{n^2}\right) \\
&= \ln Z - 1, \qquad (2.7)
\end{aligned}
$$

$$
\begin{aligned}
\frac{1}{RT}\frac{\partial}{\partial n_i}\left[\int_\infty^{V_m}\left(nP - \frac{nRT}{V_m}\right)dV_m\right] &= \frac{1}{RT}\frac{\partial}{\partial n_i}\left[\int_\infty^{V_t/n}\left(nP - \frac{n^2RT}{V_t}\right)d\left(\frac{V_t}{n}\right)\right] \\
&= \frac{1}{RT}\frac{\partial}{\partial n_i}\left[\int_\infty^{V_t}\left(nP - \frac{n^2RT}{V_t}\right)\frac{1}{n}dV_t\right] \\
&+ \frac{1}{RT}\frac{\partial}{\partial n_i}\left[\int_\alpha^{\beta}\left(nPV_t - n^2RT\right)d\left(\frac{1}{n}\right)\right] \\
&= \frac{1}{RT}\int_\infty^{V_t}\left[\left(\frac{\partial P}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} - \frac{RT}{V_t}\right]dV_t. \qquad (2.8)
\end{aligned}
$$

The α and β in one of the integrals above represent arbitrary limits of integration which are not important because the integration over $\frac{1}{n}$ will yield an expression independent of n_i, causing the derivative of this term to vanish. Applying the terms evaluated above to Equation (2.3), the partial fugacity coefficient becomes:

$$
RT\ln\hat{\phi}_i = \int_{V_t}^{\infty}\left[\left(\frac{\partial P}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} - \frac{RT}{V_t}\right]dV_t - RT\ln Z, \qquad (2.9)
$$

where the pressure and the compressibility factor are determined using equations of state that describe mixture properties, instead of pure component properties.

In Chapter 1, Section 1.4.2, many equations of state were shown. As was briefly mentioned then, the substance properties only affect the parameters in the equations, e.g. the a and b parameters in the Peng-Robinson equation of state. As a result, the equations shown there keep the same form for both pure species and mixtures, but the parameters that comprise them vary to incorporate the effects of interactions between different substances. The ways in which these interactions are taken into account are described by mixing and combining rules.

2.1 MIXING RULES

As shown for the van der Waals equation of state in Chapter 1, Section 1.4.2, the a and b parameters are dependent on the critical temperature and critical pressure of the substance of interest. The pure forms of the parameters for select equations of state will be presented later, in Chapter 3. However, regardless of the specific form of these parameters for different equations of state, in order to apply them to mixtures, a set of mixing rules must be used. Mixing rules primarily include the effect of mixture composition on the value of the parameters. This is accomplished by multiplying the pure parameters by either the liquid or vapor mole fractions and combining them in a way that sets the mixture parameter to an intermediate value between the pure parameter values. The choice of which mole fraction to use will depend on whether bubble point or dew point data is desired. To clarify this, consider the total mole fraction, in any phase, of a component i in a mixture:

$$z_i = x_i * LF + y_i * VF, \tag{2.10}$$

where x_i is the liquid mole fraction of component i in the mixture, y_i is the vapor mole fraction of component i in the mixture, LF is the total liquid fraction of the mixture, and VF is the total vapor fraction of the mixture. Now, for a bubble point calculation, the mixture is in a saturated liquid state, and as a result, $LF = 1$ and $VF = 0$. For a dew point calculation on the other hand, the mixture is saturated vapor so that $LF = 0$ and $VF = 1$. Therefore, we can write the total mole fraction of component i in a mixture at its bubble point as:

$$z_i = x_i, \tag{2.11}$$

and at its dew point as:

$$z_i = y_i. \tag{2.12}$$

In this work, the mole fraction will just be referred to as z_i, with the understanding that this refers to the liquid mole fraction of component i for the bubble point, and the vapor mole fraction for the dew point.

Now, using this mole fraction, the mixing rules can be defined. There are two primary forms of mixing rules that are implemented in this work: linear and quadratic. These are shown in Equations (2.13) and (2.14), respectively, for an arbitrary parameter, ζ:

$$\zeta = \sum_{i}^{N} z_i \zeta_{ii}, \tag{2.13}$$

$$\zeta = \sum_{i}^{N} \sum_{j}^{N} z_i z_j \zeta_{ij}, \tag{2.14}$$

where N is the total number of components in the mixture. In these expression, the ζ_{ii} terms are the pure forms of the parameters, such as would be used for the calculation of property data for a single, pure component. The ζ_{ij}s, on the other hand, will depend on mixture behavior and can be determined through the use of combining rules, which will be described in the following section. It is important to note that there are a vast number of mixing rules available and many different techniques have been used in their determination. Methods such as lumping and spectral decomposition have found particular success in the modeling of systems with large numbers of components to reduce the order or the system to be solved [31–33]. However, this work focuses on the two most popular mixing rules in standard vapor-liquid equilibrium calculations.

For binary mixtures, the linear mixing rule becomes:

$$\zeta = z_1 \zeta_{11} + z_2 \zeta_{22}, \tag{2.15}$$

and the quadratic mixing rule is:

$$\zeta = z_1^2 \zeta_{11} + z_1 z_2 \zeta_{12} + z_1 z_2 \zeta_{21} + z_2^2 \zeta_{22}. \tag{2.16}$$

Notice that the linear mixing rule does not contain a ζ_{ij} term. As a result, a parameter calculated using the linear mixing rule will not depend on a combining rule.

While the linear mixing rule is clearly just a linear combination of the two pure parameters, the quadratic mixing rule can be interpreted physically through a consideration of the probability that two molecules will interact. This is best illustrated through an example. Consider a system with only three molecules, two of type 1, and one of type 2. In this system, the probability that one will encounter a molecule of type 1 is 2 out of 3. This quantity is described by its mole fraction, z_1. Likewise, the probability of encountering a molecule of type 2 is 1 out of 3, or z_2. Following this logic, the conditional probability that a molecule of type 1 will interact with another molecule of type 1 is described by the product of the two probabilities, $z_1 * z_1$, or $\frac{4}{9}$. Similarly, the probability that a molecule of type 1 interacts with a molecule of type 2 becomes $z_1 * z_2$, or $\frac{2}{9}$. Thus, the quadratic mixing rule seeks to formulate a mixture parameter based on the probabilities of molecular interactions [34]. The importance of this will be revealed when the forms of ζ_{12} and ζ_{21} are considered in the following section.

2.2 COMBINING RULES

Combining rules are the source of much complexity in determining the properties of mixtures. While there is no quantitative, theoretical basis for combining rules, the idea behind their forms stems from intermolecular potential theory. When molecules are near each other, they have some potential energy that either pulls them closer together or pushes them farther apart. That potential depends of the distance between the molecules and is often expressed as a potential energy function, Γ. In addition to the square-well potential shown in Chapter 1, Section 1.4.1, another common potential function is the Lennard-Jones potential, shown in Figure 10 and described by Equation (2.17) [12, 35, 36]:

$$\Gamma(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{2.17}$$

where, as before, σ represents the molecular diameter, r is the distance between the two interacting molecules, and ϵ is the depth of the energy well. However, this potential describes the energy between two identical molecules. Commonly, the potential between two unlike molecules is written as:

$$\Gamma_{ij} = \epsilon_{ij} F\left(\frac{r_{ij}}{\sigma_{ij}}\right), \tag{2.18}$$

where F represents the chosen potential form, such as the Lennard-Jones potential. Now, expressions for the diameter and energy parameters, σ_{ij} and ϵ_{ij} , for two different molecules are required. For the diameter, a common choice is the Lorentz rule, shown in Equation

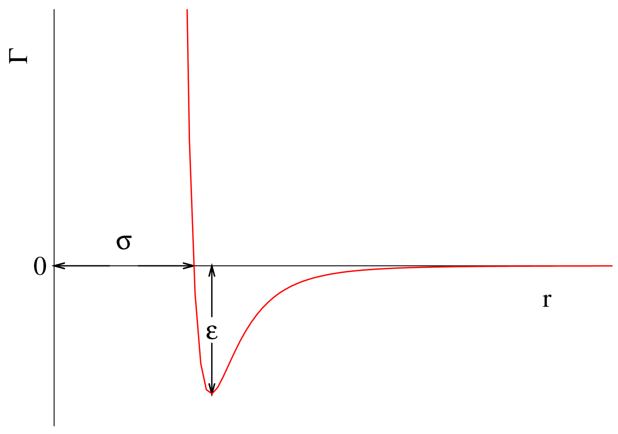


Figure 10: The Lennard-Jones potential function.

(2.19). This would be an exact expression if the molecules were actually hard spheres, following a square-well potential, in which repulsive forces are only significant when the molecules come into contact, but it is often used as an approximation nonetheless [12, 37].

$$\sigma_{12} = \frac{1}{2}\left(\sigma_{11} + \sigma_{22}\right) \tag{2.19}$$

The energy, on the other hand, is often developed by considering a simple geometric combination between energies, leading to its form as the Berthelot rule:

$$\epsilon_{12} = \left(\epsilon_{11}\epsilon_{22}\right)^{1/2}. \tag{2.20}$$

There are many different combining rules for the potential energy function parameters, but these basic expressions are formulated using a simplified understanding of the manner in which lengths and energies combine. With this background, an analogy between the parameters of the potential function and those in an equation of state can be drawn. The energy, ϵ, and the a parameter, both seek to describe the attractive energy between molecules. Similarly, the molecular diameter, σ, and the b parameter, are measures of a molecules size, assuming it occupies the space of a hard sphere. These interpretations of the parameters in equations of state allow the application of the same combining rules as follows [35]:

$$a_{ij} = (1 - k_{ij}) \sqrt{a_{ii} a_{jj}}, \tag{2.21}$$

$$b_{ij} = \frac{1}{2} (1 - l_{ij}) (b_{ii} + b_{jj}), \tag{2.22}$$

where k_{ij} and l_{ij} represent additional parameters added to fit the simulation to experimental data. It can now be seen why a consideration of the probabilities of molecular interactions is useful. The combining rules seek to describe an equation of state parameter for a mixture based on how the pure component parameters might combine. However, this combination should only affect the mixture parameter when the molecules are actually in contact. In other words, implementing a quadratic mixing rule with a combining rule results in the contribution of the cross-interaction terms being scaled based on the probability that the molecules involved are in fact interacting.

As with the development of equations of state, while the combining rules presented above are qualitatively justified, they are empirically based, and therefore can be improved in various ways. A vast number of different combining rules exists, from the simple, generic forms shown above, to forms which are highly specialized for a small subset of mixtures. Only a few will be presented in this work, with an attempt to include both simple and somewhat complex rules. Those rules, shown in Equations (2.23) - (2.26), are the Arithmetic, the Geometric, the Margules, and the van Laar combining rules, respectively, which, for binary mixtures, should be written for $i, j \in \{1, 2\} : i \neq j$ where k_{ij} and k_{ji}, referred to as binary interaction parameters, are experimental fitting parameters that can take on different values for different parameters. Equations (2.21) and (2.22) are rewritten here as well as combining rules for a generic parameter, ζ.

$$\zeta_{ij} = \frac{1}{2} (1 - k_{ij}) (\zeta_{ii} + \zeta_{jj}) \tag{2.23}$$

$$\zeta_{ij} = (1 - k_{ij}) \sqrt{\zeta_{ii} \zeta_{jj}} \tag{2.24}$$

$$\zeta_{ij} = (1 - z_i k_{ij} - z_j k_{ji}) \sqrt{\zeta_{ii} \zeta_{jj}} \tag{2.25}$$

$$\zeta_{ij} = \left(1 - \frac{k_{ij} k_{ji}}{z_i k_{ij} + z_j k_{ji}}\right) \sqrt{\zeta_{ii} \zeta_{jj}} \tag{2.26}$$

Considering the probability argument presented in the previous section, the last two combining rules shown can be seen to introduce further composition dependence. Physically, this refers to the possibility that molecules in a mixture may cluster, rather than distribute randomly, depending on their particular nature. Therefore, the more complex combining rules seek to better explain the physical phenomena that occur on a molecular level [34]. However, examining the expression for the van Laar combining rule, it becomes evident why some more specialized combining rules might not always be ideal to use. Because there is no rule governing the signs of k_{ij} and k_{ji}, it is possible that $z_i k_{ij} + z_j k_{ji} \to 0$. When this occurs, the entire numerical procedure breaks down due to the existence of a pole. Methods used to address this divergence are discussed in Section 4.4.1.

A significant aspect of these combining rules, and thus of equations of state for mixtures, are the binary interaction parameters. These are the parameters that act as experimental fitting values to improve equation accuracy. These values must be chosen separately for each fluid mixture combination, requiring experimental data of the individual mixture, and making the equations correlative, as discussed in Chapter 1. As will be shown, these parameters can have a profound effect on the accuracy of the final simulated data. Consequently, the dependence of an equation's accuracy on the availability of these parameters is a very interesting complication that arises in vapor-liquid equilibrium modeling. A study of this behavior will yield the desired analysis of the predictability and usability of equations of state that is so important in the practical implementation of VLE work.

3.0 MATHEMATICAL DERIVATIONS

The detailed mathematical derivations presented in this chapter are important aspects of the vapor-liquid equilibrium calculation process. Typically, the equations presented in many treatments of this method are only shown in a reduced form, with specific mixing rules and combining rules already implemented into the equations of state. This lack of generality makes it difficult for one to carry out VLE calculations with varying combinations of equations, removing one's ability to analyze the effect these combinations have on performance. Therefore, the mathematical derivations that follow are shown in full detail to allow the aggregation of a diverse range of equations of state, mixing rules, and combining rules.

The general form of the fugacity coefficient for each component in a mixture, shown in Equation (2.9), is repeated here for reference:

$$RT \ln \hat{\phi}_i = \int_{V_t}^{\infty} \left[\left(\frac{\partial P}{\partial n_i} \right)_{T,V_t,n_{j\neq i}} - \frac{RT}{V_t} \right] dV_t - RT \ln Z \tag{3.1}$$

This expression requires two variations of an equation of state to solve. First, a pressure explicit form is needed for the calculation of the partial derivative. Second, it is convenient to have a form of the equation of state in terms of compressibility so that its solution yields a value for Z. Thus, general formulations of both of these types are given in Equations (3.2) and (3.4). Each of the following sections then develops these equations to include the specifics of a few equations of state. It should be noted that all the parameters affected by mixing and combining rules are potentially composition dependent, and therefore, taking the partial derivative shown above is not trivial.

The general form of a pressure explicit cubic equation of state is:

$$P = \frac{RT}{V_m - b} - \frac{\theta \left(V_m - \eta \right)}{\left(V_m - b \right) \left(V_m^2 + \delta V_m + \varepsilon \right)}, \tag{3.2}$$

where θ, η, δ, and ε will be defined for each equation of state. Solving the definition of compressibility for volume gives:

$$V_m = \frac{ZRT}{P}, \tag{3.3}$$

which can be substituted into Equation (3.2), and, after some simplification, a general cubic equation of state in terms of compressibility is derived:

$$\begin{aligned} Z^3 &+ \left[\frac{P}{RT}(\delta - b) - 1\right] Z^2 + \left[\left(\frac{P}{RT}\right)^2 (\varepsilon - \delta b) - \left(\frac{P}{RT}\right)\delta + \left(\frac{P}{(RT)^2}\right)\theta\right] Z \\ &+ \left[\left(\frac{P}{RT}\right)^2 \left(-\varepsilon b \frac{P}{RT} - \varepsilon - \eta\theta\frac{1}{RT}\right)\right] = 0. \end{aligned} \tag{3.4}$$

Note that this equation will provide 3 roots, due to its cubic nature. Three possibilities exist for these roots if all are real: three distinct roots, one distinct root and a double root, or one triple root, labeled by P_{r1}, P_{r2}, and P_{r3}, respectively, in Figure 11 [26]. A triple root exists only on the critical isotherm, at $P_r = 1$ and $T_r = 1$, and corresponds to the critical point of the fluid. The other two cases can exist for a number of different P_r values. However, their stability is limited by Maxwell's equal-area rule, which claims that, at an equilibrium state, the horizontal line drawn at a constant P_r must intersect the isotherm in such a way that the areas between the isotherm and the line are equal. Physically, this requirement can be interpreted by recognizing that the mechanical work done on or by a system is equal to the area under an isotherm on a pressure-volume curve. If path CDBFA is assumed to be an equilibrium path and it is followed along the isotherm, the work done on the system can be expressed as:

$$Work_{CDBFA} = Work_{ACIH} + Work_{BDC} - Work_{ABF}. \tag{3.5}$$

Next, the path ABC can be followed back to state C by taking infinitely small steps so as to maintain reversibility, yielding the following expression for the work done by the system:

$$Work_{ABC} = Work_{ACIH}. \tag{3.6}$$

Now, because the initial and final states are the same, and because it was assumed that the paths followed were reversible, equilibrium paths, there should be no net work. Furthermore, because the temperature was held constant, no heat is transformed into work. As a result, it can be stated that, at equilibrium:

$$Work_{CDBFA} - Work_{ABC} = 0. \tag{3.7}$$

Combining Equation (3.7) with Equations (3.5) and (3.6), Maxwell's equal-area rule is obtained [38, 39]:

$$Work_{BDC} = Work_{ABF}, \tag{3.8}$$

This concept can be expressed rigorously by considering the change in chemical potential along an isotherm, from a liquid to a vapor state:

$$\mu^{vapor} - \mu^{liquid} = \int_{liquid}^{vapor} \left(\frac{\partial \mu}{\partial P}\right)_T dP. \tag{3.9}$$

However, as discussed previously, vapor-liquid equilibrium is defined as:

$$\mu^{vapor}(T, P) = \mu^{liquid}(T, P), \tag{3.10}$$

yielding the result that:

$$\int_{liquid}^{vapor} \left(\frac{\partial \mu}{\partial P}\right)_T dP = 0. \tag{3.11}$$

Using the definition of μ:

$$\mu - \mu\left(T_0, P_0\right) = \int_{T_0}^{T} S_m dT + \int_{P_0}^{P} V_m dP, \tag{3.12}$$

the partial derivative inside the integral becomes:

$$\left(\frac{\partial \mu}{\partial P}\right)_T = V_m \tag{3.13}$$

leading to the expression:

$$\int_{liquid}^{vapor} V_m dP = 0. \tag{3.14}$$

Taken along an isotherm, such as CDBFA shown in Figure 11, Equation (3.14) is the mathematical equivalent of Maxwell's equal-area rule [39]. Interestingly, by starting with Maxwell's mechanical work argument and first asserting that the areas between the isotherm and the constant pressure line must be equal at equilibrium, the mathematical derivation can be reversed. This approach then leads directly to the conclusion that the equality of chemical potentials is a requirement of equilibrium, which is the fundamental condition used in VLE modelling.

Now, due to Maxwell's equal-area rule, P_r is fixed such that it intersects the isotherm in Figure 11 at A, B, and C. The choice then arises of which values to choose of the three roots. To make this decision, the concept of stability must be introduced. While equilibrium, in general, refers to a state that is no longer changing, stability refers to the ability of an equilibrium state to return to itself following a perturbation. An unstable equilibrium state, on the other hand, given even the smallest perturbation, would be permanently changed. A third possibility, a metastable state, is also possible. This refers to a locally stable state that, given a small perturbation will persist, but with a large enough perturbation will be altered. Most real systems exist in metastable states [40]. These concepts can be visualized by considering a mechanical system under the influence of gravity, as shown in Figure 12 [6].

The mathematical criteria for stability can be determined rigorously, but the complete derivation is beyond the scope of this discussion [39–41]. Here, intuition about a physical system is adequate to justify that a state in which an increase in volume leads to an increase in pressure is unstable. Such a system would correspond to one in which the thermal compressibility is negative. This leads to the stability requirement that:

$$\left(\frac{\partial P_r}{\partial V_r}\right)_{T_r} < 0. \tag{3.15}$$

An analysis of the roots shown in Figure 11 reveals that the state labeled B is physically unstable and can never be achieved. Therefore, only states A and C are available as valid states. These two states correspond to the saturated liquid and saturated vapor states, respectively. Thus, the largest root will be chosen when the vapor phase is of interest, and the smallest root for the liquid phase.

The general forms of the equations presented here can now be adapted to specific equations of state. In this work, the Redlich-Kwong-Soave, the Peng-Robinson, and the Peng-Robinson-Stryjek-Vera equations are considered.

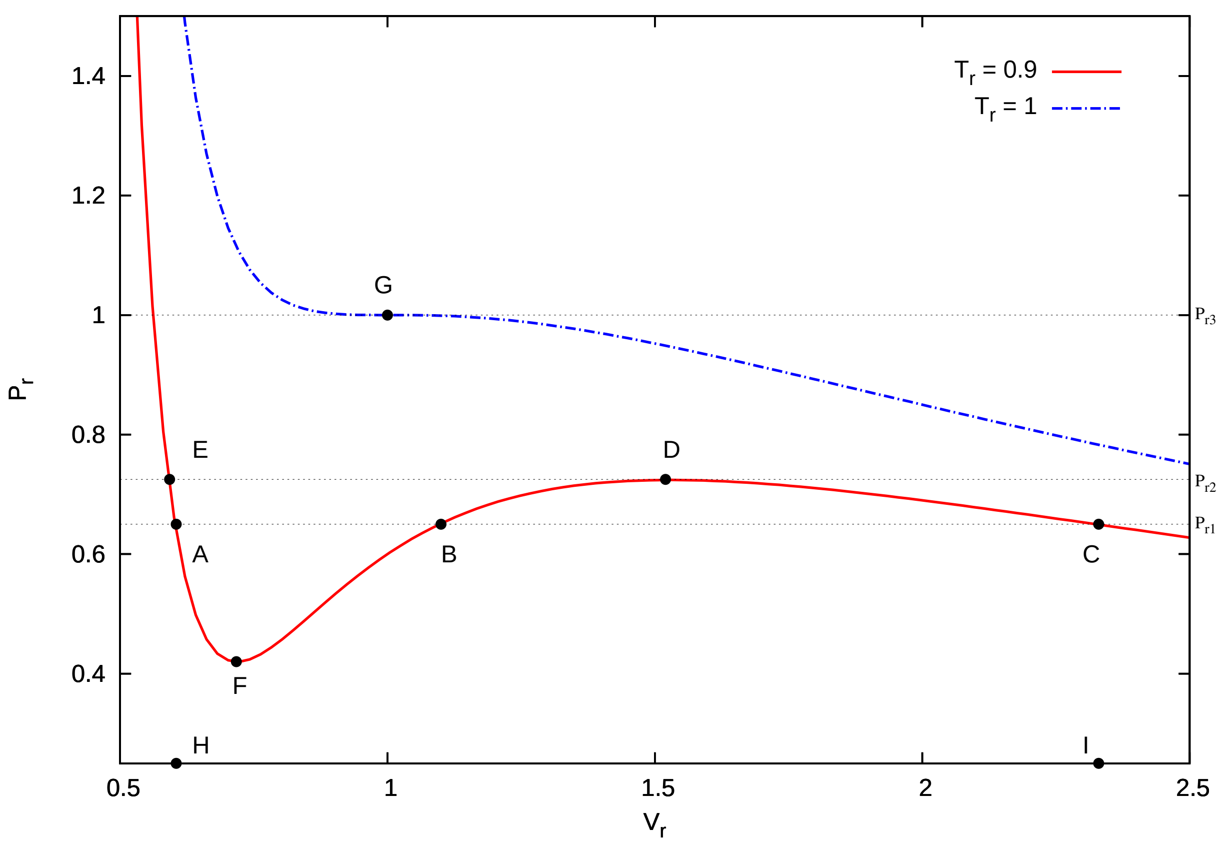


Figure 11: Isotherms calculated using the reduced form of the van der Waals equation of state. Possible roots are indicated by black dots. Adapted from Schaum's Thermodynamics Outline [26].

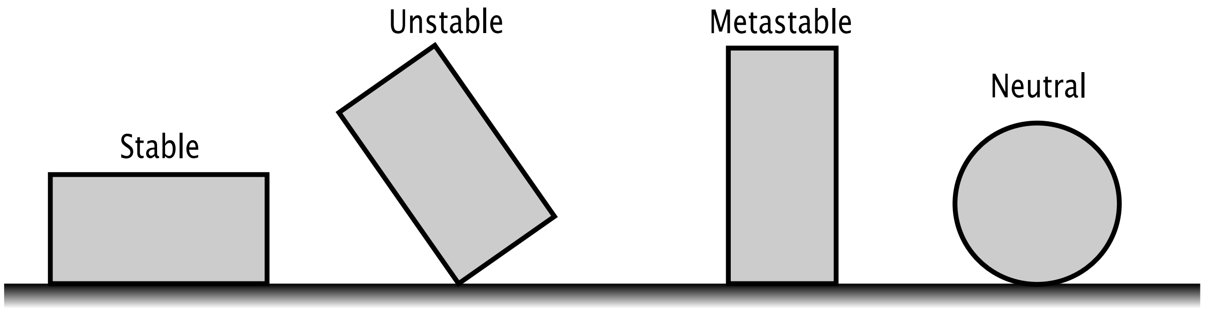


Figure 12: A mechanical system depicting stable, unstable, metastable, and neutral states. Adapted from Richet's *The Physical Basis of Thermodynamics* [6].

3.1 REDLICH-KWONG-SOAVE EQUATION OF STATE

For the Redlich-Kwong-Soave (RKS) equation of state, the parameters presented in the general form above are defined as:

$$\begin{aligned} \eta &= b, \qquad \theta = a\alpha, \\ \delta &= b, \qquad \varepsilon = 0, \end{aligned} \tag{3.16}$$

where the a, α, and b mixture parameters will be defined based on the chosen mixing rules and combining rules as discussed in Chapter 2. However, for the RKS equation, the pure component parameters that comprise the mixture parameters are defined as:

$$a_{ii} = 0.42747\frac{(RT_{c,i})^2}{P_{c,i}}, \tag{3.17}$$

$$b_{ii} = 0.08664\frac{RT_{c,i}}{P_{c,i}}, \tag{3.18}$$

$$\alpha_{ii} = \left[1 + \left(0.48508 + 1.55171\omega_i - 0.15613\omega_i^2\right)\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\right]^2. \tag{3.19}$$

In these expressions, $T_{c,i}$ and $P_{c,i}$ are the critical temperature and pressure, respectively, and ω_i is the acentric factor, all of component i.

Formulating Equation (3.2) in terms of the parameters given in Equation (3.17), yields the pressure explicit RKS equation:

$$P = \frac{RT}{V_m - b} - \frac{a\alpha}{V_m^2 + bV_m}. \tag{3.20}$$

Finally, defining $A = \frac{a\alpha P}{(RT)^2}$ and $B = \frac{bP}{RT}$ yields a simplified RKS compressibility equation:

$$Z^3 - Z^2 + \left(-B^2 - B + A\right) Z - AB = 0. \tag{3.21}$$

This cubic equation can be solved using a variety of methods to determine the compressibility, which will be used in the calculation of the fugacity coefficients.

The next requirement is to determine the partial derivative term in the fugacity coefficient. However, in order to differentiate the pressure explicit form of the RKS equation, it is necessary to write it in terms of the number of moles, n. Using the definition of the molar volume, $V_m = \frac{V_t}{n}$, where V_t is the total volume, Equation (3.20) becomes:

$$P = \frac{nRT}{V_t - nb} - \frac{n^2 a\alpha}{V_t^2 + nbV_t}, \tag{3.22}$$

where the number of moles, as before, obeys the following equation:

$$n = \sum_i n_i. \tag{3.23}$$

Now, differentiating Equation (3.22) with respect to n_i at fixed temperature and total volume:

$$\begin{aligned}\left(\frac{\partial P}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} = \frac{RT}{V_t - nb} &+ \left[\frac{nRT}{(V_t - nb)^2} + \frac{n^2 a\alpha}{V_t\left(V_t + nb\right)^2}\right]\left(\frac{\partial nb}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\ &- \left[\frac{1}{V_t\left(V_t + nb\right)}\right]\left(\frac{\partial n^2 a\alpha}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}.\end{aligned} \tag{3.24}$$

Combining Equations (3.1) and (3.24) then gives:

$$\begin{aligned}RT\ln\hat{\phi}_i = \int_{V_t}^{\infty} &\left[\frac{RTnb}{V_t\left(V_t - nb\right)} + \left(\frac{nRT}{(V_t - nb)^2} + \frac{n^2 a\alpha}{V_t\left(V_t + nb\right)^2}\right)\left(\frac{\partial\left(nb\right)}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}\right. \\ &\left.-\frac{1}{V_t\left(V_t + nb\right)}\left(\frac{\partial\left(n^2 a\alpha\right)}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}\right] \mathrm{d}V_t - RT\ln Z.\end{aligned} \tag{3.25}$$

Because the partial derivatives of the mixture parameters are independent of V_t, this expression can be integrated directly, yielding:

$$\begin{aligned} RT\ln\hat{\phi}_i = &-RT\ln\left(\frac{V_t-nb}{V_t}\right) \\ &+\left[\frac{nRT}{(V_t-nb)}-\frac{na\alpha}{b\,(V_t+nb)}+\frac{a\alpha}{b^2}\ln\left(\frac{V_t+nb}{V_t}\right)\right]\left(\frac{\partial nb}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\ &-\left[\frac{1}{b}\ln\left(\frac{V_t+nb}{V_t}\right)\right]\left(\frac{1}{n}\right)\left(\frac{\partial n^2a\alpha}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}-RT\ln Z. \end{aligned} \tag{3.26}$$

Using the definition of compressibility and the previous simplifications of $A=\frac{a\alpha P}{(RT)^2}$ and $B=\frac{bP}{RT}$, the final equation for the fugacity coefficient of each mixture component for the RKS equation of state can be written as:

$$\begin{aligned} \ln\hat{\phi}_i = &-\ln(Z-B) \\ &+\left[\frac{B}{b}\left(\frac{1}{Z-B}\right)-\frac{A}{b}\left(\frac{1}{Z+B}\right)+\frac{A}{bB}\ln\left(1+\frac{B}{Z}\right)\right]\left(\frac{\partial nb}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\ &-\left[\frac{1}{RTb}\ln\left(1+\frac{B}{Z}\right)\right]\left(\frac{1}{n}\right)\left(\frac{\partial n^2a\alpha}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}. \end{aligned} \tag{3.27}$$

In Appendix A and B, the partial derivatives for the various mixing and combining rules are presented. In the next sections, this same approach is used to develop expressions for the fugacity coefficients of the Peng-Robinson and the Peng-Robinson-Stryjek-Vera equations of state. These derivations are also provided in detail for the sake of completeness, as many treatments of this process omit too many steps.

3.2 PENG-ROBINSON EQUATION OF STATE

For the Peng-Robinson (PR) equation of state, the general parameters are:

$$\begin{aligned} \eta = b, &\qquad \theta = a\alpha, \\ \delta = 2b, &\qquad \varepsilon = -b^2, \end{aligned} \tag{3.28}$$

where, as with the RKS equation, a, α, and b are mixture parameters based on the chosen mixing rules and combining rules. The pure component parameters that comprise the mixture parameters for the PR equation are defined as:

$$a_{ii} = 0.45724\frac{(RT_{c,i})^2}{P_{c,i}}, \tag{3.29}$$

$$b_{ii} = 0.07780\frac{RT_{c,i}}{P_{c,i}}, \tag{3.30}$$

$$\alpha_{ii} = \left[1 + \left(0.37464 + 1.54226\omega_i - 0.26992\omega_i^2\right)\left(1 - \sqrt{\frac{T}{T_{c,i}}}\right)\right]^2. \tag{3.31}$$

Substituting the PR equation parameters into Equation (3.2) leads to:

$$P = \frac{RT}{V_m - b} - \frac{a\alpha}{V_m^2 + 2bV_m - b^2}. \tag{3.32}$$

Again, defining $A = \frac{a\alpha P}{(RT)^2}$ and $B = \frac{bP}{RT}$ gives the simplified PR compressibility equation:

$$Z^3 - (B-1)Z^2 + \left(-3B^2 - 2B + A\right)Z + \left(B^3 + B^2 - AB\right) = 0, \tag{3.33}$$

which is solvable for Z. Equation (3.32) can now be rewritten in terms of the number of moles and total volume as:

$$P = \frac{nRT}{V_t - nb} - \frac{n^2 a\alpha}{V_t^2 + 2nbV_t - n^2b^2}. \tag{3.34}$$

Differentiating Equation (3.34) with respect to n_i yields:

$$\begin{aligned}\left(\frac{\partial P}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} = \frac{RT}{V_t - nb} &+ \left[\frac{nRT}{(V_t - nb)^2} + \frac{2n^2 a\alpha V_t}{(V_t^2 + 2nbV_t - n^2b^2)^2}\right]\left(\frac{\partial nb}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\ &- \left[\frac{1}{(V_t^2 + 2nbV_t - n^2b^2)}\right]\left(\frac{\partial n^2 a\alpha}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\ &- \left[\frac{n^2 a\alpha}{(V_t^2 + 2nbV_t - n^2b^2)^2}\right]\left(\frac{\partial n^2 b^2}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}.\end{aligned} \tag{3.35}$$

Now, combining Equations (3.1) and (3.35):

$$
\begin{aligned}
RT\ln\hat{\phi}_i = \int_{V_t}^{\infty}\Bigg[&\frac{RTnb}{V_t\left(V_t-nb\right)} + \left(\frac{nRT}{\left(V_t-nb\right)^2} + \frac{2n^2a\alpha V_t}{\left(V_t^2+2nbV_t-n^2b^2\right)^2}\right)\left(\frac{\partial\left(nb\right)}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\
&- \frac{1}{\left(V_t^2+2nbV_t-n^2b^2\right)}\left(\frac{\partial\left(n^2a\alpha\right)}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\
&- \frac{n^2a\alpha}{\left(V_t^2+2nbV_t-n^2b^2\right)^2}\left(\frac{\partial\left(n^2b^2\right)}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}\Bigg]\mathrm{d}V_t - RT\ln Z. \qquad (3.36)
\end{aligned}
$$

Finally, integrating this and substituting in for Z, A, and B, as in the previous section, leads to the partial fugacity coefficient for the PR equation of state:

$$
\begin{aligned}
&\ln\hat{\phi}_i = -\ln\left(Z-B\right) \\
+\;&\left[\frac{B}{b}\left(\frac{1}{Z-B}\right) + \frac{A}{2b}\left(\frac{B-Z}{Z^2+2BZ-B^2}\right) + \frac{A}{4bB\sqrt{2}}\ln\left(\frac{Z+B\left(1+\sqrt{2}\right)}{Z+B\left(1-\sqrt{2}\right)}\right)\right]\left(\frac{\partial nb}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\
-\;&\left[\frac{1}{2RTb\sqrt{2}}\ln\left(\frac{Z+B\left(1+\sqrt{2}\right)}{Z+B\left(1-\sqrt{2}\right)}\right)\right]\left(\frac{1}{n}\right)\left(\frac{\partial n^2a\alpha}{\partial n_i}\right)_{T,V_t,n_{j\neq i}} \\
+\;&\left[\frac{A}{8Bb^2\sqrt{2}}\ln\left(\frac{Z+B\left(1+\sqrt{2}\right)}{Z+B\left(1-\sqrt{2}\right)}\right) - \frac{A}{4b^2}\left(\frac{Z+B}{Z^2+2BZ-B^2}\right)\right]\left(\frac{1}{n}\right)\left(\frac{\partial n^2b^2}{\partial n_i}\right)_{T,V_t,n_{j\neq i}}. \qquad (3.37)
\end{aligned}
$$

3.3 PENG-ROBINSON-STRYJEK-VERA EQUATION OF STATE

The Peng-Robinson-Stryjek-Vera (PRSV) equation of state has exactly the same form as the Peng-Robinson equation. The only difference is the way in which the pure component parameters are calculated [42, 43]. For the PRSV equation of state, the general equation parameters are defined as:

$$
\begin{aligned}
a_{ii} &= 0.457235\frac{\left(RT_{c,i}\right)^2}{P_{c,i}} \\
b_{ii} &= 0.08664\frac{RT_{c,i}}{P_{c,i}} \\
\alpha_{ii} &= \left[1+\kappa_i\left(1-\sqrt{\frac{T}{T_{c,i}}}\right)\right]^2 \\
\kappa_{ii} &= \kappa_{0,i} + \left[\kappa_{1,i}+\kappa_{2,i}\left(\kappa_{3,i}-\frac{T}{T_{c,i}}\right)\left(1-\sqrt{\frac{T}{T_{c,i}}}\right)\right]\left(1+\sqrt{\frac{T}{T_{c,i}}}\right)\left(0.7-\frac{T}{T_{c,i}}\right)
\end{aligned}
$$

where $\kappa_{0,i}$, $\kappa_{1,i}$, $\kappa_{2,i}$, and $\kappa_{3,i}$ are all pure substance specific parameters. As these have no affect on the overall form of the equation, Equations (3.33) and (3.37) can still be used for the calculation of the compressibility and the fugacity coefficients, respectively.

A detailed explanation of the derivations was shown here because, with an understanding of this process, the same method can be used to derive fugacity coefficients for any cubic equation of state. An important aspect of this process is the expression of the fugacity coefficients in terms of the partial derivatives of mixture parameters. While many transcripts include these derivatives already evaluated in the fugacity coefficient, this generalized approach allows the incorporation of any number of mixing and combining rules using the same equations. The following chapter will present the methods used to actually calculate the vapor-liquid equilibria data of interest using the fugacity coefficient and compressibility equations derived above.

4.0 NUMERICAL SIMULATIONS

In order to produce vapor-liquid equilibrium data using the equations developed in the previous chapter and to study the effect the different equations have on the results, a numerical routine was written. The code described here, and shown in full in Appendices I - L, is written in the *Fortran 90* computer language. *Fortran 90* was chosen for its combination of a simple programming language and a powerful compiler capable of fast computation times. Message Passing Interface (MPI) was incorporated, where beneficial, to parallelize the computations for execution across multiple CPUs [44]. It is estimated that this implementation improved computation time by up to 17 times on the available systems. The final computations were performed on a Dell XPS 9000 and the Warhol system at the Pittsburgh Supercomputing Center. Details of these platforms are provided in Appendix C.

The iterative numerical procedure of determining the VLE curves is shown in Figure 14 [12]. Throughout the process, there are various values that will be labeled as either fixed or guessed. These names will be used often and it should be remembered that the fixed values remain constant throughout the entire numerical step, while the guessed ones may vary greatly. Here, a numerical step corresponds to the calculation of a single bubble or dew point, as shown in Figure 13. Referring again to Figure 14, one numerical step would be the successful completion of the flow chart, arriving at "DONE." Each loop that is taken within the flow chart, on the other hand, will be referred to as a single iteration. The numerical process as a whole will be used to signify the complete calculation of a bubble or dew point curve, made up of the whole collection of points shown in Figure 13. This process is accomplished by discretizing the composition and carrying out a complete numerical step for each mole fraction element in the composition set. In summary, a single loop within the flow chart is a single iteration, the completion of the chart is one numerical step, and completing the chart multiple times for an entire composition set is the numerical process.

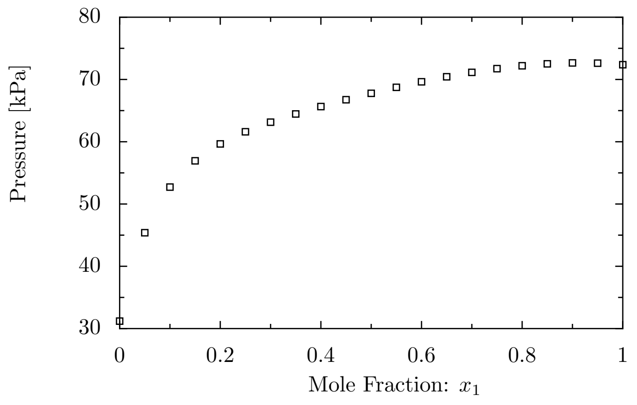


Figure 13: An example VLE data set showing individual bubble points. The entire set is referred to as the bubble point curve.

4.1 COMPUTATIONAL METHODOLOGY FOR THE VAPOR-LIQUID EQUILIBRIUM CALCULATIONS

Step 1 is to fix the mole fractions of one phase and either the temperature or the pressure, which will yield isothermal or isobaric VLE data, respectively. For the calculation of dew and bubble points, fixing the mole fractions corresponds to fixing the mole fractions of both components in one phase. This is a result of the following expression, described previously in Chapter 2.1:

$$z_i = x_i * LF + y_i * VF, \tag{4.1}$$

which reduces, for the bubble point, to:

$$z_i = x_i, \tag{4.2}$$

and for the dew point, to:

$$z_i = y_i. \tag{4.3}$$

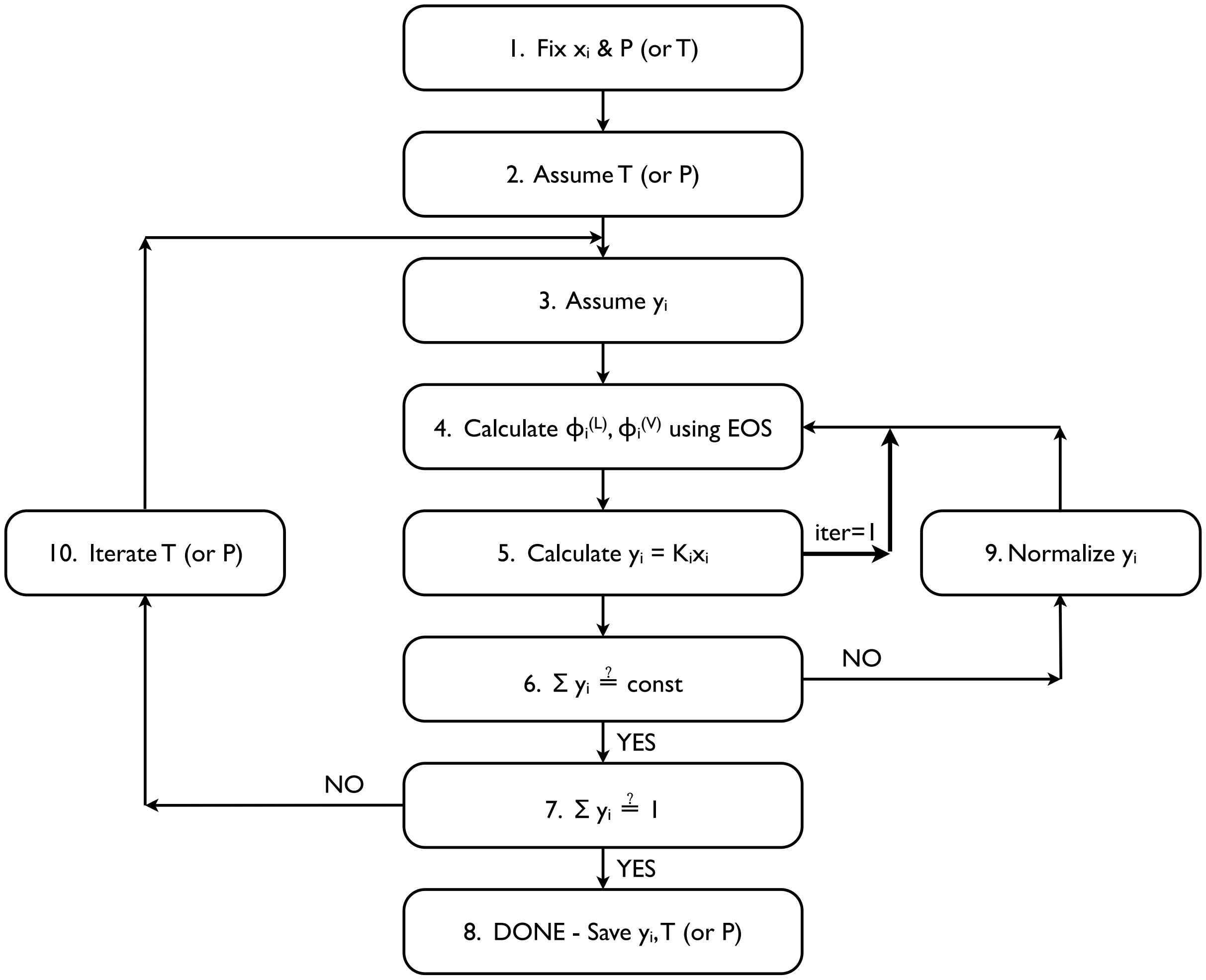

1. Fix x_i & P (or T)
2. Assume T (or P)
3. Assume y_i
4. Calculate $\phi_i^{(L)}$, $\phi_i^{(V)}$ using EOS
5. Calculate $y_i = K_i x_i$
iter=1
9. Normalize y_i
10. Iterate T (or P)
6. $\Sigma y_i \overset{?}{=}$ const
NO
YES
NO
7. $\Sigma y_i \overset{?}{=} 1$
YES
8. DONE - Save y_i, T (or P)

Figure 14: A flow chart describing the iterative procedure to determine a single numerical step of the vapor-liquid equilibrium numerical process. Adapted from Assael, et al. [12]

If the mole fraction that is fixed corresponds to the liquid mole fraction as in Equation (4.2), bubble point data will be determined at that specific mole fraction. Dew point data will also be calculated, but only at a vapor mole fraction that is also determined during the numerical step. While fixing the liquid mole fraction allows one to determine bubble point data for a set range of compositions, at a predetermined step-size, the range of dew point data determined throughout the process will most likely not span the entire range of compositions, nor will the data points correspond to the same step-size chosen for liquid mole fractions. However, if the fixed mole fraction represents the vapor mole fraction, as in Equation (4.3), the dew point data is calculated regularly at those values, with the bubble point data arising out of the calculations. As a result, in order to determine complete dew and bubble point curves, the entire numerical process is carried out twice, once fixing the liquid mole fractions at each step and once fixing the vapor mole fractions.

It is required that the sum of the mole fractions of all the components in each phase equals one, as shown in Equation (4.4).

$$\sum_i z_i = 1 \tag{4.4}$$

For a binary mixture, during the calculation of bubble points, this yields the following requirement:

$$z_1 + z_2 = x_1 + x_2 = 1. \tag{4.5}$$

Similarly, for the calculation of dew points for a binary mixture:

$$z_1 + z_2 = y_1 + y_2 = 1. \tag{4.6}$$

Therefore, in Step 1, it is necessary to fix either the temperature or the pressure and the mole fraction of only one of the components. The decision of whether bubble or dew point data is desired will determine if Equation (4.5) or Equation (4.6) is used, respectively. From this point on, it will be assumed for simplicity that bubble point data is being calculated. Following the description of the numerical process, in Section 4.2, a few notes will be made concerning important changes required for the calculation of dew points.

In Step 2, the value of the pressure or the temperature is guessed to simulate an isothermal or isobaric data set, respectively. This choice can be done arbitrarily, but making a poor initial guess can greatly decrease the speed of the numerical routine. Better options for this choice will be discussed in Section 4.3.1.

In Step 3, the mole fraction of the thus undetermined phase is guessed. Specifically, for the bubble point calculation where the liquid mole fraction was fixed, the vapor mole fraction is guessed. For convenience, this guess is taken as the current values of the mole fraction of the fixed phase, $y_i = x_i$.

In Step 4, the liquid and vapor fugacity coefficients are calculated from Equation (2.9), shown again here for convenience:

$$RT \ln \hat{\phi}_i = \int_{V_t}^{\infty} \left[\left(\frac{\partial P}{\partial n_i} \right)_{T,V_t,n_{j\neq i}} - \frac{RT}{V_t} \right] dV_t - RT \ln Z. \tag{4.7}$$

As described previously, for the liquid coefficient, the smallest root of the compressibility equation of state is chosen and for the vapor, the largest. It should be noted that both of these coefficients are guessed values, as both the pressure and the vapor mole fractions used in the calculation are only guesses of the current iteration in this numerical step.

In Step 5, the equilibrium ratio is determined in terms of the fugacity coefficients. This is done by considering that at equilibrium, the fugacity coefficients of the mixture are equal across phases, as discussed previously. This leads to an expression for the partial fugacity coefficients at equilibrium as follows:

$$y_i \hat{\phi}_i^V = x_i \hat{\phi}_i^L. \tag{4.8}$$

Then, using the definition of the equilibrium ratio:

$$K_i \equiv \frac{y_i}{x_i}, \tag{4.9}$$

Equation (4.8) can be used to write:

$$K_i = \frac{\hat{\phi}_i^L}{\hat{\phi}_i^V}. \tag{4.10}$$

These ratios are also guessed values of the current iteration due to their dependence on the guessed fugacity coefficients. Next, using these values of K_i, new values for y_i are guessed.

This is done using the original form of the equilibrium ratio in terms of vapor and liquid mole fractions, leading to the new vapor mole fraction guesses being calculated from:

$$y_i = K_i * x_i. \tag{4.11}$$

During the first iteration, Steps 4 and 5 are carried out again using these new values of y_i. Once completed, the new guesses of y_i determined in the second iteration of Step 5 are compared to those found in the previous iteration. To do this, the sum of the y_i values is compared between the two iterations to see if it is changing. An unchanging sum is represented by Equation (4.12):

$$\left|\sum_i y_i^{old} - \sum_i y_i^{new}\right| < \epsilon, \tag{4.12}$$

where ϵ is chosen as a very small number representing convergence. The value chosen for ϵ can vary and the effect of its variation will be discussed further in Section 4.4.2. Depending on whether or not Equation (4.12) is satisfied, two different approaches are taken. First, if it is satisfied, the process continues to Step 7. However, if the sum of y_is is changing, the process jumps to Step 9.

By Step 7, it has been determined that the sum of y_is is not changing. Subsequently, the value of the sum is checked to see if it is equal to one. While this is required by Equation (4.6), it has not yet been enforced. Therefore, it represents an indication that the correct values have been guessed. As a result, if Equation (4.13) is satisfied, the numerical step is complete and the current value of pressure is the bubble point pressure for the fixed values of the liquid mole fractions and the temperature. Additionally, these values of y_i correspond to dew point data, as discussed previously.

$$\left|\sum_i y_i - 1\right| < \epsilon \tag{4.13}$$

If Equation (4.13) is not satisfied, then the program progresses to Step 10.

Step 9 only occurs when the sum of y_is is changing in Step 6. When this is the case, the guessed values of the y_is are not correct and new values must be chosen. It is debatable what the best method for this update is, but a common approach is to normalize the values of y_i using the sum of the values, as shown in Equation (4.14):

$$y_i = \frac{y_i}{\sum_i y_i}. \tag{4.14}$$

After this adjustment is made, the previous steps should be repeated using the new values of the y_is. This corresponds to a return to Step 4. However, there is no need at this point to repeat the calculation of the liquid fugacity coefficient as the parameters influencing its calculation have not been modified. Only the vapor fugacity coefficients need to be recalculated.

Step 10 is executed when Equation (4.13) is not satisfied. When this occurs, the current value of the sum should be considered and the pressure should be modified accordingly. If $\sum_i y_i < 1$, then the guessed value of the pressure is leading to the prediction of too little vapor and the value should be decreased. However, for $\sum_i y_i > 1$, the guessed value of the pressure should be increased due to the prediction of too much vapor. After making this adjustment, a new iteration must be carried out using this new value for pressure. Thus, the process returns to Step 3 and repeats.

4.2 MODIFICATIONS FOR DEW POINT CALCULATIONS

As noted previously, the routine presented above focused on the calculation of bubble point data. In order to determine dew point data, a few changes must be made to the numerical routine. First, in Step 3, the value of x_i is guessed. This is done using the value of y_i by simply equating the two, ($x_i = y_i$), if no better choice exists. However, a better option for the guess value will be discussed in Section 4.3.1. Regardless of the manner in which the value is set, the significance of defining x_i as the guessed value and y_i as the fixed value will become clear in the next step.

In Step 5, the ratio of the fugacity coefficients and new values for the x_is are calculated. However, now notice that the solution of Equation (4.9) for x_i yields:

$$x_i = \frac{y_i}{K_i}, \tag{4.15}$$

where K_i is a guessed value. This is an important change, as bad guess values that cause K_i to approach zero can quickly make the guessed values of x_i blow up, creating problems with convergence. Methods to avoid this issue are discussed in Section 4.4.1.

The routine then progresses in the same manner as if calculating the bubble point data except the sums of x_is are checked for changes and equality to one instead of the sums of the y_is. When the sum of the x_is is fixed and yet not equal to one, the direction of adjustment of the pressure is opposite that of the bubble point routine. In other words, in Steps 7 and 10, if $\sum_i x_i < 1$, the predicted amount of liquid is too low and the guessed value of the pressure should be increased. On the other hand, for $\sum_i x_i > 1$, the guessed pressure value needs to be decreased, as too much liquid is predicted.

Finally, in Step 9, when the liquid mole fraction analog of Equation (4.12), expressed for simplicity in Equation (4.16), is not satisfied, the adjustment of the guessed x_i values is simply based on Equation (4.17), where the *old* and *new* values correspond to those calculated in the first and second iteration of Step 5, respectively. This approach is taken instead of normalization to avoid the possible divergence that might arise if $\sum_i x_i \to 0$.

$$\left| \sum_i x_i^{old} - \sum_i x_i^{new} \right| < \epsilon \tag{4.16}$$

$$x_i^{old} = x_i^{new} \tag{4.17}$$

One final note concerns the calculation of the temperature-composition data at a fixed pressure. When this process is carried out, the direction of the adjustments of the temperature should be reversed from those used in the isothermal routine. This corresponds, for the isobaric bubble point calculations, to:

$$T_{new} = \begin{cases} T_{old} & + \Delta T \text{ if } \sum_i y_i < 1, \\ T_{old} & - \Delta T \text{ if } \sum_i y_i > 1, \end{cases} \tag{4.18}$$

and for the dew point calculations, to:

$$T_{new} = \begin{cases} T_{old} & - \Delta T \text{ if } \sum_i x_i < 1, \\ T_{old} & + \Delta T \text{ if } \sum_i x_i > 1. \end{cases} \tag{4.19}$$

4.3 PERFORMANCE IMPROVEMENTS

There are many methods that can be used to enhance the performance of the numerical routines used for vapor-liquid equilibria calculations that are often not discussed in treatments about VLE simulations. The following sections overview a few of the more significant alterations used in this work.

4.3.1 Iterations Update Methods

The initial guess for the pressure or temperature will greatly affect the performance. As a result, for the very first composition that is calculated, the guess value is taken as an input from the user. However, subsequent guess values are taken to be the value calculated for the previous composition step. If the step-size is chosen to be small enough, this greatly improves the speed of the process by taking as a guess value, one that is very close to the actual value to be determined. If the step-size is fairly large on the other hand, a linear interpolation using the previous two steps is used instead to set the new guess value.

Furthermore, there are a variety of ways one can update the pressure or temperature. The easiest case is just to incrementally increase or decrease the value by a fixed amount each time it is necessary. However, this can cause two problems. The first issue is that too small or too large of an incremental change will lead to very slow changes or large overshoots, respectively, and consequently, the routine will be inefficient. Additionally, if changes that are too large are allowed, the possibility emerges that the solution may never actually converge, with the routine continually increasing, then decreasing the pressure or temperature, while never reaching a value that satisfies the convergence parameter. As a result, an incremental change was implemented in this work but it is chosen such that the value of that increment is dynamically altered based on the current state of the routine. If the current iteration step is not converging due to a constantly changing sum of the mole fractions, large incremental steps are taken. However, if the routine finds itself with a constant mole fraction sum that is not unity, the pressure or temperature value is refined using smaller and smaller incremental steps until convergence is achieved. The choice of the initial size of the incremental step is an interesting one that will be discussed further below.

Finally, the values of the guessed compositions can be improved when both bubble point and dew point data are being calculated together. As was discussed previously, some dew point data is obtained as a result of the bubble point determinations. By storing the liquid and vapor mole fraction data from the bubble point routine, the fixed vapor mole fractions used to calculate the dew points can be compared to those calculated in the bubble point. Then, the corresponding liquid mole fractions of the bubble points can be used as guess values for the dew points.

4.4 NUMERICAL ISSUES

4.4.1 Complications

The cubic equation of state in terms of the compressibility must be solved at the guessed pressure and mole fractions. However, methods that are capable of determining "exact solutions" can be dangerous to implement due to their occasional calculation of erroneous roots. Similarly, while routines like the Newton-Raphson method are extremely efficient and fast, they too can have some serious convergence problems if presented with improper bounds or poor initial guesses. Normally, a Newton-Raphson method or an exact solution would be completely adequate for determining vapor-liquid equilibrium curves, and these are very often used successfully. However, a major goal of this work is to test the convergence of various equations with parameters purposefully set very far from their optimal values. As a result, it is very important to know that any divergences that are occurring are a consequence of the equations and their parameters rather than a possible divergence in the calculation of a root. As a result, the root-finder implemented here is based on Laguerre's Method, as implemented in *Numerical Recipes in Fortran* [45]. Laguerre's Method is a general method that uses complex computations and a "rather drastic set of assumptions." The method assumes that the root being sought, x_1, is a distance, a, from the current guess, x, so that:

$$a \equiv x - x_1, \tag{4.20}$$

and that all of the remaining roots, x_i, regardless of where they actually are, exist at the same distance, b, from the current guess, so that:

$$b \equiv x - x_i. \tag{4.21}$$

This simplification allows the first root to be found by calculating a for a guessed value of x, then slowing refining toward the root by using $x - a$ as the subsequent guess value until a approaches 0 [7, 45]. While the use of this method adds complexity, for a polynomial with only real roots, convergence by this method is guaranteed regardless of the initial guess value. This routine is chosen as it represents an excellent trade-off between efficiency and robustness. A more comprehensive description of Laguerre's Method can be found in Press' *Numerical Recipes in Fortran*, in which the source code is also detailed.

It was mentioned before that the van Laar form of the combining rule is capable of producing very accurate results given correct binary interaction parameters, but that it can have mathematical poles given certain (k_{12}, k_{21}) pairs. As a result, it becomes important to always evaluate $z_1k_{12} + z_2k_{21}$ for the current values of the z_is and k_{ij}s. If this expression is zero, the use of the van Laar combing rule with the current values will result in divergence and the breakdown of the numerical procedure. There are, however, different paths to this divergence, each of which ought to be addressed differently. For example, if the pole occurs in the bubble point algorithm during the calculation of liquid fugacity coefficients, then it is permanent, and the combining rule must be switched if data for the current composition and binary interaction pairs is desired. Luckily, simpler combining rules can be implemented which, while not reproducing data as accurately, do not suffer from the divergence problems of the van Laar form. If the pole exists in another instance, however, such as the calculation of vapor fugacity coefficients in the bubble point routine, changing the combining rule is unnecessary. Instead, it is possible to simply change the current values of the mole fractions, because they are just guessed values in these instances, and proceed normally. The existence of poles is handled in this manner in this work. At the beginning of the bubble and dew point routines and at each new mole fraction guess, when the van Laar combining rule is used, it is first checked for poles. If they are found, they are dealt with according to the above procedure, with an alternative combining rule being specified by the user at the start of the program.

4.4.2 The Effect of Parameters on Convergence

There are several parameters used throughout the numerical routine that must be set at the beginning of the simulation. The chosen values of these parameters influence both the run time and the convergence. A data point that was known to have convergence issues was purposefully chosen here to provide a more insightful analysis. Tables 2 - 5 show the result of a study that varies the following parameters: *adj, perturb, conv, and maxiters.* For these studies, the bubble points for a mixture of 83.96% ethanol and 16.04% water are calculated. The value of each binary interaction parameter is allowed to range, inclusively, between -1 and 1, by 0.1 increments. This leads to the simulation of 21^2 data points. The set of 441 calculations is run for varying values of the parameters listed above. There are some important points to note here. First, these studies do not take performance into account. Many of the calculated values deviate greatly from the experimental data due to their use of extremely erroneous binary interaction pairs. The only interest is whether the points are capable of, or are allowed to, converge. Additionally, while this takes the form of a sensitivity study, the objective is very different. Many of these parameters affect the outcome of the simulations, especially *maxiters*, and this is largely deliberate. Rather than attempting to find the parameter space that does not impact the results of the calculations, the intention is to determine the parameter values that provide the best trade-off between routine convergence and computation times. This focus falls in line with the desire to develop computational methods with a high degree of usability. While a simulated data point that converges after multiple hours of iterations does indeed converge, it is not practical to allow, or even attempt, convergence. It would be better instead, to treat this point as one which will never converge and move on with the simulation.

The *adj* parameter is the initial value that is used to adjust the guessed pressure value. It is important to remember that this value will be changed throughout the routine in order to reduce the possibility of non-convergence. However, the initial value still affects the overall numerical process. *adj* is implemented additively, as shown in Equation (4.22).

$$P_{new} = P_{old} \pm adj \tag{4.22}$$

perturb is a multiplicative value that corresponds to the amount the guessed mole fraction values, the y_is in this case, are perturbed when a pole is encountered in the combining rule or when the fugacity coefficients are not converging properly. The expression used for this perturbation takes the following form:

$$y_{i,new} = y_{i,old} * perturb \tag{4.23}$$

conv is the parameter that determines when a solution has converged. In the previous text, *conv* is referred to as ϵ.

maxiters corresponds to the maximum number of iterations that are allowed before the numerical routine decides it is not converging to a solution. When this value is reached, the program attempts to find a new pressure value that will allow convergence. This is done by taking a jump from the current guessed pressure value, first to one that is significantly higher and second, if necessary, to one that is lower. Once one jump is made, the current iteration count is reset and the routine proceeds another *maxiters* iterations, trying to converge. After making both jumps, if the program still has not found a value, it is assumed that the current k_{ij} pair will never converge. As a result, this parameter has a profound effect on the outcome of the routine. If the parameter is too large, the routine could get stuck trying to convergence to an invalid value, greatly increasing runtime with little benefit. However, with too small of a value, the guessed pressure value will behave sporadically, never having adequate time to converge, and no solution will be found, despite its existence.

The following chapter will present the results obtained using the numerical methods described above. The parameters which were varied in this chapter are subsequently fixed to provide a decent trade-off between convergence and runtime and any numerical issues are dealt with as described in the preceding sections.

Table 2: Effect of *adj* Value on Runtime and Convergence

adj	Runtime [s]	Non-converged Points	*adj*	Runtime [s]	Non-converged Points
0.228365	534.4	387	150.000	85.99	157
0.342548	567.7	379	225.000	81.65	151
0.513823	4495.	358	337.500	39.66	141
0.770734	330.3	309	506.250	63.27	84
1.15610	108.4	262	759.375	23.25	81
1.73415	88.50	255	1139.06	14.04	58
2.60122	87.65	253	1708.59	62.01	77
3.90184	90.02	252	2562.89	21.64	77
5.85276	93.68	250	3844.33	10.12	38
8.77914	101.1	244	5766.50	10.55	39
13.1687	96.42	235	8649.75	52.41	38
19.7530	143.9	215	12974.6	13.35	50
29.6296	170.2	200	19461.9	15.83	50
44.4444	130.2	187	29192.9	18.15	49
66.6666	110.7	173	43789.3	61.47	50
100.000	88.47	168			

$perturb = 1.001$, $conv = 1e-5$, and $maxiters = 1e4$

Table 3: Effect of *perturb* Value on Runtime and Convergence

perturb	Runtime [s]	Non-converged Points
1.00000000001	47.82	168
1.0000000001	87.72	168
1.000000001	87.71	168
1.00000001	87.84	168
1.0000001	87.86	168
1.000001	47.86	168
1.00001	87.88	168
1.0001	87.87	168
1.001	87.91	168
1.01	87.85	168
1.1	47.89	168
$adj = 100$, $conv = 1e - 5$, and $maxiters = 1e4$		

Table 4: Effect of *conv* Value on Runtime and Convergence

conv	Runtime [s]	Total Non-converged Points
1e-9	99.75	199
1e-8	95.60	195
1e-7	48.35	191
1e-6	89.50	179
1e-5	87.02	168
1e-4*	100.8	160
1e-3*	109.1	153
1e-2*	77.09	83
1e-1*	235.5	694

$adj = 100$, $perturb = 1.001$, and $maxiters = 1e4$

* = Convergence parameters this large will likely produce erroneous results.

Table 5: Effect of *maxiters* Value on Runtime and Convergence

maxiters	Runtime [s]	Total Non-converged Points
1	0.184	418
10	0.644	415
100	4.486	359
1e3	47.79	237
1e4	47.20	168
1e5	255.5	59
1e6	1502.	47
1e7	25407	46
1e8	N/A	N/A
$adj = 100$, $perturb = 1.001$, and $conv = 1e-5$		
N/A = Runtime was prohibitively long.		

5.0 RESULTS AND DISCUSSION

A typical VLE study involves the calculation of bubble point and dew point data at either a fixed temperature or fixed pressure, using a specific combination of equations of state, mixing rules, and combining rules. In this chapter, the results from such studies are presented to show the fundamental operation of the developed method, utilizing the designed flexibility of the program to model many different equation combinations. This will be followed by a discussion concerning the effect of the binary interaction parameters on varying combinations of equations, as well as the results from an analysis of a broad range of these parameters. Finally, it will be shown how the availability of data can affect which equations should be used for VLE modeling.

Throughout the text, the equations that are used are assigned numbers for simplicity. These numbers are then combined into a string representing the equation set with the form: equation of state (EOS), a parameter combining rule (CR), b parameter CR, a mixing rule (MR), b MR. For example, the string 23010 represents the use of EOS number 2, a CR 3, b CR 0, a MR 1, and b MR 0. Where it is possible without creating too much confusion, abbreviations are also provided. Table 6 provides a key between the equations and their numbers and abbreviations. In this study, for simplicity, the linear mixing rule is used for the b parameter at all times. While this is not required, it is a common practice, as b simply represents the co-volume parameter [46].

The experimental data used for the following studies, which is summarized in Table 7, was taken from volumes of the *Vapor Liquid Equilibrium Data Collection* compiled by Gmehling, Onken, and Arlt [47, 48].

Table 6: Numeric indicators for different equations. These are combined in the form: (EOS, a CR, b CR, a MR, b MR).

Equation	Numeric Indicator	Abbreviation
Redlich-Kwong-Soave EOS	0	RKS
Peng-Robinson EOS	1	PR
Peng-Robinson-Stryjek-Vera EOS	2	PRSV
Arithmetic CR	0	A
Conventional CR	1	C
Margules Form CR	2	M
van Laar Form CR	3	VL
Linear MR	0	L
Quadratic MR	1	Q

Table 7: Summary of the experimental VLE data used in this study.

Temperature [K]	Number of datapoints	Reference
298.15	10	[47]
313.15	13	[48]
323.65	9	[48]
333.75	11	[48]
343.15	13	[48]

5.1 THE EFFECT OF EQUATION COMBINATIONS ON STANDARD VLE STUDIES

A standard VLE calculation was first performed at a fixed temperature of $T = 323.65K$ to develop bubble and dew point curves for the various equations previously described. Figure 15 compares only a change in the equation of state, with fixed combining and mixing rules. It should be noted that although some plots are seemingly neglected, this is due to the fact that certain equation combinations produce identical data. Specifically, if a mixing rule of 0 is chosen for a parameter, ζ, this corresponds to linear mixing ($\zeta_{mix} = z_1\zeta_{11} + z_2\zeta_{22}$). Because the combining rule, as discussed previously, is used for the calculation of the cross-terms (ζ_{12} and ζ_{21}), varying the combining rule with a linear mixing rule would not affect the data. For those simulations that are affected by the combining rule however, optimal values are used for the binary interaction parameters, k_{12} and k_{21}, in this analysis. These values are shown in Table 8 and are also presented on each plot. Additionally, the $\%\ Error_{EXP}$, labeled simply $\%\ Error$ on the plots for convenience, is supplied. This value refers to an average of the percent errors calculated at each available experimental data point, as shown in Equation (5.1):

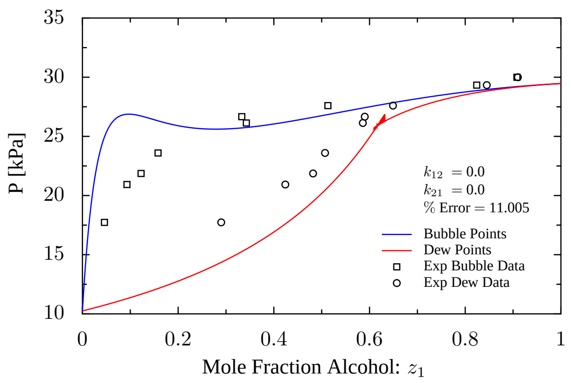
$$\%\ Error_{EXP} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{P_{i,calc} - P_{i,exp}}{P_{i,exp}}\right| * 100. \tag{5.1}$$

As can be seen by examining the values of this error in Figure 15, the choice of equation of state alters the performance of the simulated results, the accuracy improving with increasing equation complexity. However, it is evident that even the most complex equation, the PRSV, still calculates an erroneous overall shape for the bubble point curve.

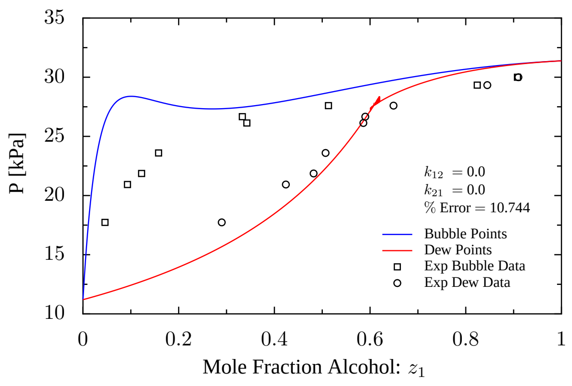
Figures 16 - 18 present all of the equation combinations considered in this study. Here, it can be seen that by varying the complexity of the combining rule, the performance of the calculated data is significantly affected, even more so than by modifying only the equation of state. For all of the equations of state, as the complexity of the combining rule increases, so does the performance of the simulation. However, the simple arithmetic (0) and conventional (1) combining rules have very similar behavior, as do the complex Margules form (2) and van Laar form (3) combining rules. While all the equations of state calculate reasonable data

Table 8: Optimal k_{ij} parameters used for each equation set at $T = 323.65K$.

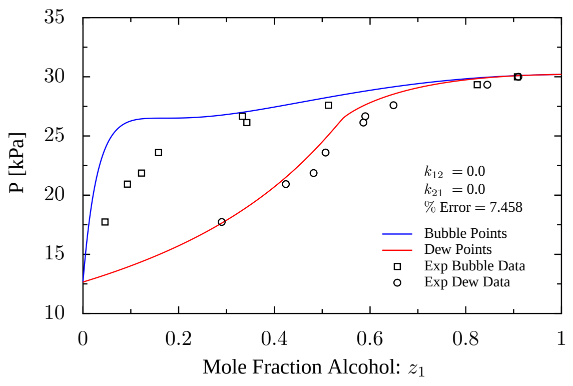
Equation Set	k_{12}	k_{21}
00010	-0.1	0.07
01010	-0.1	-0.1
02010	-0.06	-0.12
03010	-0.12	-0.07
10010	-0.1	0.06
11010	-0.1	-0.12
12010	-0.09	-0.12
13010	-0.12	-0.09
20010	-0.1	0.07
21010	-0.1	-0.11
22010	-0.08	-0.12
23010	-0.12	-0.09



(a) EOS = 0 (RKS)



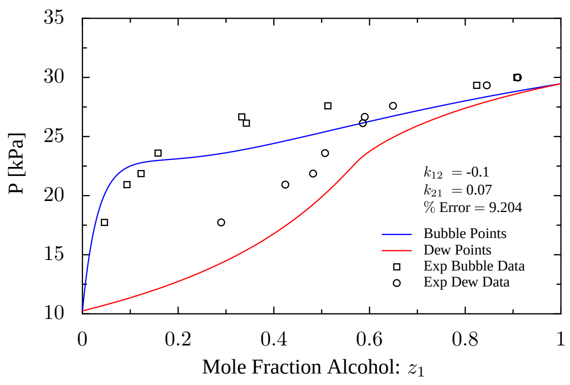
(b) EOS = 1 (PR)



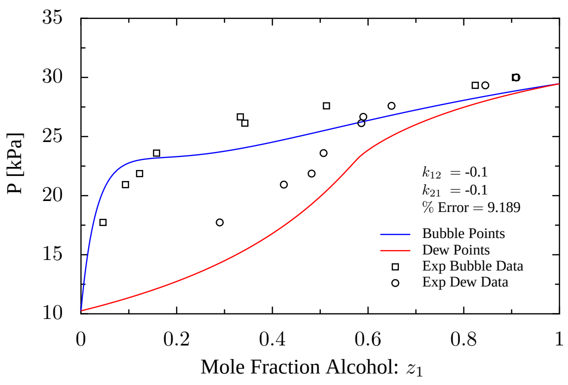
(c) EOS = 2 (PRSV)

Figure 15: VLE results at $T = 323.65K$ for varying equation of state.

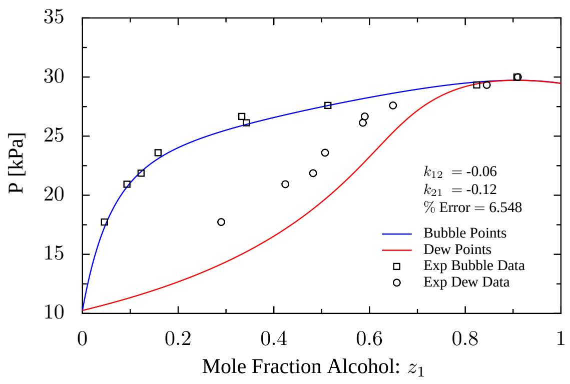
for the bubble point curves for the complex combining rules, an increase in complexity leads primarily to improvements in the dew point data simulation. As shown in Figure 18d, the use of complex equations, such as the Peng-Robinson-Stryjek-Vera equation, with the van Laar combining rule and the quadratic mixing rule for the a parameter, yields very accurate data reproduction. A similar presentation of figures, which show the VLE simulations at a fixed pressure, can be found in Appendix D.



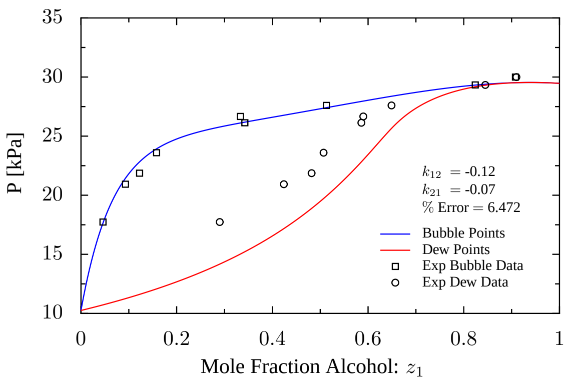
(a) CR = 0 (A)



(b) CR = 1 (C)



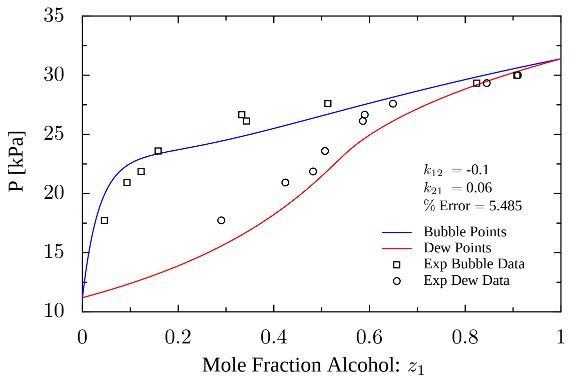
(c) CR = 2 (M)



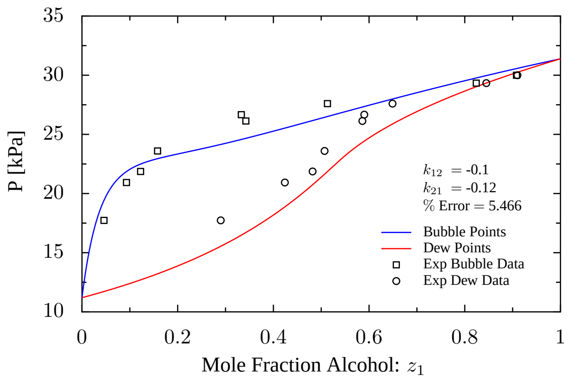
(d) CR = 3 (VL)

Figure 16: VLE results at $T = 323.65K$ for varying combining rules. EOS = 0 (RKS).

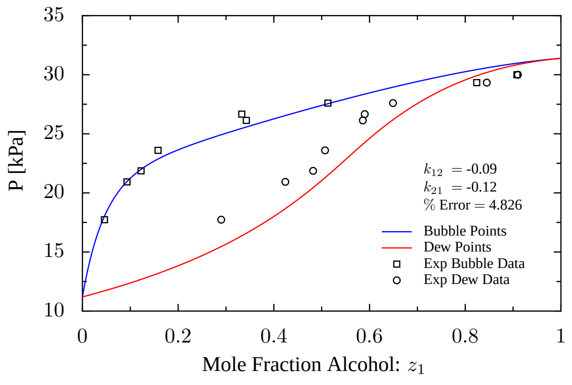
In addition to varying equation combinations, the program can also calculate data for a variety of temperatures and pressures. Figure 19 shows the VLE simulations at multiple temperatures for a few of the different equation combinations. The remaining combinations can be found in Appendix E. As can be seen in these plots, the more complex combining rules tend to give more accurately shaped curves across the temperature range considered.



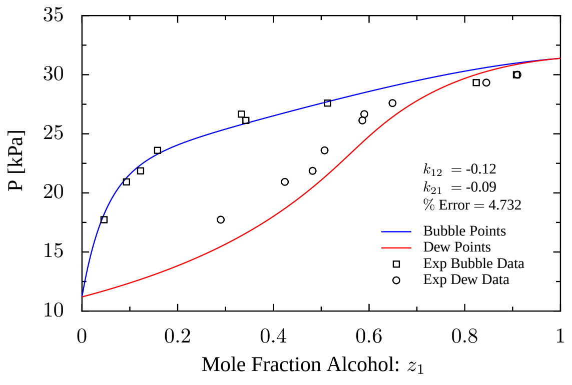
(a) CR = 0 (A)



(b) CR = 1 (C)

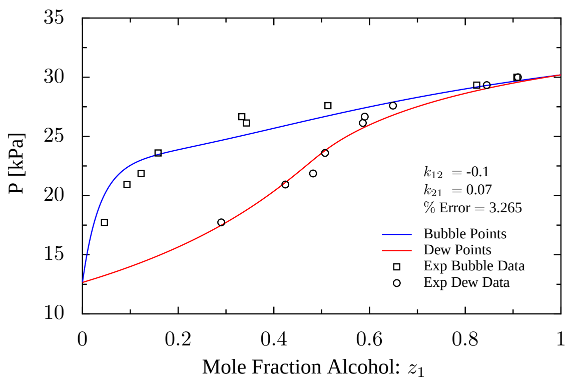


(c) CR = 2 (M)

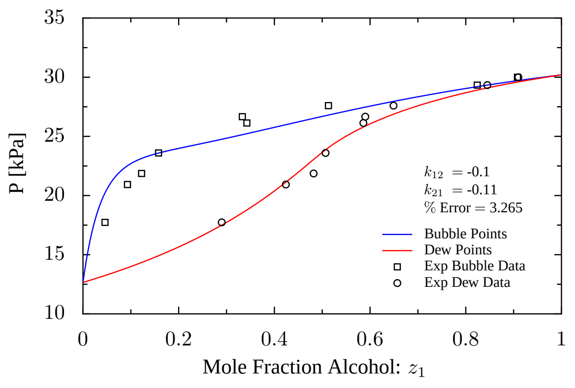


(d) CR = 3 (VL)

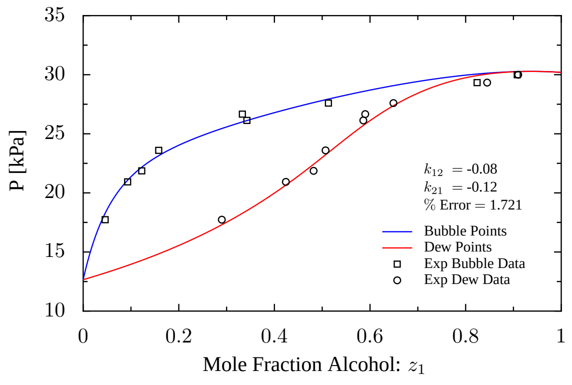
Figure 17: VLE results at $T = 323.65K$ for varying combining rules. EOS = 1 (PR).



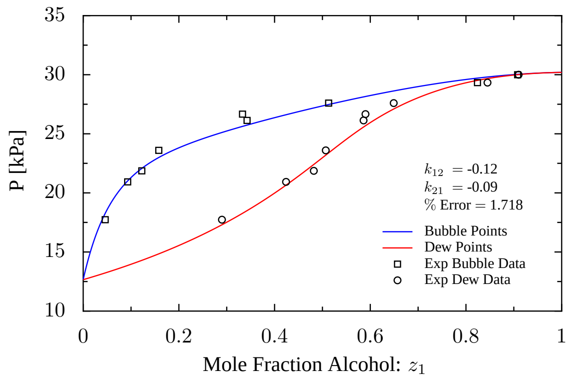
(a) CR = 0 (A)



(b) CR = 1 (C)

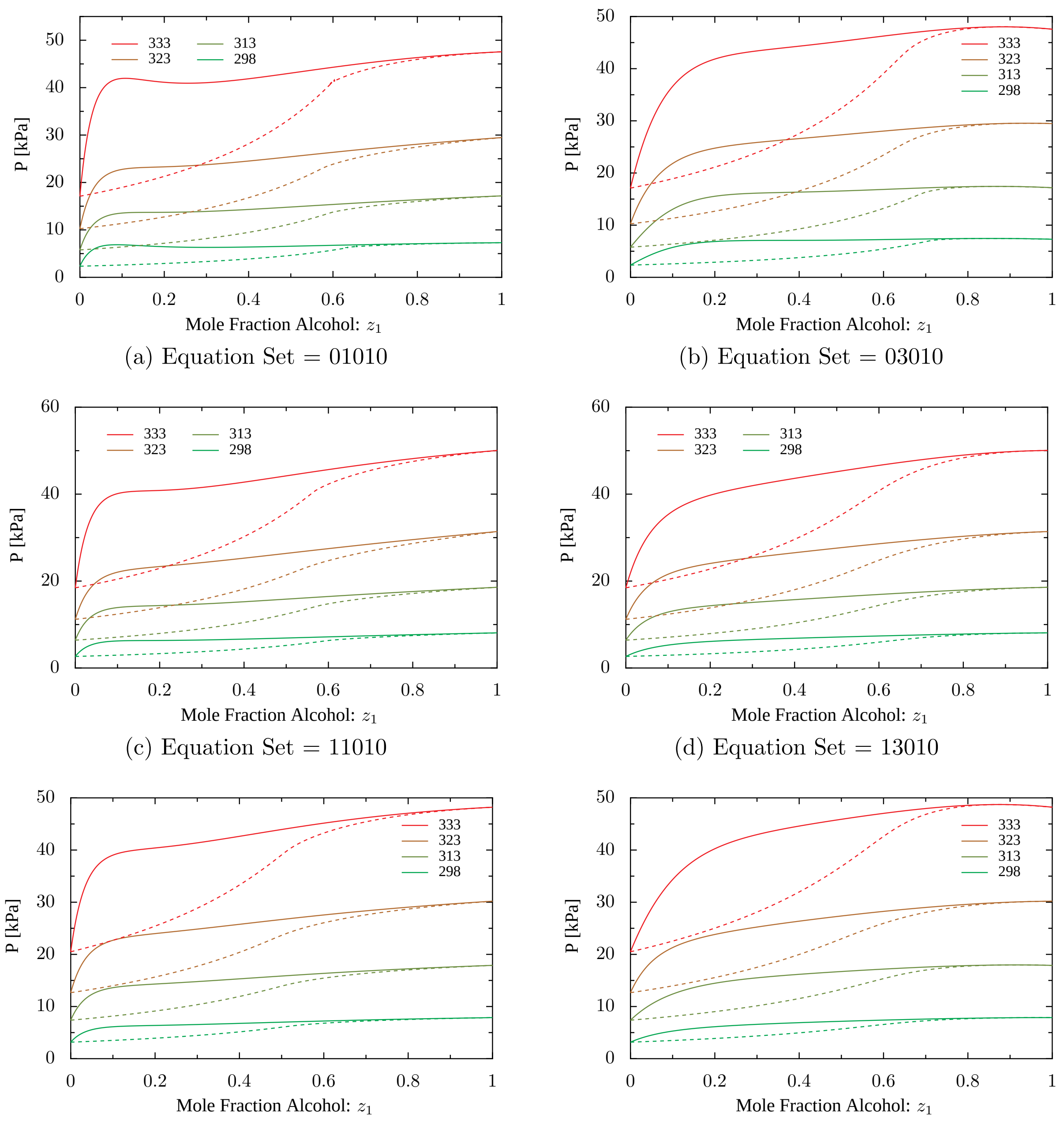


(c) CR = 2 (M)



(d) CR = 3 (VL)

Figure 18: VLE results at $T = 323.65K$ for varying combining rules. EOS = 2 (PRSV).



(a) Equation Set = 01010

(b) Equation Set = 03010

(c) Equation Set = 11010

(d) Equation Set = 13010

(e) Equation Set = 21010

(f) Equation Set = 23010

Figure 19: VLE results at varying temperatures for varying equation combinations.

5.2 PERTURBATION OF THE BINARY INTERACTION PARAMETERS

The figures presented above have shown the effect that the choice of combining rule can have on the performance of a VLE simulation. It is clear that, in general, as the complexity of the combining rule increases, the performance of the simulation improves. This makes sense as that is the reason why more complex combining rules are created in the first place. However, the previous calculations were all carried out using binary interaction parameters that were optimized for the specific equations and at the specific temperatures being implemented. However, as available experimental data decreases in either quantity or quality, this optimization becomes less reliable. It is therefore important to understand the limitations of these equations. To elucidate this concept, each simulation was run again, but in addition to the optimal binary interaction parameters, perturbed values were also considered. The perturbations represent 15% deviations from optimal, as shown in Equation (5.2):

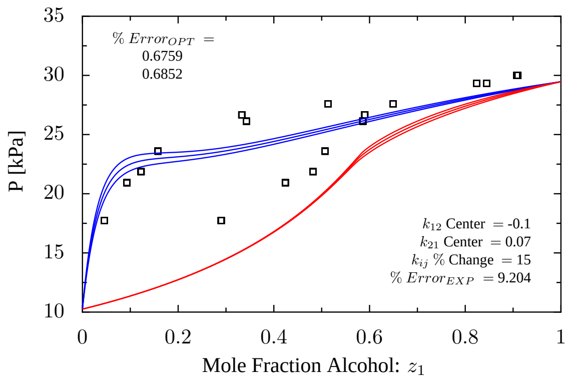
$$k_{ij,perturbed} = k_{ij,optimal} \pm k_{ij,optimal} * 0.15. \tag{5.2}$$

The choice of 15% for the magnitude of the perturbation is based on an analysis of the optimal k_{ij} parameters, which vary up to 15% over the temperature range considered here, as will be discussed later. It should be remembered though, for larger temperature ranges, these values could change much more than this.

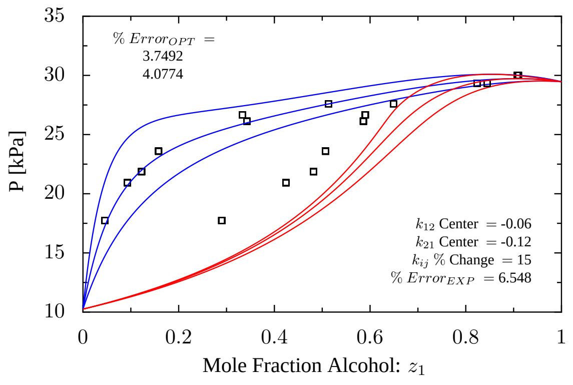
Three datasets are calculated, the first uses optimal parameters, the second uses parameters that are both 15% above optimal, and the third, parameters that are both 15% below. Each plot presents the optimal parameters, the amount of perturbation applied, and two errors. The first error, labeled $\%\ Error_{EXP}$, shows the average of the percent error between simulated values and experimental data, as before. The second error, $\%\ Error_{OPT}$, describes the difference between the data calculated using perturbed binary interaction pairs and the data calculated using optimal pairs, as shown in Equation (5.3):

$$\%\ Error_{OPT} = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{P_{i,pert} - P_{i,opt}}{P_{i,opt}}\right| * 100. \tag{5.3}$$

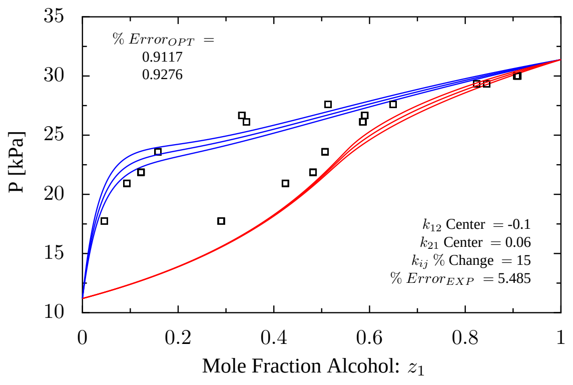
Figure 34 shows a selection of the results of this study. The remaining plots can be found in Appendix F. All of these errors are also presented in Table 9 for easy comparison. As



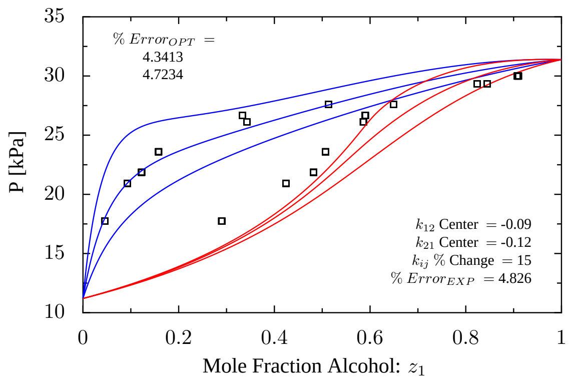
(a) Equation Set = 00010



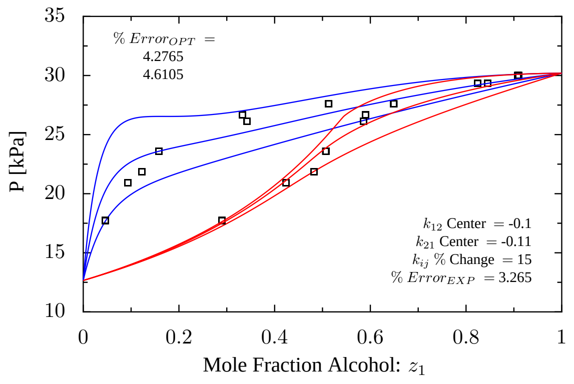
(b) Equation Set = 02010



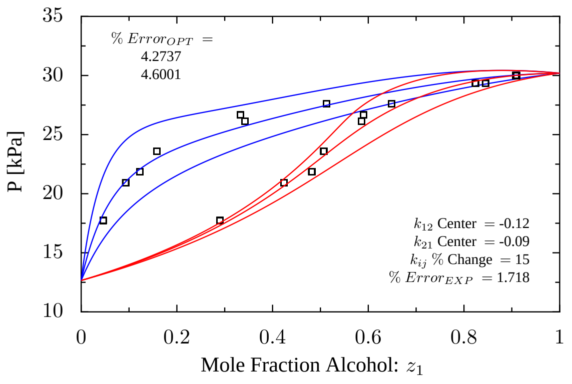
(c) Equation Set = 10010



(d) Equation Set = 12010



(e) Equation Set = 21010



(f) Equation Set = 23010

Figure 20: VLE results using perturbed k_{ij} pairs for varying equation combinations.

described previously, increasing the complexity of the combining rule improves performance compared to experimental data. However, it is evident from this analysis, this only holds true if the optimal binary interaction parameters are known. When the parameters are perturbed, the error introduced between the data calculated with optimal values and that calculated with perturbed values increases with the complexity of the combining rule. For example, the experimental error, $\%Error_{EXP}$ of the best performing equation combination, 23010, is 1.718. However, the total error, $\%Error_{EXP} + \%Error_{OPT}$, is $\sim$ 6.155. When compared to the an equation set with the same equation of state, but a simple combining rule, such as 20010, the total error is seen to be only $\sim$ 3.954. It can therefore be concluded that, although with optimal k_{ij} parameters set 23010 performs best, with only a %15 perturbation of those values, 20010 becomes more accurate. This behavior indicates that when limited experimental data is available, it might actually be better to choose a combining rule that is less affected by the accuracy of the fitting parameters. Using a simple combining rule will generate at least $\%\ Error_{EXP}$, but the calculated data will remain near this deviation, even with inaccurate binary interaction pairs. Conversely, with inadequate experimental data to optimize binary interaction parameters, it becomes harder to predict whether the data simulated with complex combining rules will even lie close to a certain error. In the following section, the meaning of limited experimental data will be quantitatively defined through the examination of VLE data created using k_{ij} pairs optimized at different experimental conditions.

5.3 A QUANTITATIVE EXPLANATION OF LIMITED EXPERIMENTAL DATA

The previous section showed that by perturbing the k_{ij} values, simple combining rules may provide more accurate results than the complex ones. However, it is difficult to see the relationship between perturbed binary interaction parameters and the availability of experimental data. To address this, another standard VLE simulation was run in which the k_{ij} pairs used were determined by optimizing the experimental data at conditions differing from

Table 9: Errors associated with binary interaction parameter choice for various equation combinations at $T = 323.65K$.

Equation Combination	% $Error_{EXP}$	% $AverageError_{OPT}$	Total Error
00010	9.204	0.681	9.885
01010	9.189	4.204	13.393
02010	6.548	3.913	10.461
03010	6.472	3.877	10.349
10010	5.485	0.920	6.405
11010	5.466	4.679	10.145
12010	4.826	4.532	9.358
13010	4.732	4.459	9.191
20010	3.265	0.689	3.954
21010	3.265	4.444	7.709
22010	1.721	4.320	6.041
23010	1.718	4.437	6.155

those in the simulation. For example, the VLE simulation was run at a temperature of 343.15 K. However, the binary interaction parameters used were the optimal pairs determined at different temperatures. This is a common occurrence in the practical implementation of VLE modeling as very often data exists at one temperature and is used to determine optimal binary interaction parameters, but it is the data set of a different temperature that one is interested in predicting. As can be seen in Table 10, as the distance between the simulation temperature and the temperature used to determine optimal binary interaction parameters, T_{OPT}, is increased, the experimental errors increase much more significantly for the complex equation combinations. For example, if a researcher is interested in simulating data at $T = 343.15K$ but only has data for $T = 298.15K$, the best performing equation combination is 20010, not 23010. This also highlights the importance of not just considering the equation of state or the combining rule, but taking into account the entire set of equations, together. In this instance, for example, with the RKS equation, the best choice for the researcher to make would still be combining rule 3 (VL) but with the PR and PRSV equations, combining rule 0 (A) performs best.

In the following section, the perturbation idea is extended and the performance of the VLE simulation is considered for a wide range of binary interaction parameter values.

5.4 ANALYSIS OF A BINARY INTERACTION PARAMETER MESH

If a large range of binary interaction parameters is now considered, maps can be created that show the behavior of the VLE simulation as the parameters change. To do this, both k_{12} and k_{21} are allowed to vary from -1 to 1, independently, with a step-size of 0.01. This range, corresponding to 40,401 pairs, is chosen because a k_{ij} value large than 1 is physically unrealistic, causing the mixture parameter to become negative. The lower bound is selected by symmetry and proved adequate for the determination of a complete profile. A numerical step is performed for each pair, at each available experimental data point. This step, as described in the previous chapter, calculates the bubble and dew point at each experimental value. The errors between the simulated and experimental data are then found and an

Table 10: Average experimental errors calculated from VLE simulations performed at T = 343.15 K using optimal binary interaction parameters from different temperatures.

Equation Set	$\%ERROR_{EXP}$ calculated for T_{OPT} =				
	343.15 K	333.75 K	323.65 K	313.15 K	298.15 K
00010	6.174	6.174	6.819	6.388	6.252
01010	6.168	6.168	7.249	7.249	6.285
02010	3.6	4.525	5.625	4.525	5.699
03010	3.987	3.987	5.217	4.525	4.525
10010	3.696	3.696	4.419	3.901	3.696
11010	3.723	3.723	5.557	4.586	4.586
12010	3.085	4.329	5.202	4.329	5.202
13010	2.986	3.484	4.701	4.701	6.073
20010	2.722	2.844	3.135	3.135	2.722
21010	2.693	2.693	3.791	3.791	3.791
22010	0.678	1.946	3.098	3.098	6.015
23010	0.803	1.648	3.804	3.378	5.158

average error over all of the available experimental data at the fixed temperature is produced. This process is described in the brief section of code shown in Figure 21, which is written for an artificially small dataset for clarity.

The result of the analysis can be visualized in a number of ways. First, the entire dataset can be considered. This corresponds to a $3D$ surface where each (k_{12}, k_{21}) pair has some average percent deviation associated with it, as shown in Figures 22 and 23 with percent deviation expressed on a log axis, for a simple and a complex combining rule, respectively. While this gives an interesting picture of the entire dataset, it is difficult to see the most important data, those binary interaction parameters that yield the most accurate results. This is because some of the percent deviations being shown are artificially large values used to incorporate k_{ij} pairs that do not converge in the numerical routine. Figures 24 and 25 show the same plots, but with a linear axis and a dataset that is limited to those values of $k_{ij}s$ that produce data with an average percent deviation less than 100%. This gives a better view of the nature of the combining rules. It can be seen that there is a fairly large range of values for which the simple combining rules produce good results, while only a small portion of the parameter space produces good data when a complex combining rule is used. This is further highlighted when $2D$ maps of this same data is considered. Figures 26 and 27 show the entire datasets and Figures 28 and 29 show the truncated sets. From this view, the complex combining rule is shown to produce good results for a very tight range, whereas the simple combining rule actually seems to have a linear correlation between k_{12} and k_{21}, which will be described further in the following section.

These figures show that, as one might predict, the simpler combining rule gives more flexibility but the more complex rule gives greater accuracy. This can be quantified by considering the number of k_{ij} pairs that yield data better than a given percent deviation. Because 40,401 pairs are analyzed, a log plot is shown of the full dataset in addition to a linear plot of a partial set. This analysis is shown in Figure 30. Each of these plots shows the range of combining rules for a single equation of state. A quadratic mixing rule for the a parameter and a linear mixing rule for the b parameter are used for all of these figures and the simulation is performed at $T = 323.65K$. These plots highlight the trade-off that one must make when choosing a combining rule. To obtain very accurate VLE data, it is

```
k12_arr = [-1,0,1]      ! An array of all possible values of k12
k21_arr = [-1,0,1]      ! All possible values of k21
P_arr = [2,3,4]         ! Experimental pressure values
x_arr = [0.1,0.5,0.9]   ! Experimental composition values that
                        ! correspond to bubble point data.
y_arr = [0.2,0.3,0.8]   ! Experimental composition values that
                        ! correspond to dew point data.
k12_loop: do i=1,3

   k21_loop: do j=1,3

      k12 = k12_arr(i)   ! Set the kij pairs
      k21 = k21_arr(j)
      BubbleError=0      ! Initialize the errors for this pair
      DewError=0

      exp_loop: do k=1,3 ! Step through each experimental datapoint
         Pexp = P_arr(k)
         x = x_arr(k)
         y = y_arr(k)
         call VLEB(T,x,PB)  ! Call the VLE program at the given
                            ! values of T and x. The calculated
                            ! value of P is the bubble point.
         call VLED(T,y,PD)  ! Next, calculate dew point
                            ! pressure.

         bubble_percent_error = abs(PB - Pexp)/Pexp*100
         dew_percent_error = abs(PD - Pexp)/Pexp*100

         BubbleError = BubbleError + bubble_percent_error
         DewError = DewError + dew_percent_error
      end do exp_loop

      AveError_B = BubbleError/3
      AveError_D = DewError/3
      Average_Percent_Error = (AveError_B + Ave_Error_D)/2

   end do k21_loop
end do k12_loop
```

Figure 21: A skeleton of the code used to determine the average percent error for each binary interaction parameter pair.

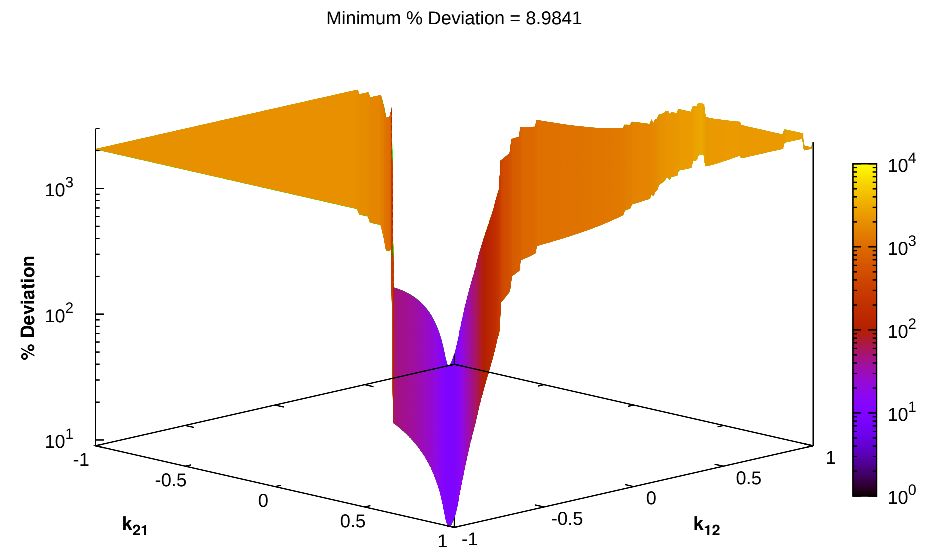


Figure 22: The $3D$ surface showing average percent deviation for all k_{ij} pairs in the set $[-1, 1]$ using a simple combining rule at $T = 323.65K$.

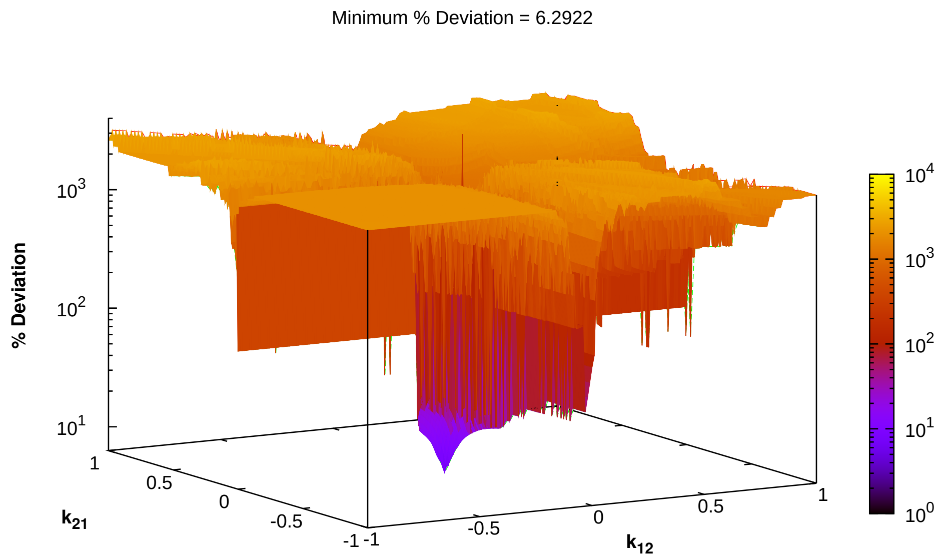


Figure 23: The $3D$ surface showing average percent deviation for all k_{ij} pairs in the set $[-1, 1]$ using a complex combining rule at $T = 323.65K$.

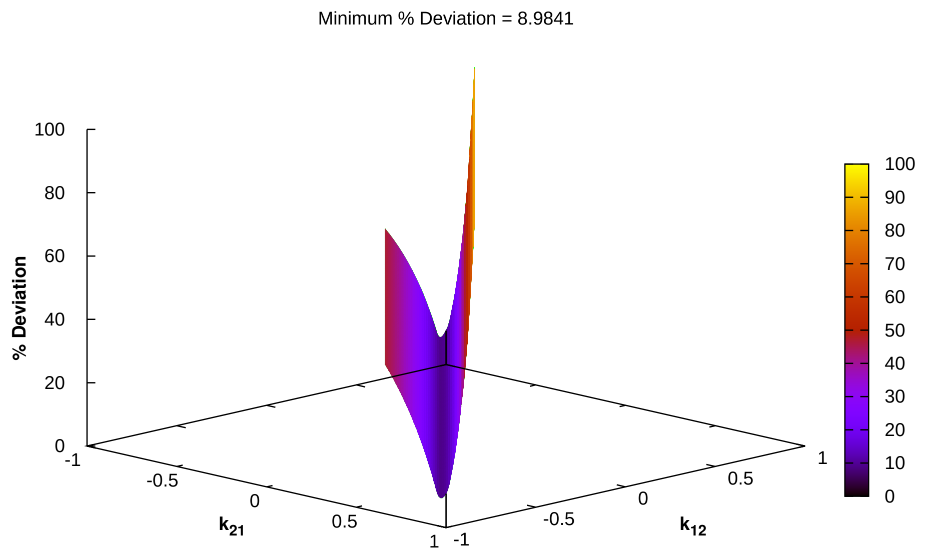


Figure 24: The 3D surface showing average percent deviation for the k_{ij} pairs producing data with less than 100% error using a simple combining rule at $T = 323.65K$.

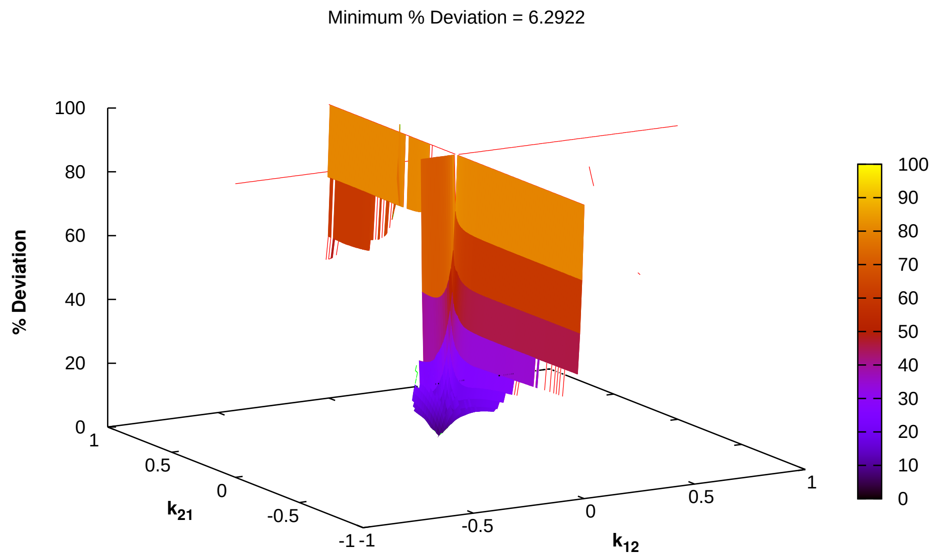


Figure 25: The 3D surface showing average percent deviation for the k_{ij} pairs producing data with less than 100% error using a complex combining rule at $T = 323.65K$.

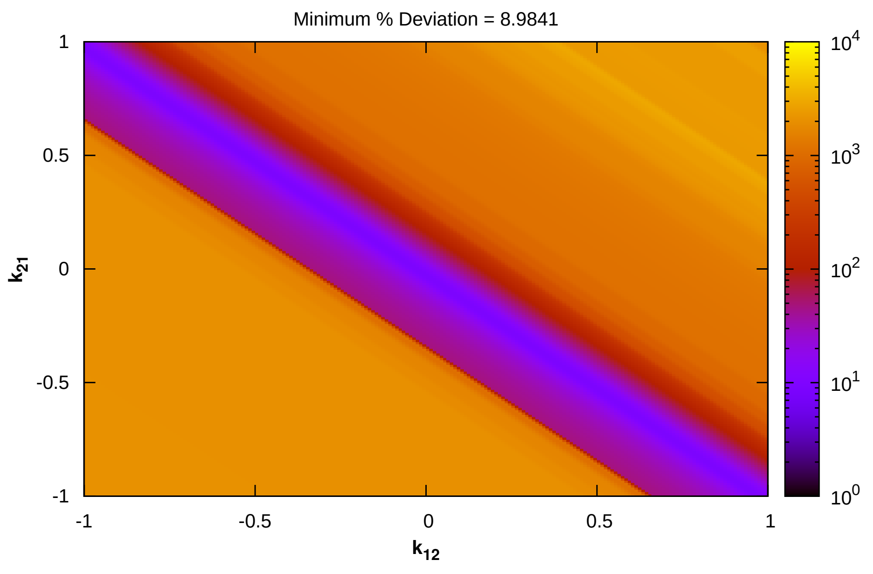


Figure 26: The $2D$ map of average percent deviation for all k_{ij} pairs in the set $[-1, 1]$ using a simple combining rule at $T = 323.65K$.

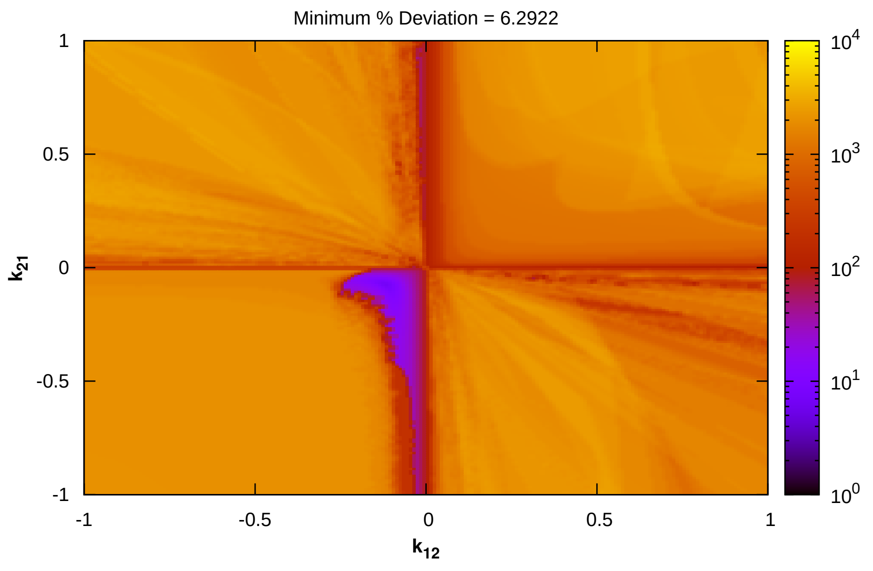


Figure 27: The $2D$ map of average percent deviation for all k_{ij} pairs in the set $[-1, 1]$ using a complex combining rule at $T = 323.65K$.

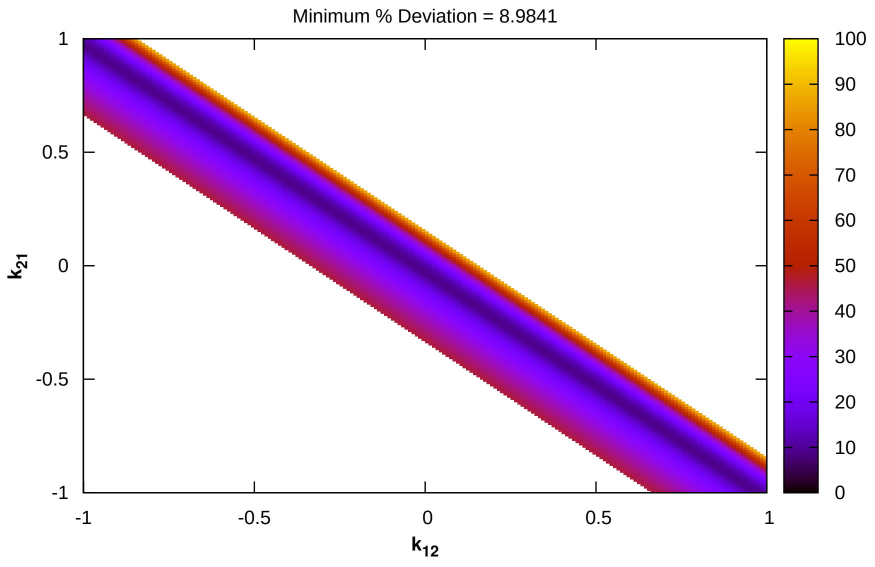


Figure 28: The $2D$ map of average percent deviation for the k_{ij} pairs producing data with less than 100% error using a simple combining rule at $T = 323.65K$.

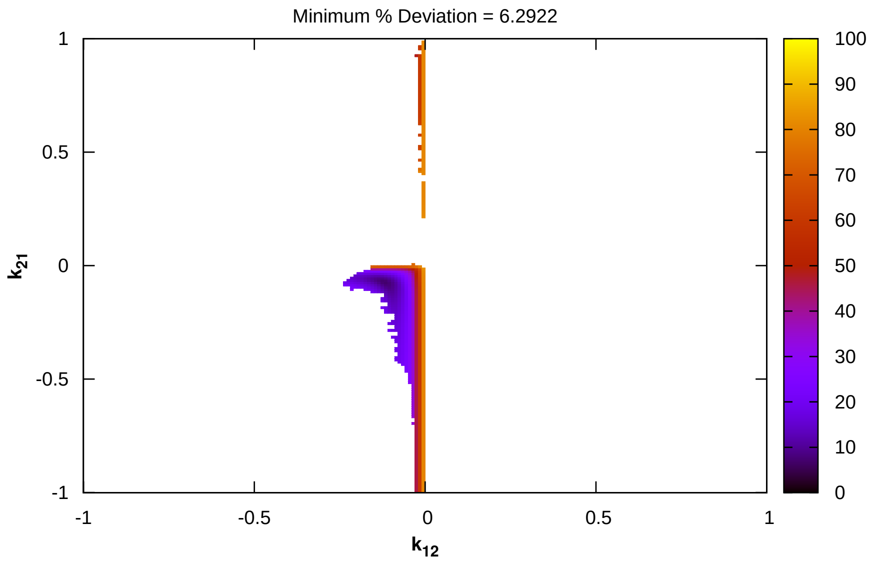


Figure 29: The $2D$ map of average percent deviation for the k_{ij} pairs producing data with less than 100% error using a complex combining rule at $T = 323.65K$.

necessary to use a complex combining rule. However, once a certain error is surpassed, the lines in these plots cross each other, signifying that simpler combining rules rely less heavily on the accuracy of k_{ij} data, and therefore, a wider range of k_{ij} pairs yield better results.

5.5 ANALYSIS OF THE OPTIMIZED BINARY INTERACTION PARAMETERS

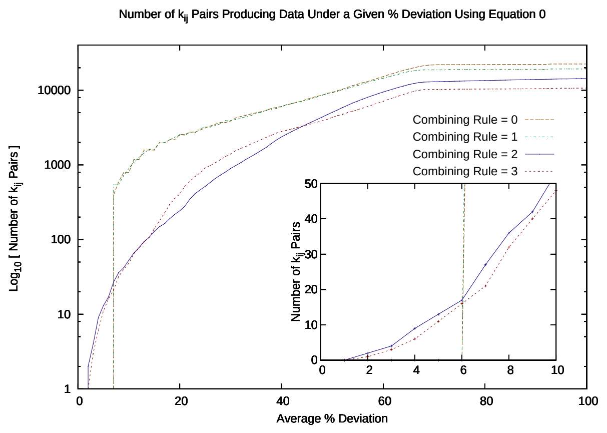
When the maps above are considered, it seems that the optimal binary interaction parameters for some of the combining rules show a correlation. This effect can be better understood by focusing on only those k_{ij} pairs that give the best performance. As it turns out, when this is done, entire lines of pairs are revealed for the simple arithmetic and conventional combining rules, and single pairs arise for the more complex Margules and van Laar form combining rules. Figure 31 shows these pairs graphically. When a single pair is reported, the values of the pair are shown, and when a line of pairs exists, the linear fit corresponding to that line is given. If all of the simple combining rule equation combinations are considered, it turns out that the linear fits are all exact and all have a slope of -1. This leads to the important result that only one value is actually required as a binary interaction parameter for the simple combining rules. Instead of requiring separate values for k_{12} and k_{21}, by specifying a y-intercept, b, a value for k_{12} can be chosen arbitrarily and using Equation 5.4, k_{21} is automatically determined.

$$k_{21} = -k_{12} + b \tag{5.4}$$

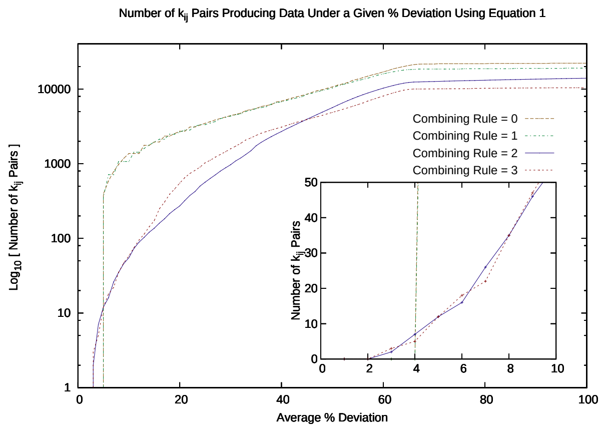
Furthermore, contrary to common practice, because the slope is -1 and the y-intercepts are often non-zero, it is not sufficient, for this specific system, to simply equate the binary interaction parameters [49, 50]. In fact, for the arithmetic combining rule, it is better practice to set:

$$k_{21} = -k_{12}, \tag{5.5}$$

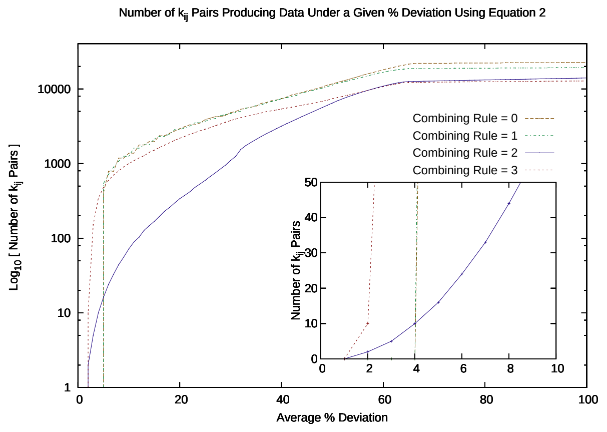
if a simplification is required.



(a) EQ = 0

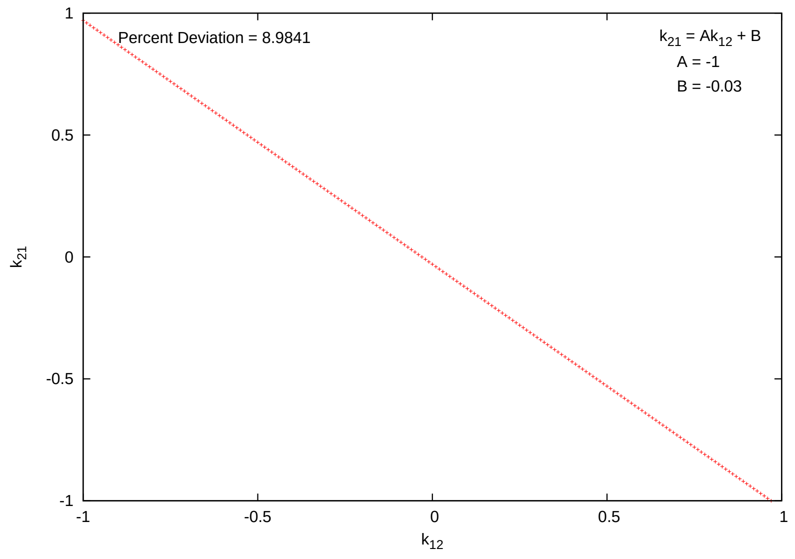


(b) EQ = 1

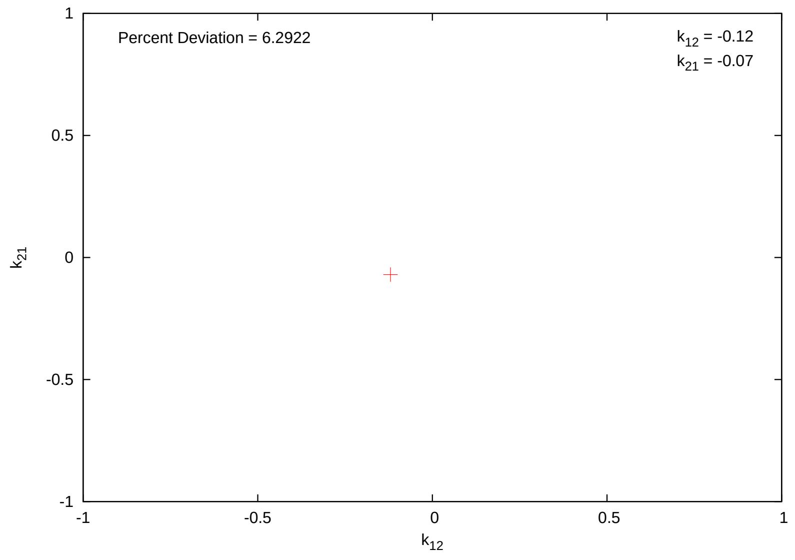


(c) EQ = 2

Figure 30: The number of k_{ij} pairs that produce results below an average percent deviation.



(a) Equation Set = 00010



(b) Equation Set = 03010

Figure 31: Optimal binary interaction parameter pairs, calculated average percent deviation from experimental data, and exact linear fits to reported data.

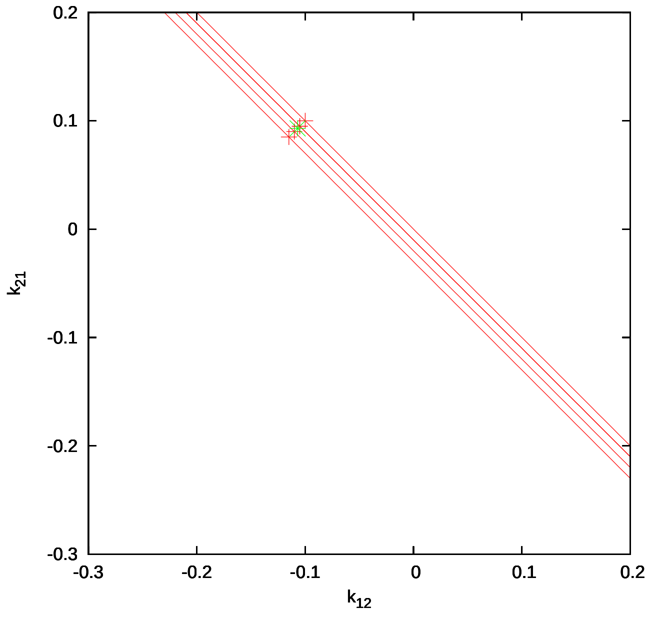
5.6 EFFECT OF TEMPERATURE ON THE OPTIMAL BINARY INTERACTION PARAMETERS

Finally, the goal of predicting VLE data relies on the ability to use binary interaction parameters obtained for one set of experimental data and apply them to the simulations of data at different conditions. As discussed previously, while experimental datasets can be used to determine binary interaction parameters, if those parameters change significantly with temperature, they are not useful for data prediction at other temperatures. In order to better understand the effect of temperature on these values, the optimized k_{ij} pairs for each equation set are plotted together for various temperatures. Two examples are shown in Figure 32, for two different equation combinations. The remaining combinations are shown in Appendix H. In order to quantify the scatter of these values, the averages of the optimal pairs are calculated, which are also shown on the figures. For the simple combining rules with multiple optimal pairs, single pairs must be chosen first for each temperature. This is done by choosing a pair for one temperature and then choosing the pairs that are closest to it on adjacent lines. The average of these chosen pairs is then determined normally.

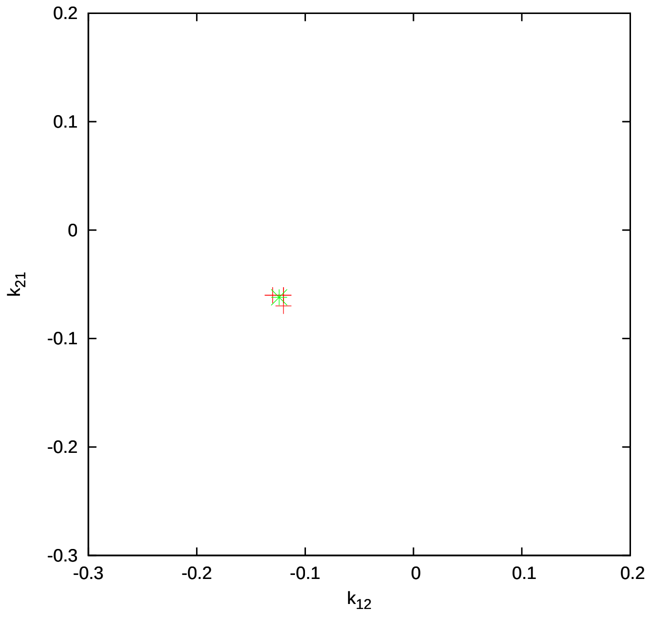
Using these averages, the maximum percent deviations are calculated from:

$$PD_{k_{12},max} = max_i \left(\left| \frac{k_{12,i} - k_{12,ave}}{k_{12,ave}} \right| \right), \tag{5.6}$$

where $k_{12,i}$ represents the k_{12} value optimized at temperature T_i. This is done for each equation combination so that two values, $PD_{k_{12},max}$ and $PD_{k_{21},max}$, can be used as indicators of the deviation that is possible in optimal binary interaction parameters over a temperature range of 45 K. The results of these calculations are shown in Table 11. It is important to note that these deviations can change based on the mean values. For the complex combining rules, this is not important as the binary interaction parameters are fixed. For the simple combining rules, on the other hand, the first parameter is chosen arbitrarily. In this analysis, the first parameter is chosen as -0.1, in order to match the choice in the perturbation study. In this manner, the values here and those used in the perturbation study can be directly compared. Additionally, this choice keeps the resultant binary interaction parameters on



(a) Equation Set = 00010



(b) Equation Set = 03010

Figure 32: Effect of temperature on the optimal k_{ij} pairs.

the same scale as those that are fixed. Figure 33 shows an average of the maximum percent deviations for a given equation set:

$$PD_{ave} = \frac{PD_{k_{12},max} + PD_{k_{21},max}}{2}, \tag{5.7}$$

and highlights the importance of considering the entire combination of equations. Each cluster of bars in the figure represents an equation of state, with the bars in the cluster representing the combining rules, increasing in complexity from left to right. It is clear that the temperature does significantly affect the optimal k_{ij} values that are calculated. However, it is most interesting to note that the effect of temperature varies among both combining rules and equations of state. One combining rule may yield the best performance for one equation of state, but another rule may be preferred for a different equation. For the Redlich-Kwong-Soave and the Peng-Robinson-Stryjek-Vera equations of state for example, the Margules form combining rule is significantly more dependent on changes in temperature. However, with the Peng-Robinson equation of state, all of the combining rules produce similar deviations, with the Margules form the least dependent.

When considered together, all of these points highlight the importance of taking care when choosing which equation of state, mixing rules, and combining rules should be used to simulate VLE data. If the desire is simply to fill in missing composition information for a dataset with large amounts of accurate experimental data, it is best to choose complex equations of state and combining rules that will give the most accurate correlations. However, if one is trying to predict the values for a unique set of conditions, it may be better to choose a simpler set of equations that, while not as accurate, will more reliably give a solution within a determinable error.

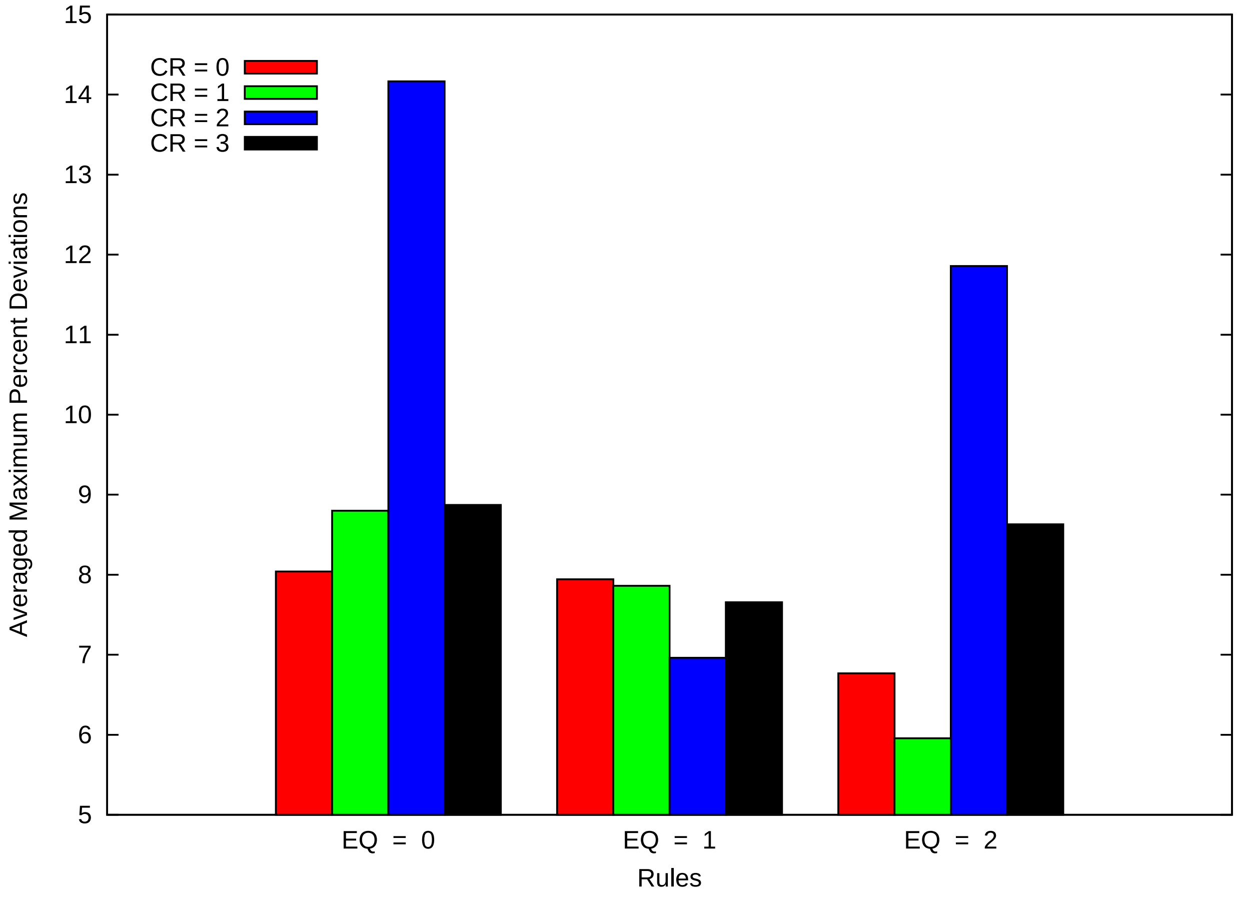


Figure 33: Comparison of the averaged maximum percent deviations of k_{ij} pairs optimized at different temperatures for each equation set.

Table 11: Maximum percent deviation for each binary interaction parameter for each equation set over varying temperature.

Equation Combination	$k_{12,ave}$	$k_{k21,ave}$	$PD_{k_{12},max}$	$PD_{k_{21},max}$
00010	-0.107	0.093	7.5	8.6
01010	-0.102	-0.082	7.8	9.8
02010	-0.05	-0.12	20	8.3
03010	-0.124	-0.062	4.8	12.9
10010	-0.103	0.077	6.8	9.1
11010	-0.097	-0.107	8.2	7.5
12010	-0.084	-0.118	7.1	6.8
13010	-0.12	-0.086	8.3	7.0
20010	-0.101	0.079	5.9	7.6
21010	-0.096	-0.106	6.3	5.7
22010	-0.078	-0.12	15.3	8.3
23010	-0.126	-0.08	4.8	12.5

6.0 CONCLUSIONS

Vapor-liquid equilibrium (VLE) data is used in many applications, from the design of chemical separation plants, to the optimization of heat transfer in cooling devices for electronics. In general, obtaining this data experimentally is a time-consuming process, and, as a result, numerical methods have been developed to allow their simulation. The method described in this work relies on the thermodynamics of equilibrium states, where the fugacities of different phases become equal. The fugacity of a system, whose mathematical derivation was described in Chapter 1, was shown to be a theoretical pressure at which a substance that is a non-ideal gas takes on the properties that an ideal gas would have at the actual pressure of that system. The fugacity coefficient, developed to simplify the mathematics, was defined as the fugacity divided by the pressure. This coefficient relies on a relationship between the temperature, pressure, and volume of the system, called an equation of state. While many forms of equations of state exist, they can be divided into theoretical equations, whose bases lie in statistical thermodynamics and intermolecular potential theory, and empirical equations, which are primarily correlative relationships. While theoretical equations have the ability to better predict VLE data, their use is extremely computationally intensive and the simulation of large, complex systems is often prohibitively difficult. As a result, empirical equations of state are used to a much greater extent. However, to increase the usability of these equations, it is necessary to realize their limitations in terms of correlative versus predictive ability. This work has presented a method which can be utilized to improve the understanding of these equations and aid in the decision process used to choose the optimal sets of equations for VLE modeling.

The numerical process used in this study was based on the $\phi - \phi$ method, as described in Chapter 4. The code was written in *Fortran 90* utilizing the message passing interface, where beneficial, for the parallelization of data execution across multiple CPUs. Laguerre's Method was used for finding the roots of the equations of state, in order to guarantee convergence, and a seeding process was implemented to provide the best initial guess possible and decrease computation times. Additionally, a technique was developed to dynamically update the manner in which the temperature or pressure is modified during the iterative routine. The algorithm also takes into account the possibility of divergence, working to avoid mathematical poles, and decrease the time spent on attempting the calculation of non-convergent data points. Finally, the dependence of the routine on various numerical parameters was detailed.

A comprehensive look at the role of equations of state, combining rules, and mixing rules on the performance of vapor-liquid equilibrium simulations was undertaken. The Redlich-Kwong-Soave, Peng-Robinson, and Peng-Robinson-Stryjek-Vera equations of state were considered, along with linear and quadratic mixing rules. Arithmetic, conventional, Margules form, and van Laar form combining rules were used to determine the cross-parameters in the mixing rule equations. These types represent a small subset of the vast library of equations that is available, but the method developed, as detailed in Chapter 3, was created to preserve generality in such a way that many different equation combinations can be implemented with relative ease. A mixture of water and ethanol was chosen for this analysis at a range of temperatures and pressures where adequate experimental data existed. Standard VLE bubble point and dew point curves were first modeled using binary interaction parameters specifically optimized for the equations and conditions of interest. It was shown that as the complexities of the equations of state and combining rules increase, the accuracy of data correlation improves greatly, as expected.

The optimal binary interaction parameters were subsequently perturbed and VLE data was calculated using these new values. From this analysis it became apparent that while complex combining rules provide the best data correlation, the use of poor binary interaction parameters introduces significant error into the simulation. The simple combining rules, on the other hand, while yielding a greater error with optimal parameters, show much less deviation in accuracy when those parameters are perturbed. This is explicitly shown through the simulation of VLE data at a temperature of 343.15 K using binary interaction parameters optimized with experimental data at different temperatures. It can be concluded from this study that when the distance between the conditions of interest and the conditions where experimental data are available increases sufficiently, simple combining rules provide more accurate prediction of VLE data than their complex counterparts. For example, for the Peng-Robinson and the Peng-Robinson-Stryjek-Vera equations of state, it was shown that the use of binary interaction parameters optimized with data at 323.65 K was enough to make the error between simulated and experimental data at 343.15 K larger for the van Laar type combining rule than for the arithmetic rule.

In order to see the full effect of binary interaction parameters on VLE modeling, a mesh of pairs from -1 to 1 for k_{12} and k_{21} was created such that 40,401 different pairs could be used, independently, for the simulation of data. 3-D surface plots and 2-D maps of the average percent deviation from experimental data were created to show the nature of the influence of these parameters. These plots shed light on the conclusions that were developing in the previous studies. It was shown that the simple combining rules yield surfaces with broad valleys of minimum deviation, whereas complex combining rules come to sharp points. This difference was quantified by comparing the number of pairs that converged under a certain deviation for different combining rules. This depiction made it clear that although complex combining rules are capable of providing higher accuracy, a trade-off point exists for each equation of state where complex combining rules have fewer pairs than simple combining rules that yield data within a certain error. Specifically, for the RKS equation of state, around 20 binary interaction pairs give an error of less than 7% for the Margules form and van Laar form combining rules. For the arithmetic and conventional rules, on the other hand, nearly 550 pairs give less than 7% error.

The 2-D maps were subsequently processed to extract only the best performing binary interaction pairs for each equation set. It was determined that this corresponded to a single pair for complex combining rules and an entire line of pairs for the simple rules. When a fit was applied the line of pairs, it was discovered that an exact linear correlation existed and that the equation for the fit always had a slope of -1, regardless of whether the arithmetic or conventional combining rule was used, and regardless of the equation of state chosen. This leads to the interesting result that, for the simple combining rules considered here, only one experimental fitting parameter is actually required for the determination of the cross-term in the equation of state parameters. Given only a y-intercept value, b, the value of the first binary interaction parameter can be set arbitrarily, with the other being fully defined by:

$$k_{21} = -k_{12} + b. \tag{6.1}$$

This is contrary to the requirement of complex combining rules, for which two independent binary interaction parameters must be specified. Additionally, because it was found that the slope of this function is negative and that the value of the y-intercept is very often non-zero, the common practice of simply fixing both binary interaction parameters to the same value is not the best manner in which to calculate accurate data for the system under consideration. An analysis of the y-intercepts averaged over temperature, shown in Table 12, addresses the difference between the arithmetic and the conventional combining rules. From these values, the argument could be made that the simplification:

$$k_{21} = -k_{12}, \tag{6.2}$$

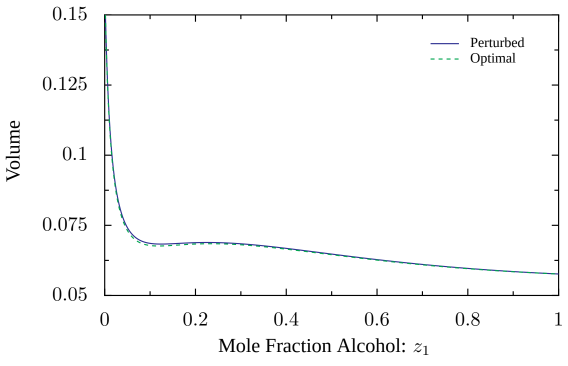
while insufficient for the conventional combining rule, could be used to generate reasonable results using the arithmetic combining rule. This approach, which corresponds to a collapse down to a linear mixing rule, would likely be a better method than simply equating the two parameters.

Table 12: The y-intercept of the linear fit for simple combining rules.

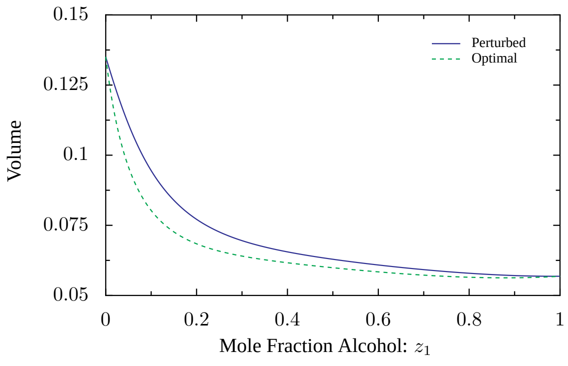
Equation Combination	y-intercept (b)
00010	-0.014
01010	-0.184
10010	-0.026
11010	-0.204
20010	-0.022
21010	-0.202

To determine the underlying cause of the differing effects of binary interaction parameters on different equations, the predicted volumes were considered. In an analysis which mimics the binary interaction parameter perturbation study presented in Chapter 5.2, the volume, instead of the pressure or temperature, was plotted against composition using both optimal and perturbed parameters. Those results, shown in Figures 34a and 34b, highlight the effect on the volume of a parameter perturbation with a simple combining rule and complex one, respectively. The dashed lines show the calculated volume using optimal parameters, and the solid lines show the perturbed case. The differing predictions of volume give a possible insight into why the complex combining rules are more strongly dependent on the accuracy of the binary interaction parameters. The quadratic mixing rule considered in both of these simulations is sensitive to differences in volumes between molecules in a system. In fact, a system with a large disparity in volume between its components simply cannot be accurately modeled with this rule [51]. The water-ethanol system under consideration maintains an actual volume ratio within the usable limitation of this mixing rule and the simple combining rule, while not predicting a completely accurate volume, calculates a volume within this range, even with non-optimal binary interaction parameters. However, it is possible that the complex combining rules do not always predict volumes that fall within this range when provided poor binary interaction parameters. It is thought that this is a potential reason for the maintenance of reasonable accuracy with simple combining rules and the abrupt failure of complex combining rules, with parameter perturbation. As a result, if this prediction holds, it is likely that related systems will show results similar to the water-ethanol system considered in this work. In support of this claim, preliminary results for a water-butanol system have been analyzed and it has been observed that many of the same patterns described in Chapter 5 hold.

In conclusion, it can be stated definitively that the choice of equations used to model vapor liquid equilibrium data should be based on the availability and quality of experimental data at or near the parameters of interest. Furthermore, it is not sufficient to choose the best equation of state and the best combining rule independently. These decisions should be made together, keeping in mind that some combining rules will be more predictive than others, depending on the chosen equation of state. This analysis has chosen a relatively



(a) Equation Set = 00010



(b) Equation Set = 23010

Figure 34: Calculated volumes using perturbed k_{ij} pairs.

small set of equations to provide an insight into these effects. However, the model has been created such that a large number of equation combinations can be simulated with relative ease. In this manner, equation sets can be compared to determine which is optimal for any given simulation.

Future work in this area would benefit from the inclusion of multiple mixtures and a larger number of equations of state, combining rules, and mixing rules. Using the analysis developed here, a greater understanding of the different interactions between equations could be obtained. It is possible that, as with the arithmetic combining rule shown here, additional relationships could be introduced that relate the binary interaction parameters for other combining rules. This would decrease the experimental fitting parameters required for VLE modeling and give better approximations for researchers when data is unavailable or unreliable. Finally, performing such an analysis for a larger set of equations could lead to the development of a decision tree that would provide a detailed method in which the best equation combinations could be determined for any given set of experimental data, greatly increasing the usability of empirical VLE modeling.

APPENDIX A

DERIVATIVES OF MIXING RULE TERMS

Derivatives of mixing rules used in this work are presented below as they are an important aspect of the calculation of fugacity coefficients. Note that for mixtures, the mole fractions are more conveniently expressed in terms of the number of moles using the following expressions:

$$z_i = \frac{n_i}{n}, \tag{A.1}$$

$$n = \sum_{i}^{N} n_i. \tag{A.2}$$

Various forms of each equation are shown for clarity, but all are based on the calculation of a binary mixture. Also, at times the derivatives are expressed with what seem like arbitrary multiplications by $\frac{1}{n}$. This is done to make substitution into the general fugacity coefficient expression more convenient.

A.1 LINEAR MIXING RULE

$$\zeta = z_1\zeta_{11} + z_2\zeta_{22} = \frac{n_1}{n}\zeta_{11} + \frac{n_2}{n}\zeta_{22} \tag{A.3}$$

$$n\zeta = n_1\zeta_{11} + n_2\zeta_{22} \tag{A.4}$$

$$n^2\zeta = nn_1\zeta_{11} + nn_2\zeta_{22} \tag{A.5}$$

$$n^2\zeta^2 = n_1^2\zeta_{11}^2 + 2n_1n_2\zeta_{11}\zeta_{22} + n_2^2\zeta_{22}^2 \tag{A.6}$$

$$\frac{\partial\left(n\zeta\right)}{\partial n_1} = \zeta_{11} \tag{A.7}$$

$$\frac{\partial\left(n\zeta\right)}{\partial n_2} = \zeta_{22} \tag{A.8}$$

$$\frac{1}{n}\frac{\partial\left(n^2\zeta\right)}{\partial n_1} = \zeta_{11}\left(z_1 + 1\right) + z_2\zeta_{22} \tag{A.9}$$

$$\frac{1}{n}\frac{\partial\left(n^2\zeta\right)}{\partial n_2} = z_1\zeta_{11} + \zeta_{22}\left(z_2 + 1\right) \tag{A.10}$$

$$\frac{1}{n}\frac{\partial\left(n^2\zeta^2\right)}{\partial n_1} = 2z_1\zeta_{11}^2 + 2z_2\zeta_{11}\zeta_{22} \tag{A.11}$$

$$\frac{1}{n}\frac{\partial\left(n^2\zeta^2\right)}{\partial n_2} = 2z_1\zeta_{11}\zeta_{22} + 2z_2\zeta_{22}^2 \tag{A.12}$$

A.2 QUADRATIC MIXING RULE

$$\begin{aligned}\zeta &= z_1^2\zeta_{11}^2 + z_1z_2\zeta_{12} + z_1z_2\zeta_{21} + z_2^2\zeta_{22}^2 \\ &= \frac{n_1^2}{n^2}\zeta_{11}^2 + \frac{n_1n_2}{n^2}\zeta_{12} + \frac{n_1n_2}{n^2}\zeta_{21} + \frac{n_2^2}{n^2}\zeta_{22}^2 \qquad \text{(A.13)} \\ n\zeta &= \frac{n_1^2}{n}\zeta_{11}^2 + \frac{n_1n_2}{n}\zeta_{12} + \frac{n_1n_2}{n}\zeta_{21} + \frac{n_2^2}{n}\zeta_{22}^2 \qquad \text{(A.14)} \\ n^2\zeta &= n_1^2\zeta_{11}^2 + n_1n_2\zeta_{12} + n_1n_2\zeta_{21} + n_2^2\zeta_{22}^2 \qquad \text{(A.15)}\end{aligned}$$

$$\begin{aligned}n^2\zeta^2 = {} & \frac{n_1^4}{n^2}\zeta_{11}^2 + \frac{2n_1^3n_2}{n^2}\zeta_{11}\zeta_{12} + \frac{2n_1^3n_2}{n^2}\zeta_{11}\zeta_{21} + \frac{2n_1^2n_2^2}{n^2}\zeta_{11}\zeta_{22} + \frac{n_1^2n_2^2}{n^2}\zeta_{12}^2 \\ & + \frac{n_1^2n_2^2}{n^2}\zeta_{21}^2 + \frac{2n_1^2n_2^2}{n^2}\zeta_{12}\zeta_{12} + \frac{2n_1n_2^3}{n^2}\zeta_{12}\zeta_{22} + \frac{2n_1n_2^3}{n^2}\zeta_{21}\zeta_{22} + \frac{n_2^4}{n^2}\zeta_{22}^2 \qquad \text{(A.16)}\end{aligned}$$

$$\frac{\partial(n\zeta)}{\partial n_1} = z_1\zeta_{11}(2-z_1) + \left(z_2^2\zeta_{12} + z_1z_2n\frac{\partial\zeta_{12}}{\partial n_1}\right) + \left(z_2^2\zeta_{21} + z_1z_2n\frac{\partial\zeta_{21}}{\partial n_1}\right) - z_2^2\zeta_{22} \qquad \text{(A.17)}$$

$$\frac{\partial(n\zeta)}{\partial n_2} = -z_1^2\zeta_{11} + \left(z_1^2\zeta_{12} + z_1z_2n\frac{\partial\zeta_{12}}{\partial n_2}\right) + \left(z_1^2\zeta_{21} + z_1z_2n\frac{\partial\zeta_{21}}{\partial n_2}\right) + z_2\zeta_{22}(2-z_2) \qquad \text{(A.18)}$$

$$\frac{1}{n}\frac{\partial(n^2\zeta)}{\partial n_1} = z_2\zeta_{12} + z_2\zeta_{21} + 2z_1\zeta_{11} + z_1z_2n\frac{\partial\zeta_{12}}{\partial n_1} + z_1z_2n\frac{\partial\zeta_{21}}{\partial n_1} \qquad \text{(A.19)}$$

$$\frac{1}{n}\frac{\partial(n^2\zeta)}{\partial n_2} = z_1\zeta_{12} + z_1\zeta_{21} + 2z_2\zeta_{22} + z_1z_2n\frac{\partial\zeta_{12}}{\partial n_2} + z_1z_2n\frac{\partial\zeta_{21}}{\partial n_2} \qquad \text{(A.20)}$$

$$\begin{aligned}\frac{1}{n}\frac{\partial\left(n^2\zeta^2\right)}{\partial n_1} &= z_1^3\left(4-2z_1\right)\zeta_{11}^2 - 2z_2^4\zeta_{22}^2 + 4z_1z_2^2\left(1-z_1\right)\zeta_{11}\zeta_{22} \\ &+ 2z_1z_2^2\left(1-z_1\right)\left(\zeta_{12}^2+2\zeta_{12}\zeta_{21}+\zeta_{21}^2\right) \\ &+ z_1^2z_2^2\left(2\zeta_{12}n\frac{\partial\zeta_{12}}{\partial n_1}+2\zeta_{12}n\frac{\partial\zeta_{21}}{\partial n_1}+2\zeta_{21}n\frac{\partial\zeta_{12}}{\partial n_1}+2\zeta_{21}n\frac{\partial\zeta_{21}}{\partial n_1}\right) \\ &+ 2\zeta_{11}\left[\left(\zeta_{12}+\zeta_{21}\right)z_1^2z_2\left(3-2z_1\right)+z_1^3z_2\left(n\frac{\partial\zeta_{12}}{\partial n_1}+n\frac{\partial\zeta_{21}}{\partial n_1}\right)\right] \\ &+ 2\zeta_{22}\left[\left(\zeta_{12}+\zeta_{21}\right)z_2^3\left(1-2z_1\right)+z_1z_2^3\left(n\frac{\partial\zeta_{12}}{\partial n_1}+n\frac{\partial\zeta_{21}}{\partial n_1}\right)\right] \qquad \text{(A.21)}\end{aligned}$$

$$\begin{aligned}\frac{1}{n}\frac{\partial\left(n^2\zeta^2\right)}{\partial n_2} &= z_2^3\left(4-2z_2\right)\zeta_{22}^2 - 2z_1^4\zeta_{11}^2 + 4z_1^2z_2\left(1-z_2\right)\zeta_{11}\zeta_{22} \\ &+ 2z_1^2z_2\left(1-z_2\right)\left(\zeta_{12}^2+2\zeta_{12}\zeta_{21}+\zeta_{21}^2\right) \\ &+ z_1^2z_2^2\left(2\zeta_{12}n\frac{\partial\zeta_{12}}{\partial n_2}+2\zeta_{12}n\frac{\partial\zeta_{21}}{\partial n_2}+2\zeta_{21}n\frac{\partial\zeta_{12}}{\partial n_2}+2\zeta_{21}n\frac{\partial\zeta_{21}}{\partial n_2}\right) \\ &+ 2\zeta_{11}\left[\left(\zeta_{12}+\zeta_{21}\right)z_1^3\left(1-2z_2\right)+z_1^3z_2\left(n\frac{\partial\zeta_{12}}{\partial n_2}+n\frac{\partial\zeta_{21}}{\partial n_2}\right)\right] \\ &+ 2\zeta_{22}\left[\left(\zeta_{12}+\zeta_{21}\right)z_1z_2^2\left(3-2z_2\right)+z_1z_2^3\left(n\frac{\partial\zeta_{12}}{\partial n_2}+n\frac{\partial\zeta_{21}}{\partial n_2}\right)\right] \qquad \text{(A.22)}\end{aligned}$$

APPENDIX B

DERIVATIVES OF COMBINING RULE TERMS

B.1 ARITHMETIC RULE

$$\zeta_{12} = \frac{1}{2}\left(1 - k_{12}\right)\left(\zeta_{11} + \zeta_{22}\right) \tag{B.1}$$

$$\zeta_{21} = \frac{1}{2}\left(1 - k_{21}\right)\left(\zeta_{11} + \zeta_{22}\right) \tag{B.2}$$

$$n\frac{\partial \zeta_{12}}{\partial n_1} = n\frac{\partial \zeta_{21}}{\partial n_1} = 0 \tag{B.3}$$

$$n\frac{\partial \zeta_{12}}{\partial n_2} = n\frac{\partial \zeta_{21}}{\partial n_2} = 0 \tag{B.4}$$

B.2 GEOMETRIC RULE

$$\zeta_{12} = \left(1 - k_{12}\right)\sqrt{\zeta_{11}\zeta_{22}} \tag{B.5}$$

$$\zeta_{21} = \left(1 - k_{21}\right)\sqrt{\zeta_{11}\zeta_{22}} \tag{B.6}$$

$$n\frac{\partial \zeta_{12}}{\partial n_1} = n\frac{\partial \zeta_{21}}{\partial n_1} = 0 \tag{B.7}$$

$$n\frac{\partial \zeta_{12}}{\partial n_2} = n\frac{\partial \zeta_{21}}{\partial n_2} = 0 \tag{B.8}$$

B.3 MARGULES RULE

$$\zeta_{12} = \zeta_{21} = \sqrt{\zeta_{11}\zeta_{22}}\,(1 - z_1 k_{12} - z_2 k_{21}) \tag{B.9}$$

$$n\frac{\partial \zeta_{12}}{\partial n_1} = n\frac{\partial \zeta_{21}}{\partial n_1} = \sqrt{\zeta_{11}\zeta_{22}}\,(-k_{12} z_2 + k_{21} z_2) \tag{B.10}$$

$$n\frac{\partial \zeta_{12}}{\partial n_2} = n\frac{\partial \zeta_{21}}{\partial n_2} = \sqrt{\zeta_{11}\zeta_{22}}\,(k_{12} z_1 - k_{21} z_1) \tag{B.11}$$

B.4 VAN LAAR RULE

$$\zeta_{12} = \zeta_{21} = \sqrt{\zeta_{11}\zeta_{22}}\left(1 - \frac{k_{12}k_{21}}{z_1 k_{12} + z_2 k_{21}}\right) \tag{B.12}$$

$$n\frac{\partial \zeta_{12}}{\partial n_1} = n\frac{\partial \zeta_{21}}{\partial n_1} = \frac{k_{12}k_{21}\sqrt{\zeta_{11}\zeta_{22}}}{(z_1 k_{12} + z_2 k_{21})^2}\,(k_{12} z_2 - k_{21} z_2) \tag{B.13}$$

$$n\frac{\partial \zeta_{12}}{\partial n_2} = n\frac{\partial \zeta_{21}}{\partial n_2} = \frac{k_{12}k_{21}\sqrt{\zeta_{11}\zeta_{22}}}{(z_1 k_{12} + z_2 k_{21})^2}\,(k_{21} z_1 - k_{12} z_1) \tag{B.14}$$

APPENDIX C

SYSTEMS USED FOR COMPUTATIONS

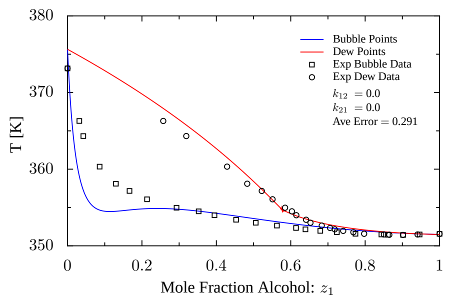
The Dell XPS 9000 PC consists of a 3.06 GHz Intel Core i7-950 Processor. The processor consists of 4 physical cores, each supporting 2 virtual cores, for a total of 8 working cores with 4GB of shared SDRAM.

Warhol, an HP BladeSystem c3000 at the Pittsburgh Supercomputing Center, is comprised of 8 nodes, each made up of 2 Intel E5440 quad-core 2.83 GHz processors, for a total of 64 cores. Each of the nodes use 16 GB of shared memory and communicate using an InfiniBand communications link. More information can be found on the Pittsburgh Supercomputing Center's website [52].

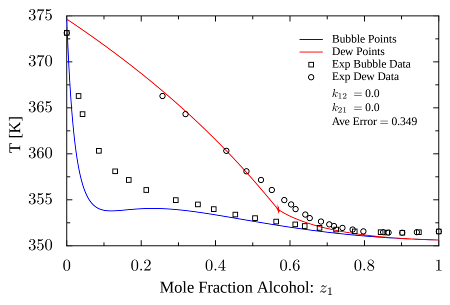
APPENDIX D

FIXED PRESSURE T-X PLOTS FOR VARIOUS EQUATION COMBINATIONS

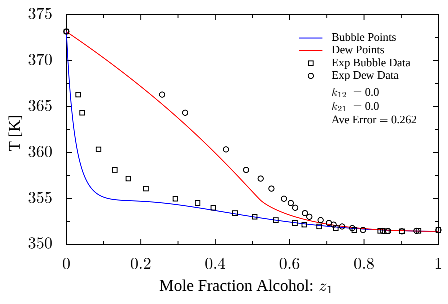
See Section 5.1 for an explanation of these figures.



(a) EOS = 0 (RKS)

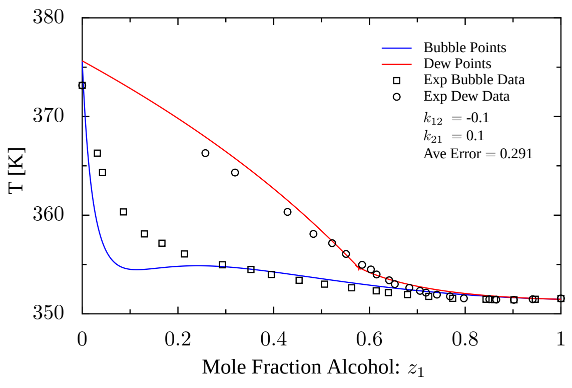


(b) EOS = 1 (PR)

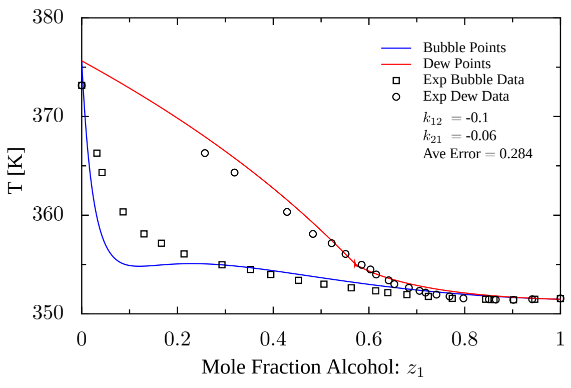


(c) EOS = 2 (PRSV)

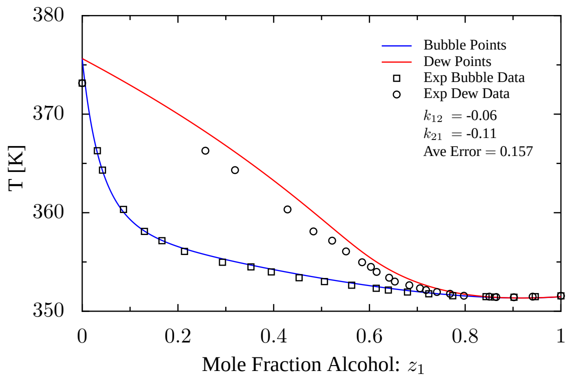
Figure 35: VLE Results at $P = 101325\ Pa$ for varying equation of state.



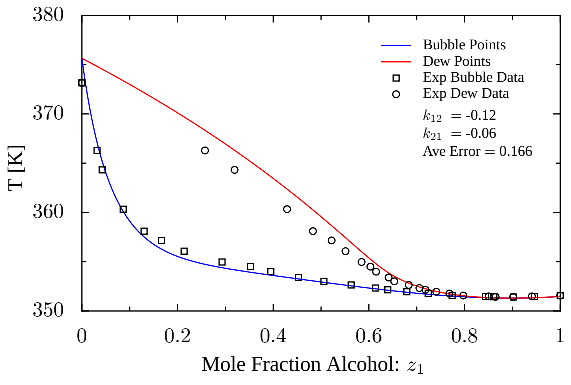
(a) CR = 0 (A)



(b) CR = 1 (C)

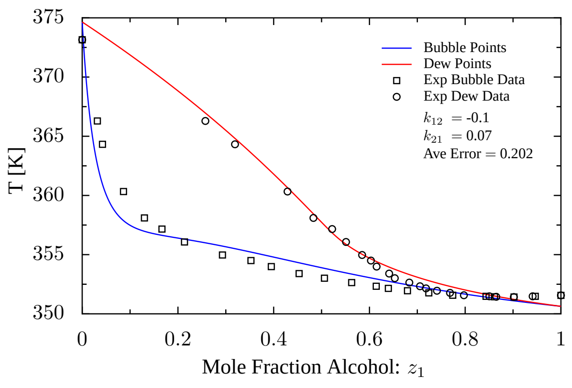


(c) CR = 2 (M)

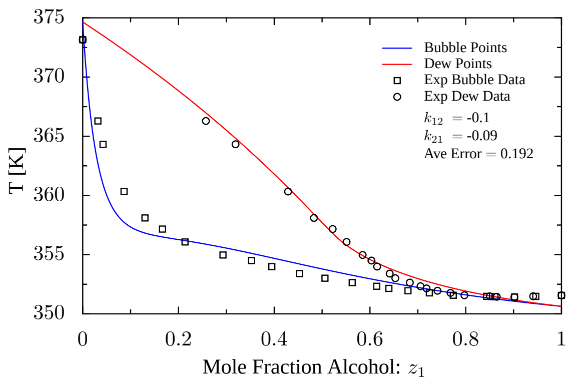


(d) CR = 3 (VL)

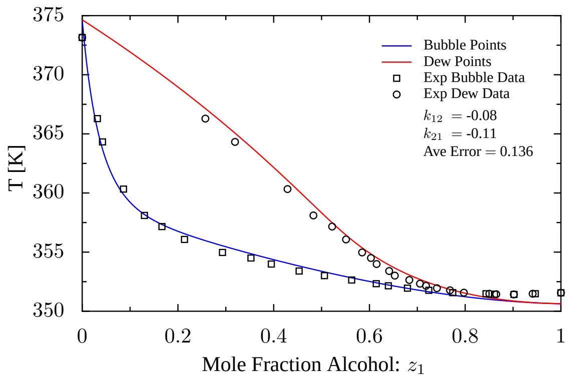
Figure 36: VLE Results at $P = 101325$ Pa for varying combining rules with EOS = 0.



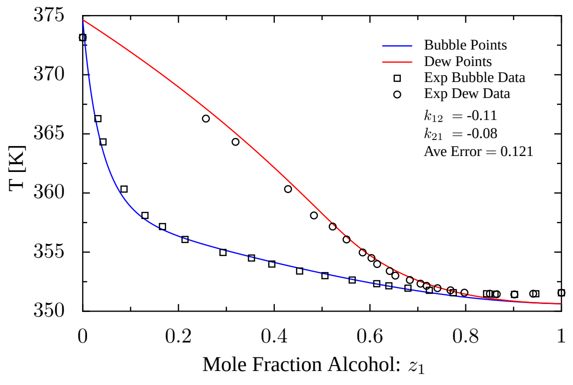
(a) CR = 0 (A)



(b) CR = 1 (C)

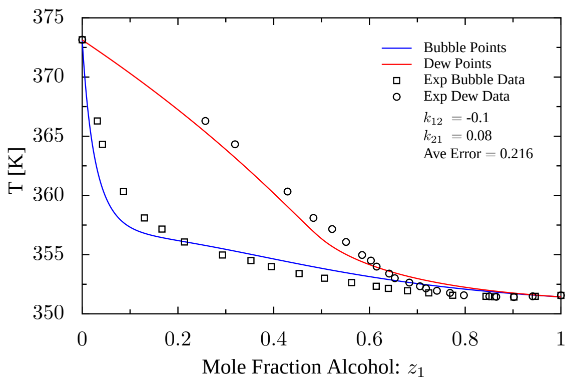


(c) CR = 2 (M)

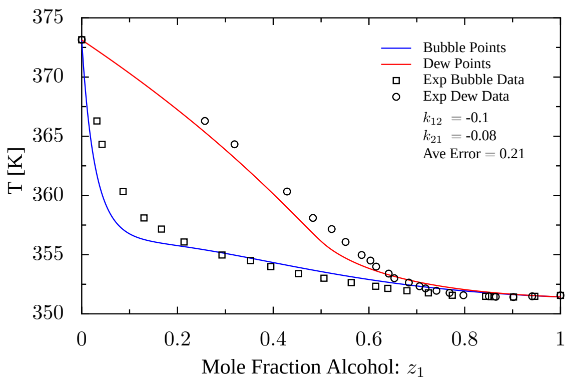


(d) CR = 3 (VL)

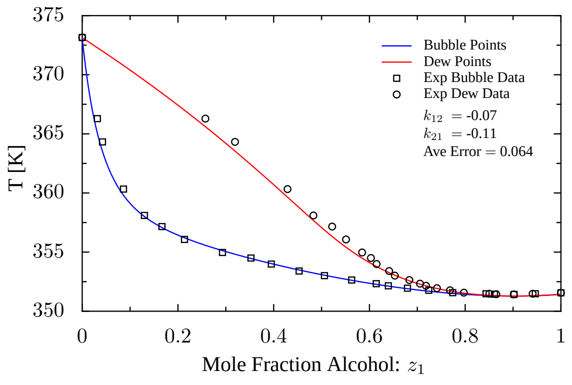
Figure 37: VLE Results at $P = 101325\ Pa$ for varying combining rules with EOS = 1.



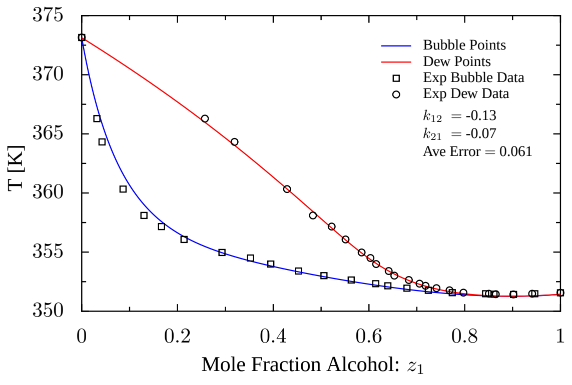
(a) CR = 0 (A)



(b) CR = 1 (C)



(c) CR = 2 (M)



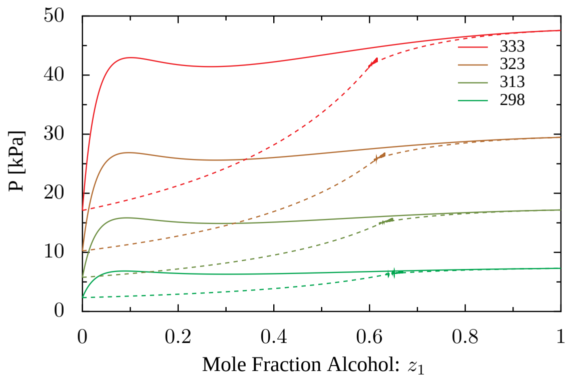
(d) CR = 3 (VL)

Figure 38: VLE Results at $P = 101325$ Pa for varying combining rules with EOS = 2.

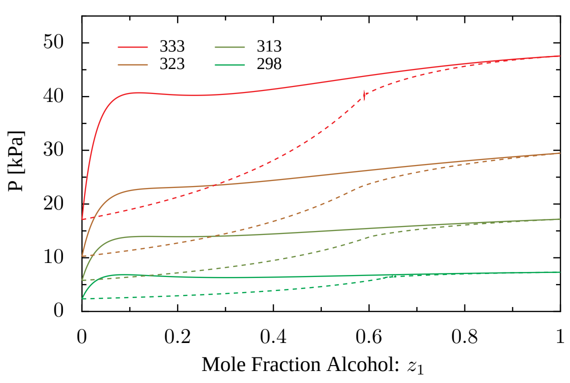
APPENDIX E

MULTIPLE TEMPERATURE P-X PLOTS FOR VARIOUS EQUATION COMBINATIONS

See Section 5.1 for an explanation of these figures.

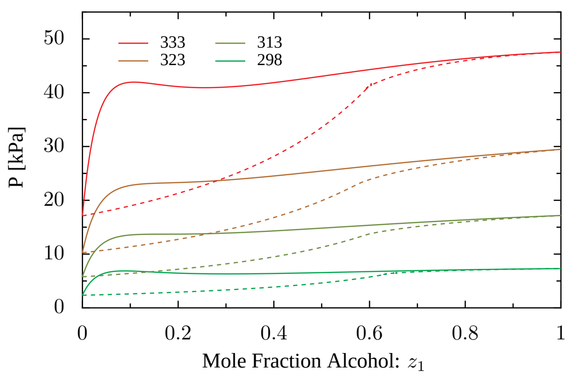


(a) Equation Set = 00000

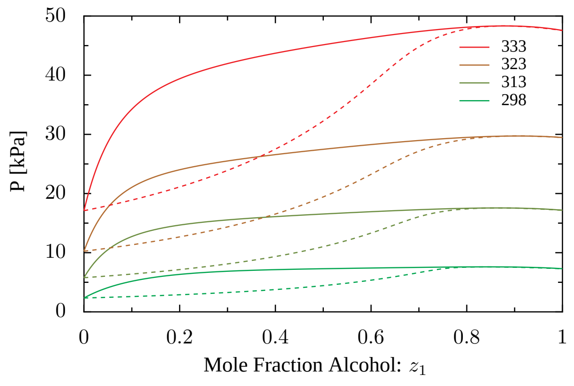


(b) Equation Set = 00010

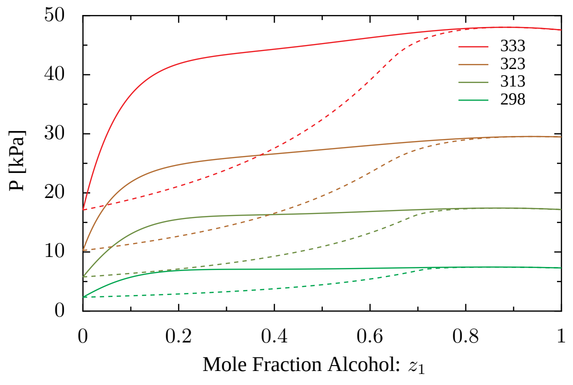
Figure 39: VLE results at varying temperatures for EOS = 0.



(c) Equation Set = 01010

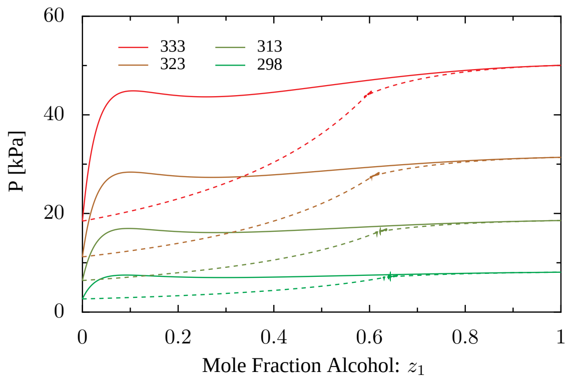


(d) Equation Set = 02010

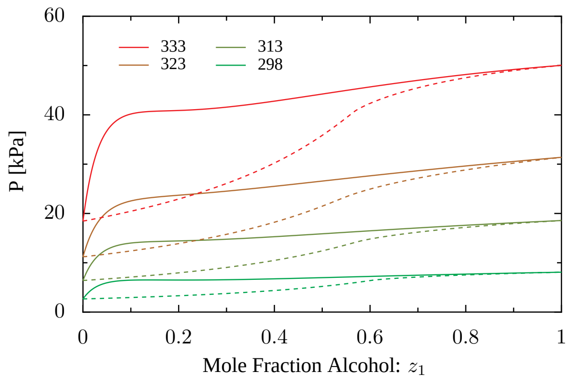


(e) Equation Set = 03010

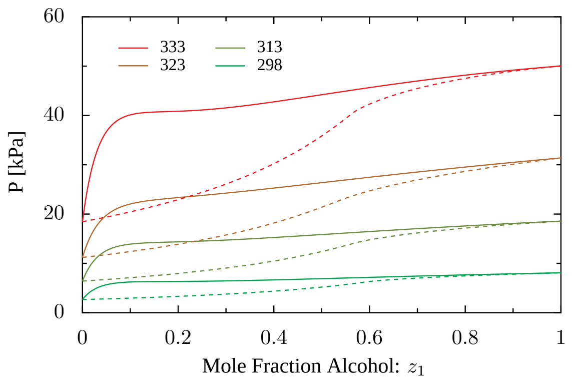
Figure 39: VLE results at varying temperatures for EOS = 0.



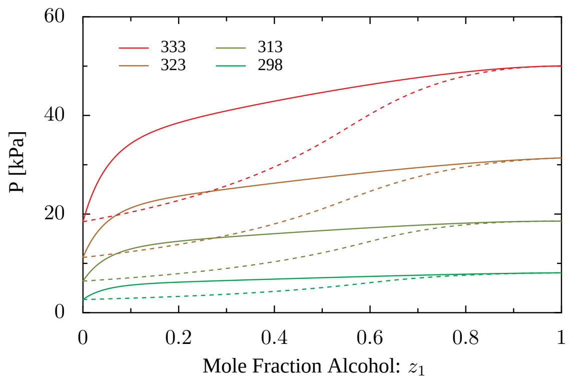
(a) Equation Set = 10000



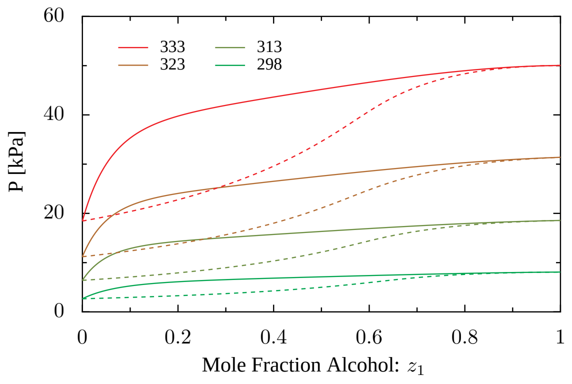
(b) Equation Set = 10010



(c) Equation Set = 11010

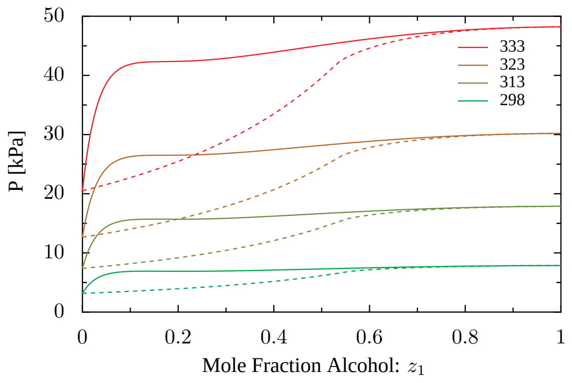


(d) Equation Set = 12010

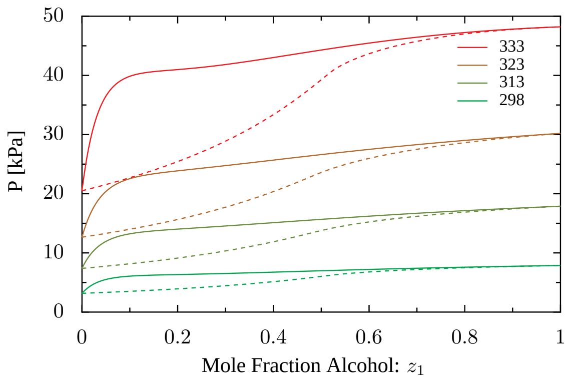


(e) Equation Set = 13010

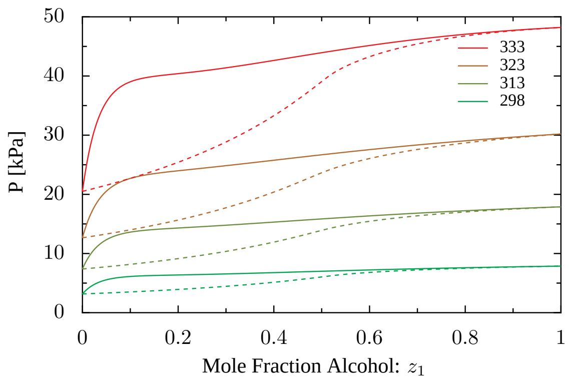
Figure 40: VLE results at varying temperatures for EOS = 1.



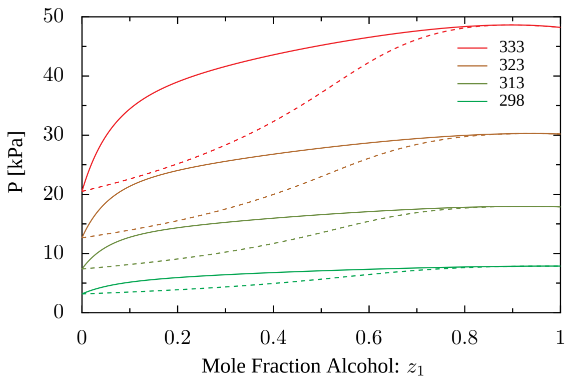
(a) Equation Set = 20000



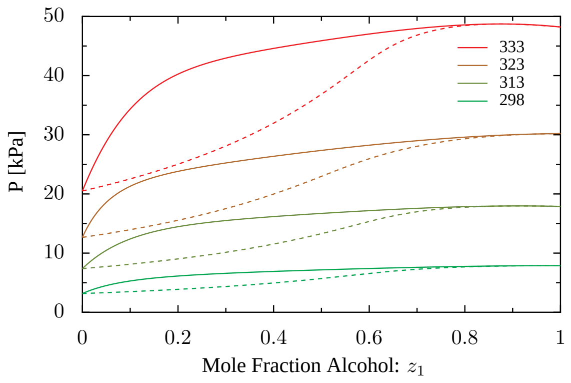
(b) Equation Set = 20010



(c) Equation Set = 21010



(d) Equation Set = 22010



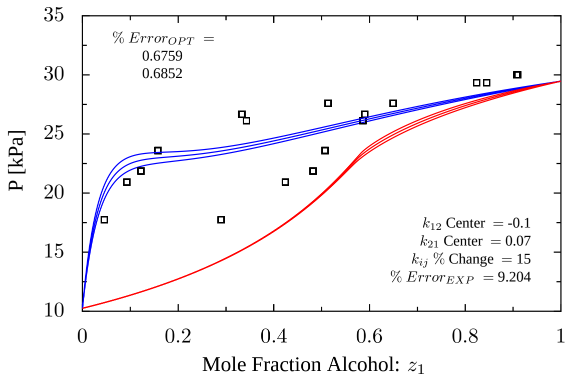
(e) Equation Set = 23010

Figure 41: VLE results at varying temperatures for EOS = 2.

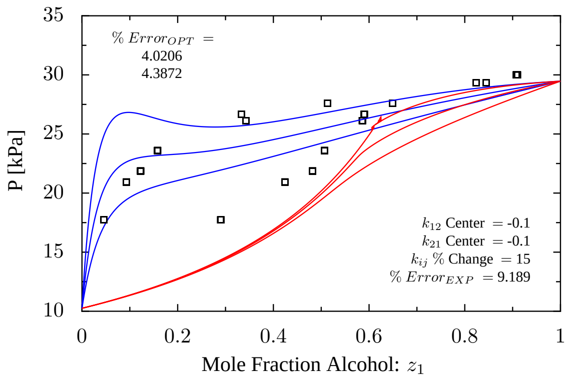
APPENDIX F

PERTURBATION OF THE BINARY INTERACTION PARAMETER

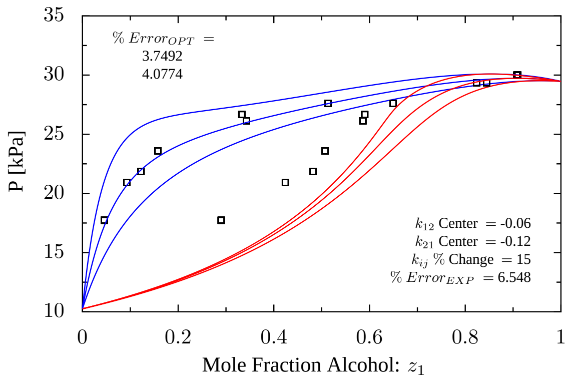
See Section 5.2 for an explanation of these figures.



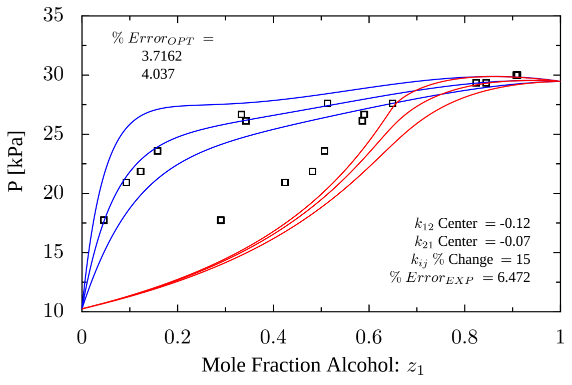
(a) Equation Set = 00010



(b) Equation Set = 01010

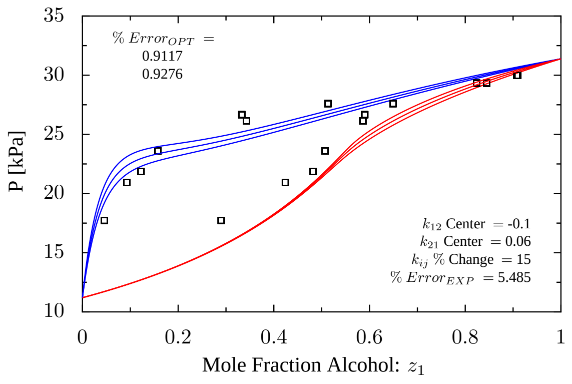


(c) Equation Set = 02010

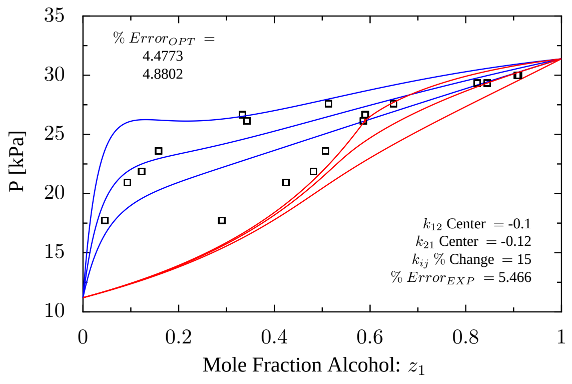


(d) Equation Set = 03010

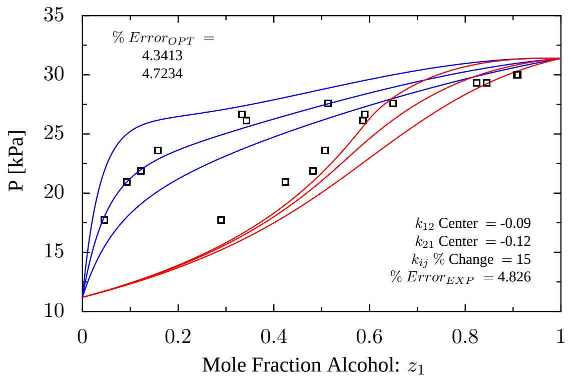
Figure 42: VLE results using perturbed binary interaction parameters for EOS = 0.



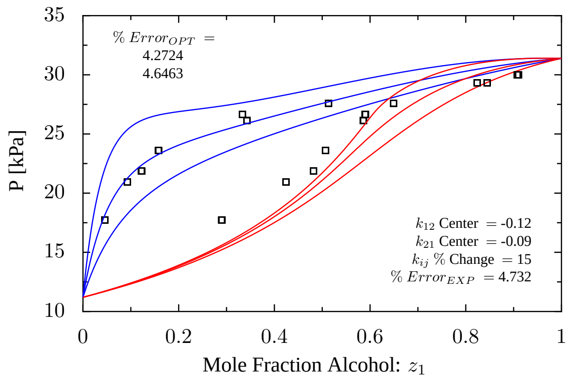
(a) Equation Set = 10010



(b) Equation Set = 11010

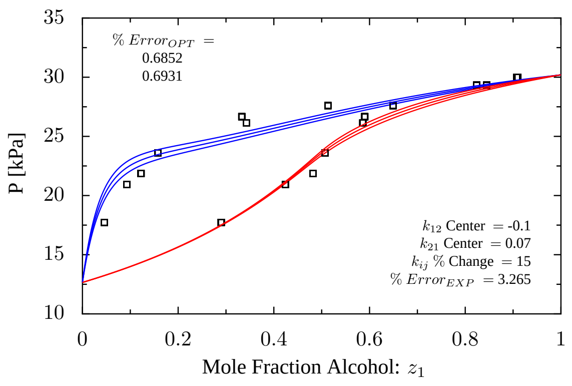


(c) Equation Set = 12010

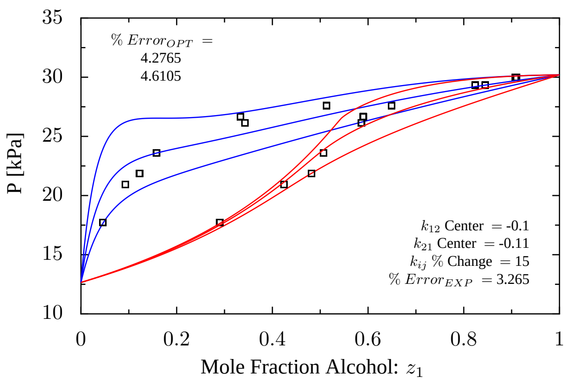


(d) Equation Set = 13010

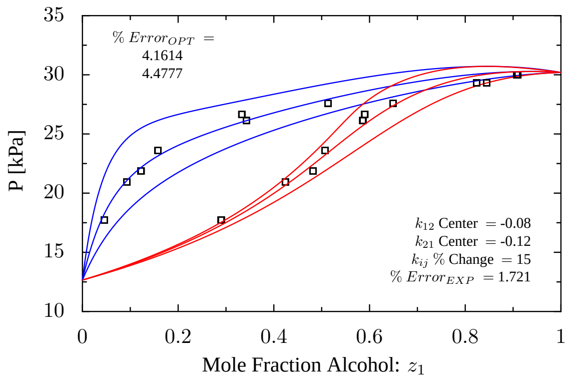
Figure 43: VLE results using perturbed binary interaction parameters for EOS = 1.



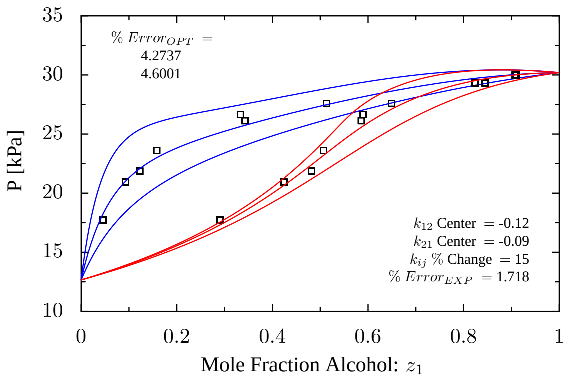
(a) Equation Set = 20010



(b) Equation Set = 21010



(c) Equation Set = 22010



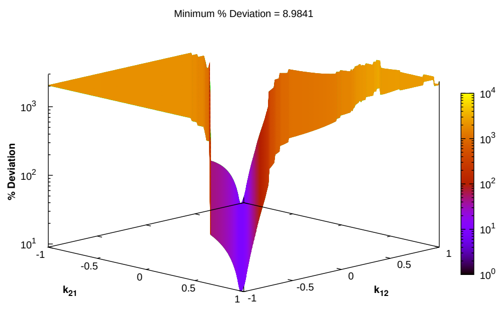
(d) Equation Set = 23010

Figure 44: VLE results using perturbed binary interaction parameters for EOS = 2.

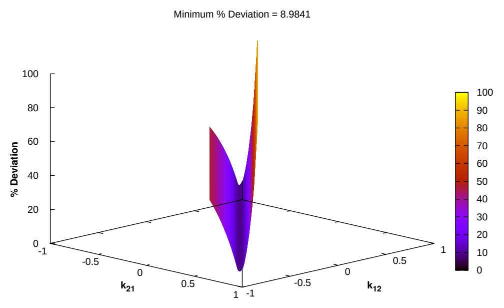
APPENDIX G

COMPLETE ANALYSIS OF THE BINARY INTERACTION PARAMETERS

See Section 5.4 for an explanation of these figures.

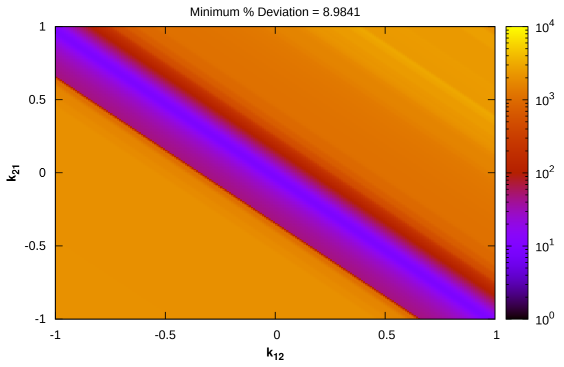


(a) Full Surface

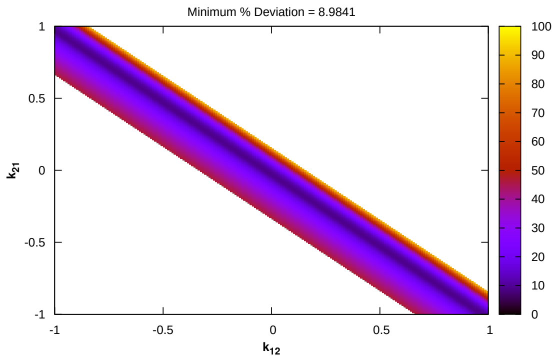


(b) Partial Surface

Figure 45: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 00010.

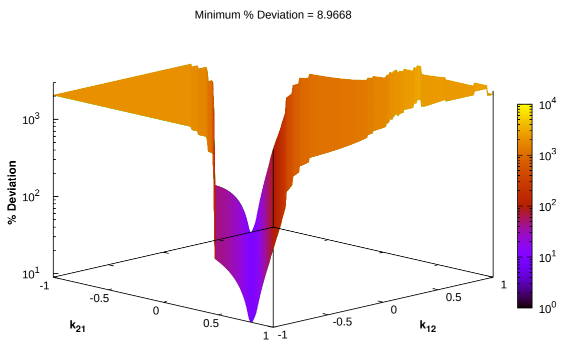


(c) Full Map

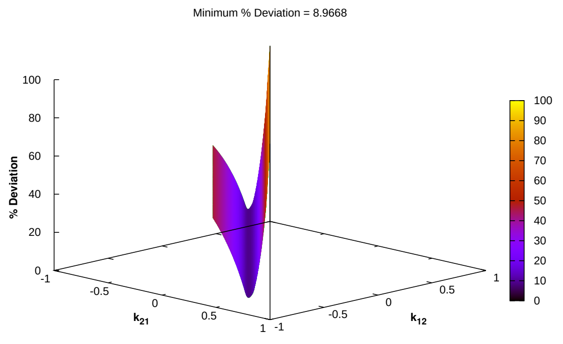


(d) Partial Map

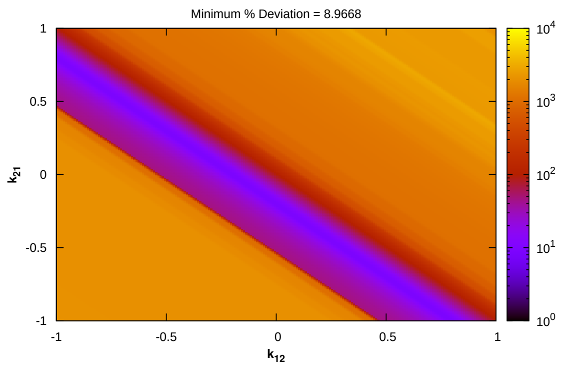
Figure 45: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 00010.



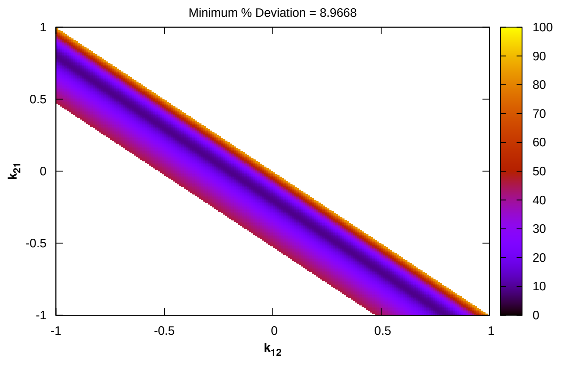
(a) Full Surface



(b) Partial Surface

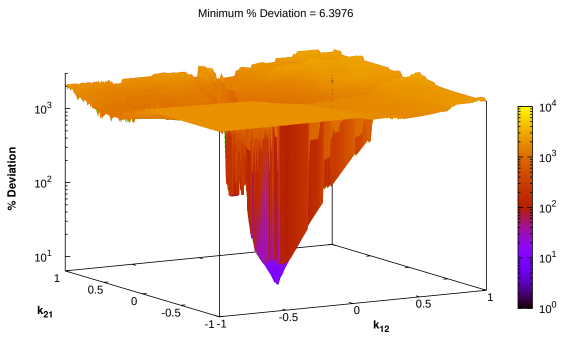


(c) Full Map

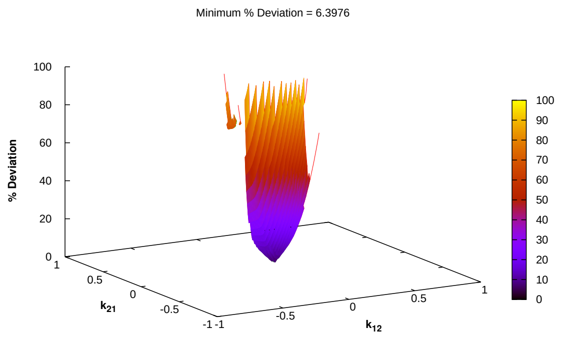


(d) Partial Map

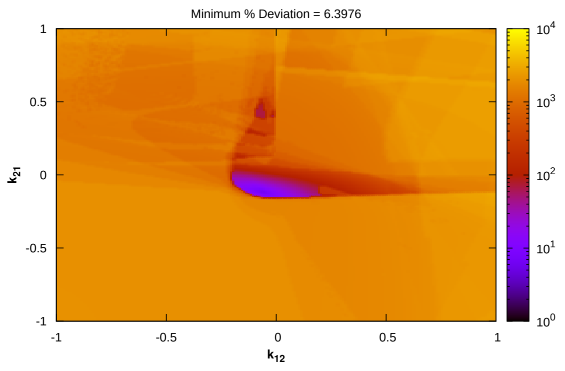
Figure 46: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 01010.



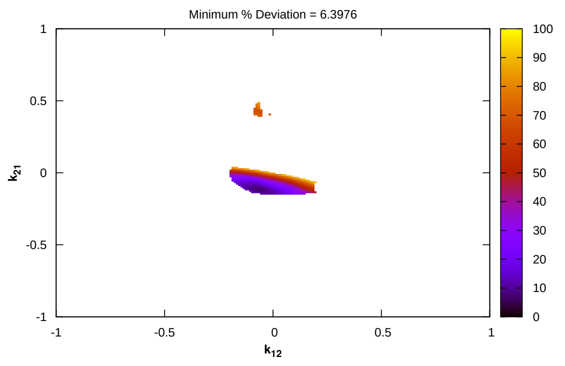
(a) Full Surface



(b) Partial Surface

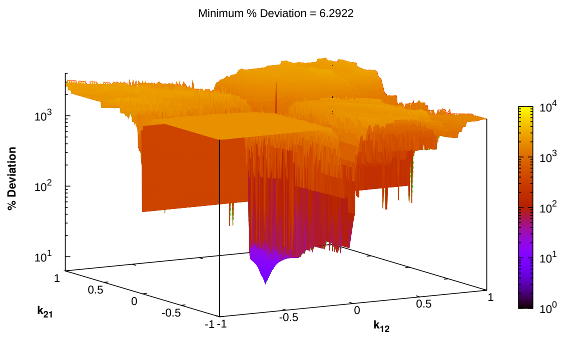


(c) Full Map

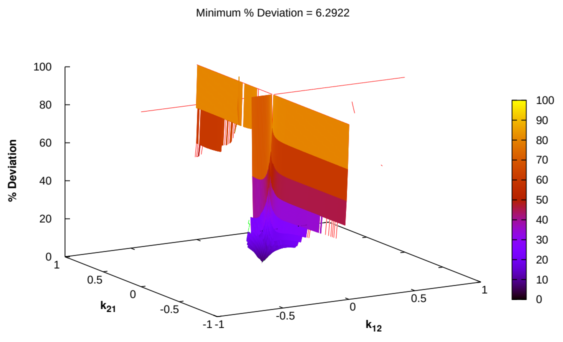


(d) Partial Map

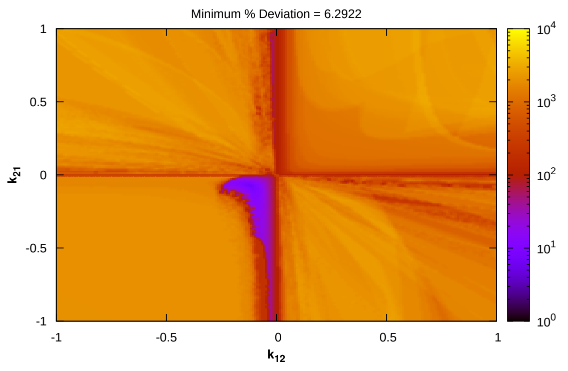
Figure 47: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 02010.



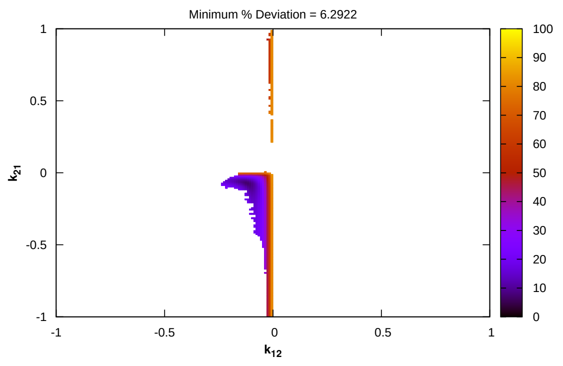
(a) Full Surface



(b) Partial Surface

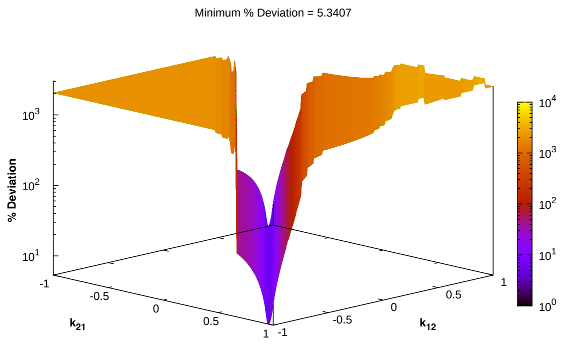


(c) Full Map

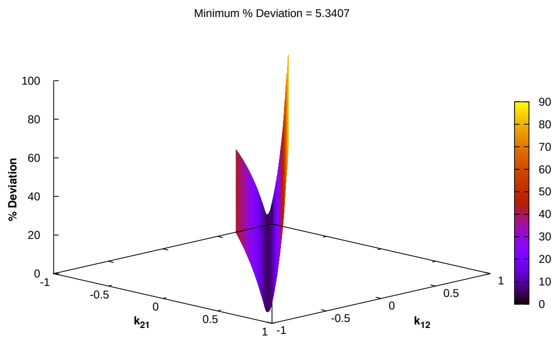


(d) Partial Map

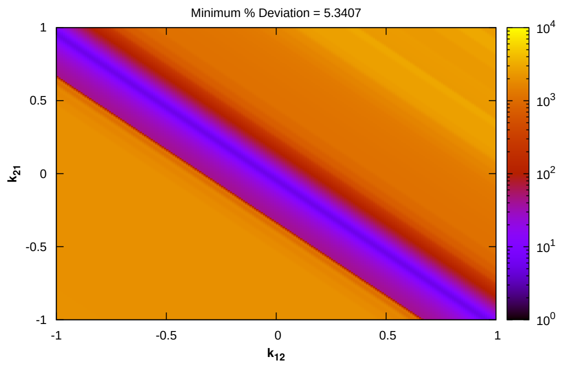
Figure 48: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 03010.



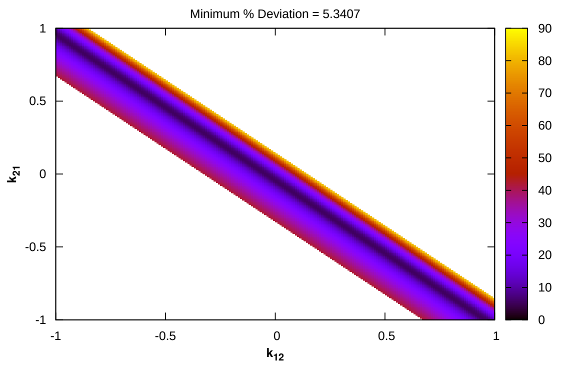
(a) Full Surface



(b) Partial Surface

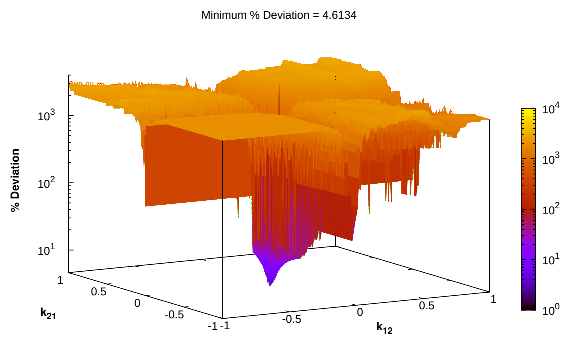


(c) Full Map

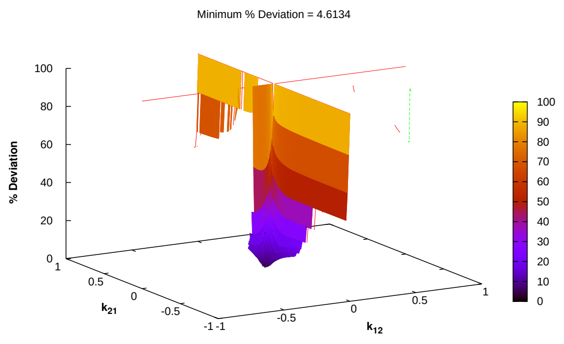


(d) Partial Map

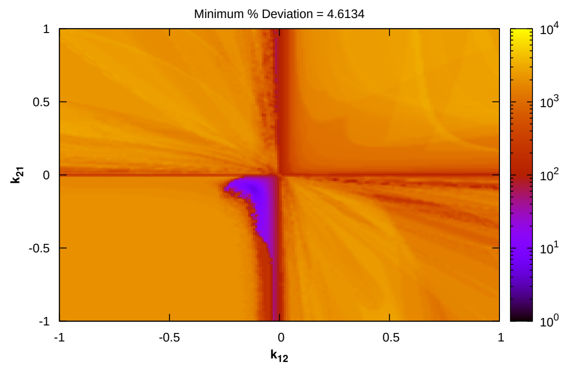
Figure 49: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 10010.



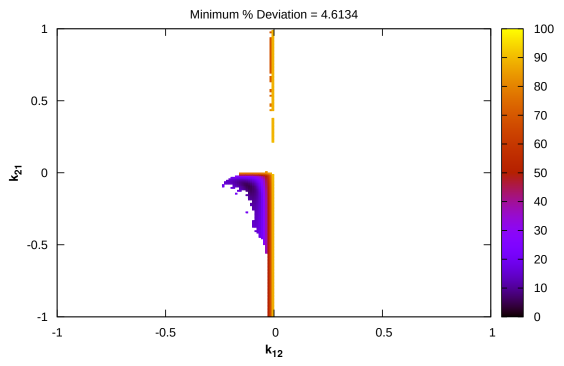
(a) Full Surface



(b) Partial Surface

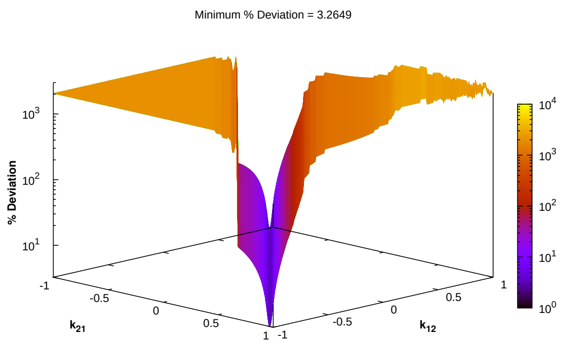


(c) Full Map

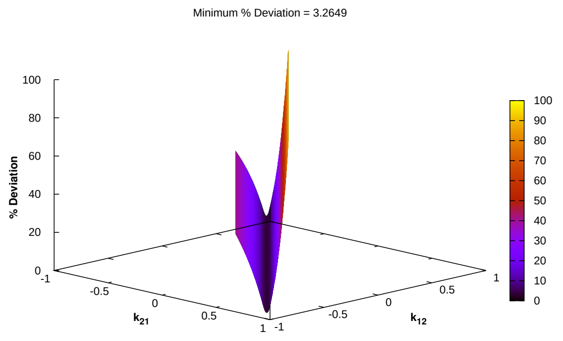


(d) Partial Map

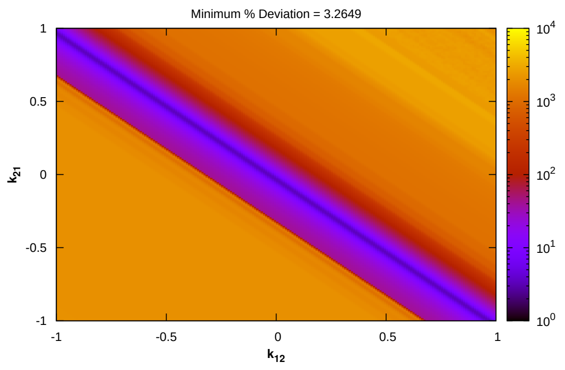
Figure 50: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 13010.



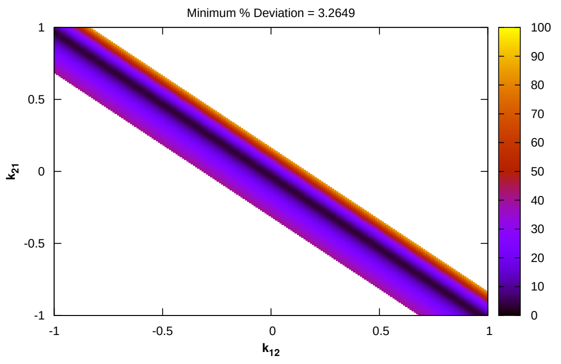
(a) Full Surface



(b) Partial Surface

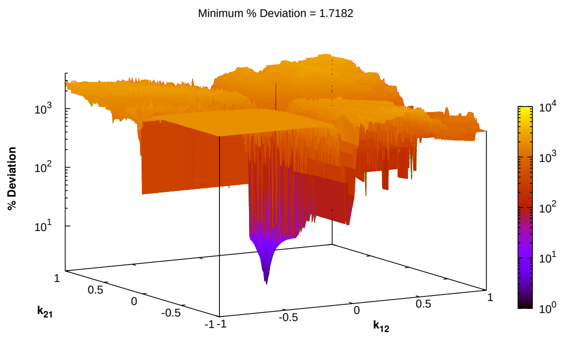


(c) Full Map

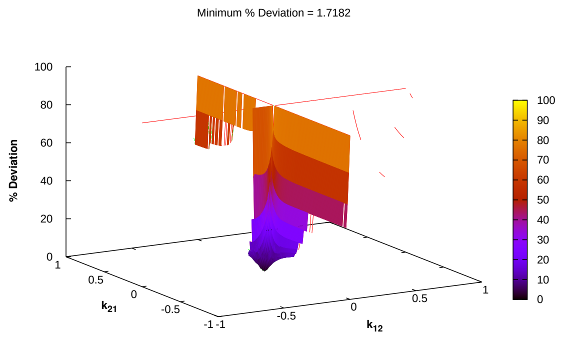


(d) Partial Map

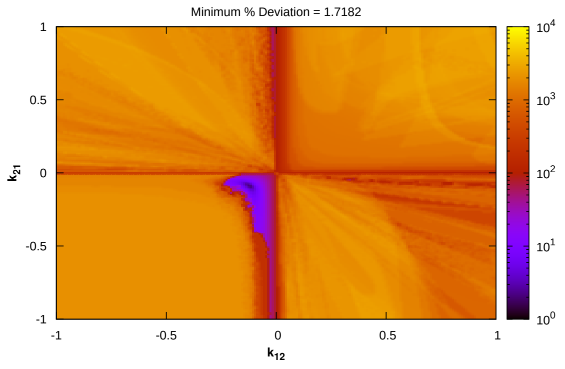
Figure 51: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 20010.



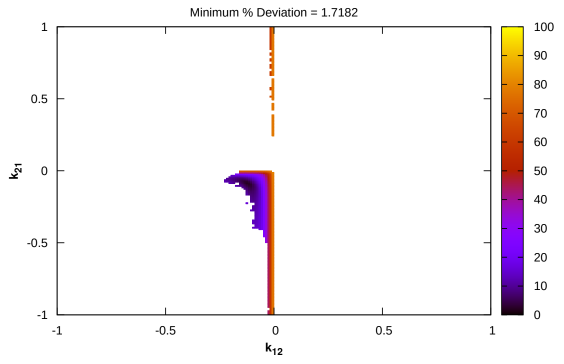
(a) Full Surface



(b) Partial Surface



(c) Full Map



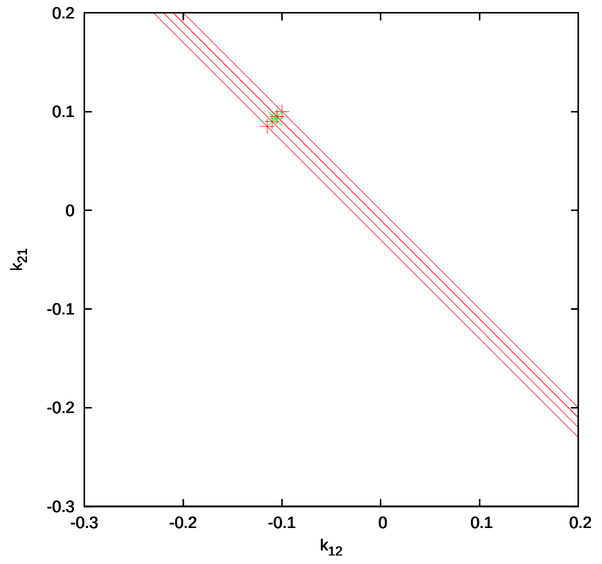
(d) Partial Map

Figure 52: Average percent deviation for k_{ij} pairs at $T = 323.65K$ using equation set 23010.

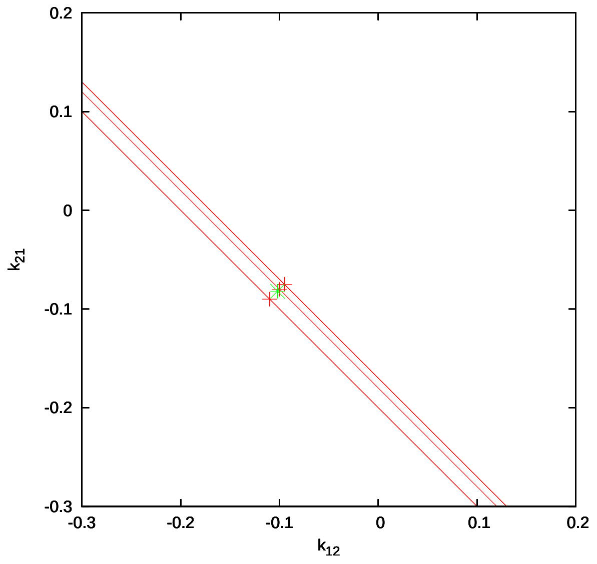
APPENDIX H

EFFECT OF TEMPERATURE ON THE OPTIMAL BINARY INTERACTION PARAMETERS

See Section 5.6 for an explanation of these figures.

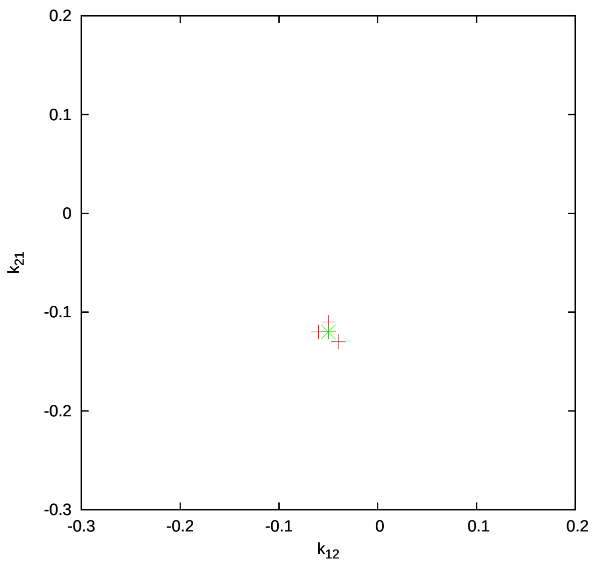


(a) Equation Set: 00010

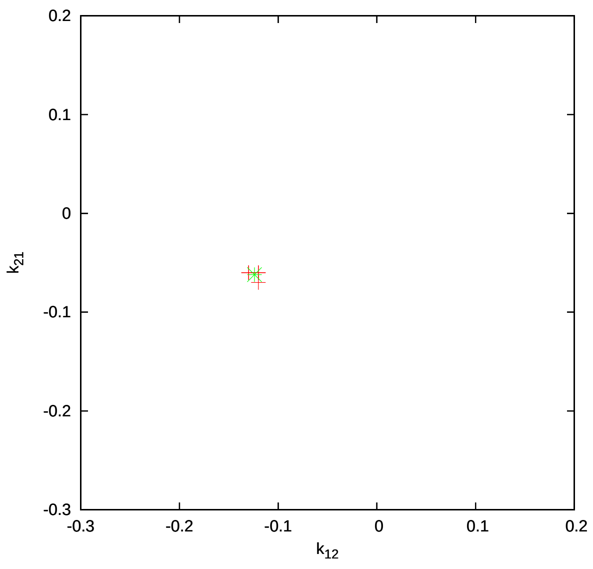


(b) Equation Set: 01010

Figure 53: Effect of temperature on the optimal k_{ij} pairs using EOS = 0.

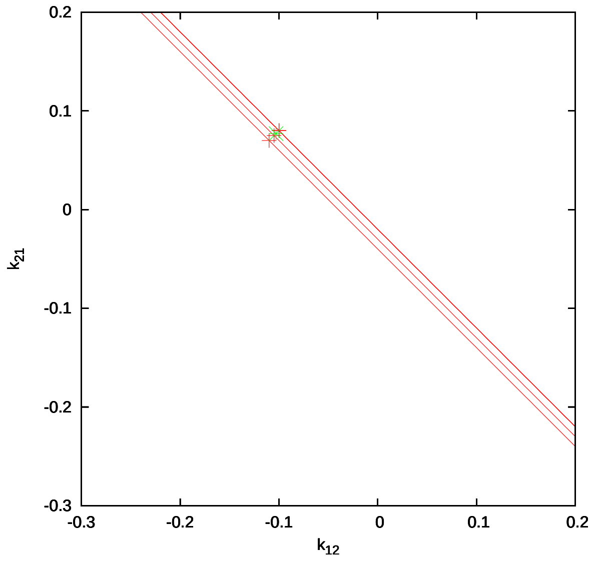


(c) Equation Set: 02010

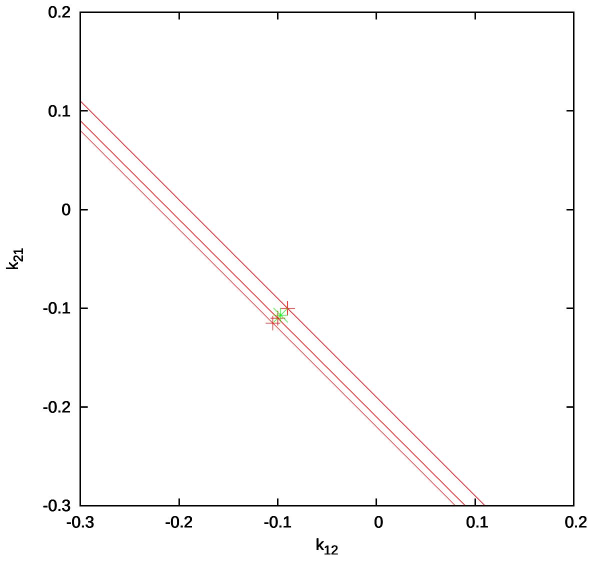


(d) Equation Set: 03010

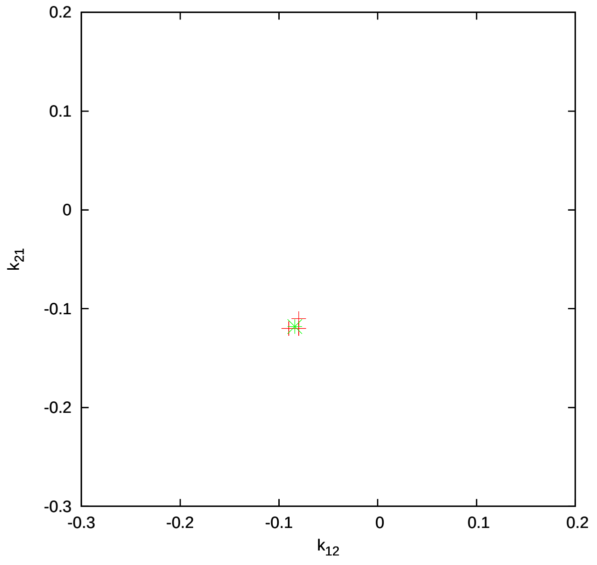
Figure 53: Effect of temperature on the optimal k_{ij} pairs using EOS = 0.



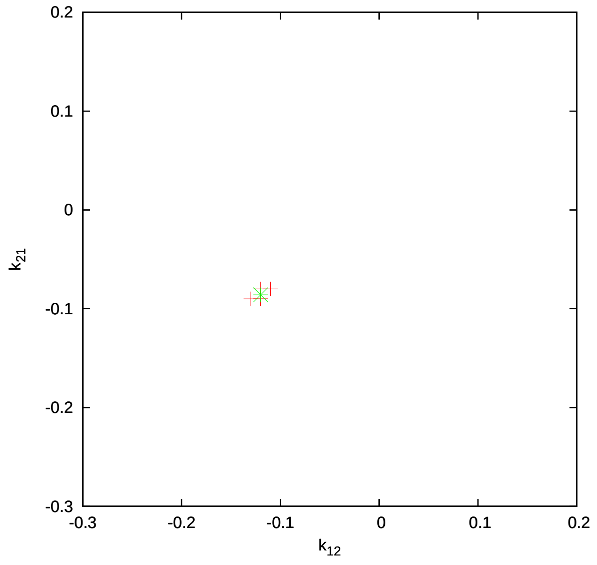
(a) Equation Set: 10010



(b) Equation Set: 11010

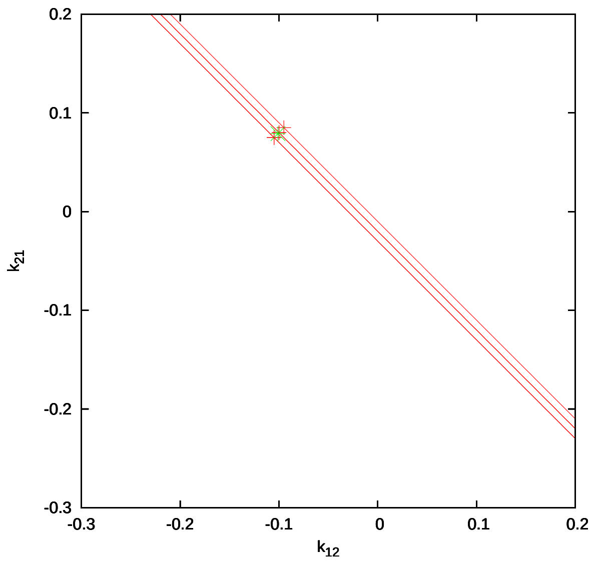


(c) Equation Set: 12010

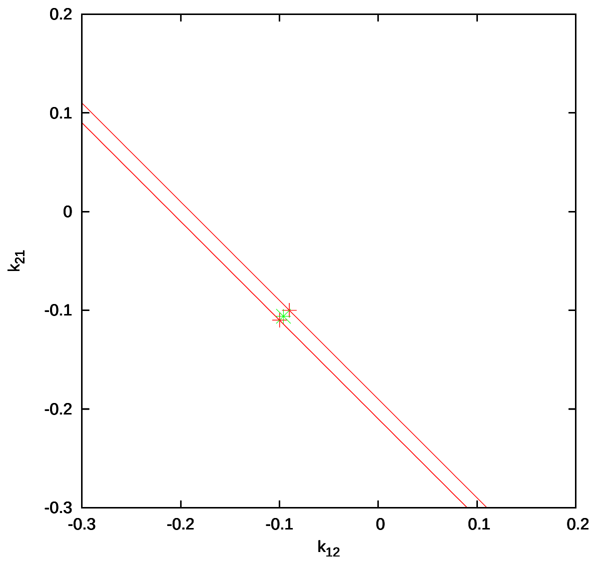


(d) Equation Set: 13010

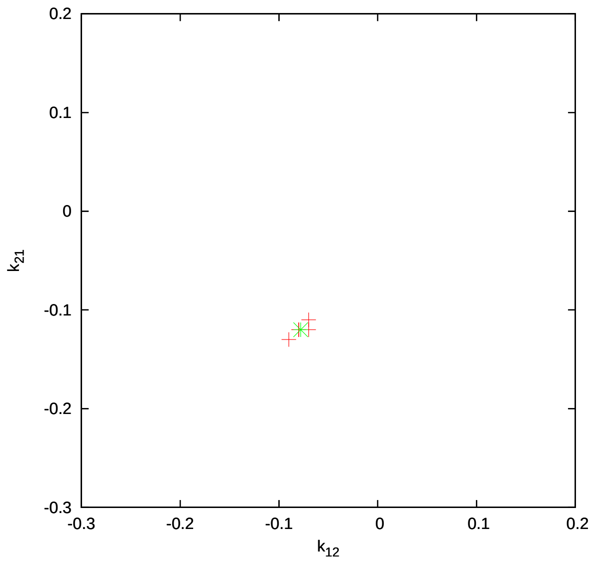
Figure 54: Effect of temperature on the optimal k_{ij} pairs using EOS = 1.



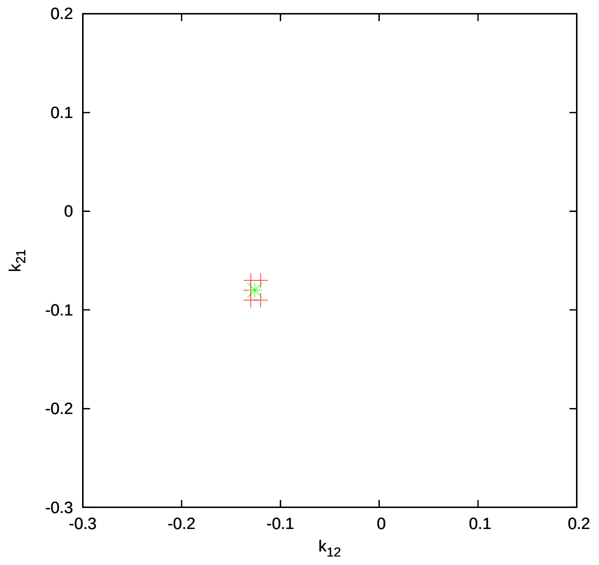
(a) Equation Set: 20010



(b) Equation Set: 21010



(c) Equation Set: 22010



(d) Equation Set: 23010

Figure 55: Effect of temperature on the optimal k_{ij} pairs using EOS = 2.

APPENDIX I

CODE FOR MAIN VLE CALCULATIONS

I.1 *VLEMAIN.F90*

The vlemain.f90 code is the frontend for the standard VLE calculations. It calls other subroutines that are included in Appendix L.

```
PROGRAM vlemain

   ! This program is used to call the general VLE calculations.
   ! Provided the files 'eth.in', 'kijpairs', and 'fits', it
   ! will calculate bubble and dew point curves at various
   ! temperatures or pressures over full ranges of compositions using
   ! optimal binary interaction parameter pairs.  Various
   ! equations of state, combining rules and mixing rules can be
   ! implemented for these calculations.

   ! This program file consists primarily of methods to read data
   ! from input files as well as experimental data files which
   ! are used as part of the simulation.

   use nrtype; use global; use vlesolve; use devcalc;
   use kijmod; use vlecalcs; use rules;
   implicit none

   character(len=20) ::infile,inputstr,rulestr
   character(len=6) ::pstr,valstr
   character(len=3) :: tstr
   logical(4) :: d1,d2,d3,incheck,datcheck
   integer(I4B) :: i,n,bvalidpt,dvalidpt,btotpts,dtotpts,method
   integer(I4B) :: numeos,nummix,numcomb,numvars,BN
   integer(I4B), dimension(:,:), allocatable :: rulesarr
   real(DP) :: T,P,Tadj,Padj
real(DP) :: bubavedev,bubaveperdev,dewavedev,dewaveperdev
real(DP) :: btotad,btotapd,dtotad,dtotapd
real(DP) :: type_d,variations_d,eq_d
real(DP), dimension(2) :: Tc_2,Pc_2,w_2,kap1_2,kap2_2,kap3_2
real(DP), dimension(2) :: mixrules_d,comrules_d
real(DP), dimension(:), allocatable :: bubarr,dewarr,zarr,xarr,yarr
real(DP), dimension(:), allocatable :: texp_temp,pexp_temp, &
     xexp_temp,yexp_temp,npexp_temp

delta = 10000     ! Discretization of the composition array
conv = 0.00001d0 ! A convergence indicator for equivalence checking
Tadj = -1.0d0     ! Initial adjustment for the temperature [K]
Padj = 3844.0d0  ! Initial adjustment for the pressure [Pa]

inputstr='ethanol'
infile = 'eth.in'
datafile = 'eth.dat'

! Allocate arrays to store the initial composition and the results.
allocate(zarr(delta+1),xarr(delta+1),yarr(delta+1), &
     bubarr(delta+1),dewarr(delta+1))

! Initialize the main composition array.
do i=1,delta+1
   zarr(i) = i-1
end do
zarr = zarr/delta


! Read the input file into the correct variables
inquire(file=infile,exist=incheck)
if (incheck) then
   open(10,file=infile,access='sequential', &
        form='formatted',status='old')
   read(10,fmt="(/A1)")d1,d2,d3
   read(10,fmt="(/F18.8)")T,P,type_d,variations_d,eq_d, &
        comrules_d(1),comrules_d(2),mixrules_d(1),mixrules_d(2), &
        Tc(1),Tc(2),Pc(1),Pc(2),w(1),w(2),rho(1),rho(2), &
        kap1(1),kap1(2),kap2(1),kap2(2),kap3(1),kap3(2), &
        Tc_2(1),Tc_2(2),Pc_2(1),Pc_2(2),w_2(1),w_2(2), &
        kap1_2(1),kap1_2(2),kap2_2(1),kap2_2(2),kap3_2(1),kap3_2(2)
   close(10)
   ! Convert data into the correct numeric types
   type = int(type_d); variations = int(variations_d); eq = int(eq_d)
   mixrules(1) = int(mixrules_d(1)); mixrules(2) = int(mixrules_d(2))
   comrules(1) = int(comrules_d(1)); comrules(2) = int(comrules_d(2))
   write(tstr,fmt="(I3)")floor(T)
   write(pstr,fmt="(I6)") floor(P)
else
   print *, "ERROR: No input file available.  Terminating"
   stop
endif

! Allocate variables and store experimental data if present.
inquire(file=datafile,exist=datcheck)
if (datcheck) then
   BN = 1000 ! A big number
   allocate(texp_temp(BN),xexp_temp(BN),yexp_temp(BN), &
        pexp_temp(BN),npexp_temp(BN))
   open(20,file=datafile,status='old')
   lines = 0
   do i=1,BN
      ! Read each line of the exp data file into local variables
      read(20,fmt=*,IOSTAT=EOF)texp_temp(i),xexp_temp(i), &
           yexp_temp(i),pexp_temp(i),npexp_temp(i)
      if(EOF==-1)exit
      lines = lines+1 ! Keep track of the number of lines.
      pexp(i) = pexp(i)/1000.0d0 ! Convert pressures to kPa.
   end do
   close(20)
   ! Reset to permanent arrays of the correct size.
   allocate(texp(lines),pexp(lines),npexp(lines), &
        xexp(lines),yexp(lines))
   do i=1,lines
      texp(i) = texp_temp(i); pexp(i) = pexp_temp(i)
      xexp(i) = xexp_temp(i); yexp(i) = yexp_temp(i)
      npexp(i) = npexp_temp(i)
   enddo
   deallocate(texp_temp,pexp_temp,xexp_temp,yexp_temp,npexp_temp)
else
   print *, "ERROR: No experimental datafile included."
endif

! Set the amount to adjust the guessed pressure or temperature
! during the VLE calculation based on the type of calculation
! that is being performed.  Note: the sign is important here.
if (type==0) then
   adjust = Padj  ! Amount to change the pressure each iteration [Pa]
else
   adjust = Tadj  ! Amount to change the temp each iteration [K]
endif

! If the equation of state, combining rules, and mixing rules are
! supposed to vary, setup an array to hold all of the combinations.
! There are 3 EOS (RKS, PR, PRSV), 2 mixing (linear, quadratic),
! and 4 combining rules (arithmetic, conventional ,margules ,
! van Laar) currently implemented.  They will be stored in rulesarr
! as (eq,comb(1),comb(2),mix(1),mix(2))
if (variations == 1) then
   numeos=3; nummix=2; numcomb=4; numvars = numeos*nummix*numcomb
   allocate(rulesarr(numvars, 5))
   call varyrules(numeos,nummix,numcomb,numvars,rulesarr)
else
   numvars=1
endif

! Loop over the possible eos/cr/mr combinations.
n=1
ruleloop: do while(n<=numvars)
   ! Set the current rules from rulesarr if necessary.
   if (variations==1) then
      eq = rulesarr(n,1)
      comrules(1) = rulesarr(n,2); comrules(2) = rulesarr(n,3)
      mixrules(1) = rulesarr(n,4); mixrules(2) = rulesarr(n,5)
      comrules_fix(:)=comrules(:)
   endif

   ! Set the kij values based on the current rule combination.
   write(rulestr,fmt="(I1I1I1I1I1)")eq,comrules,mixrules
   if(mixrules(1)==0) then ! kij values should be 0 (no effect)
      method=0
   elseif(comrules(1)==0.or.comrules(1)==1 )then ! Use the fits file
      method=1
   else ! Use the kijpairs files
      method=2
   endif
   call setkij(method,rulestr,tstr,pstr)

   ! If the PRSV-EOS is specified, reset to a different set of
   ! variables to keep internal consistency with the PRSV parameters.
   if (eq==2) then
      Tc = Tc_2; Pc = Pc_2; w = w_2;
      kap1 = kap1_2; kap2 = kap2_2; kap3 = kap3_2;
   endif

   ! Initialize the results arrays and call the VLE calculations.
   bubarr(:)=0.0d0; dewarr(:)=0.0d0; xarr(:)=0.0d0; yarr(:)=0.0d0
   call vle(T,P,zarr,bubarr,dewarr,xarr,yarr)

   ! Check the Bubble point array against some experimental data and
   ! calculate both the percent deviations and total deviations.
! Store this in a file so it can be plotted.
! The .dat experimental file must exist in order to do this.
if (datcheck) then

   ! Initialize the errors and point counts.
   btotpts = 0; btotapd = 0.0d0; btotad = 0.0d0
   dtotpts = 0; dtotapd = 0.0d0; dtotad = 0.0d0
   open(30,file=rulestr,status='unknown',position='append')

   ! Calculate the deviations by calling the devcalc module.
   do i = 1,delta+1
      call bubdev(T,P,zarr(i),bubarr(i),bvalidpt, &
           bubavedev,bubaveperdev)
      call dewdev(T,P,zarr(i),dewarr(i),dvalidpt, &
           dewavedev,dewaveperdev)
      if(type==0)then
         bubarr(i) = bubarr(i)/1000.0d0
         dewarr(i) = dewarr(i)/1000.0d0
      endif

      ! Write the calculated errors to a file when they are found.
      ! If no experimental data exists at this point, just write
      ! a 'NA' to the file.
      if(bvalidpt==1.and.dvalidpt==1) then
         write(30,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              bubavedev,bubaveperdev,dewavedev,dewaveperdev
         btotpts = btotpts+1; dtotpts = dtotpts+1
         btotapd = btotapd + bubaveperdev
         btotad = btotad + bubavedev
         dtotapd = dtotapd + dewaveperdev
         dtotad = dtotad + dewavedev
      elseif(bvalidpt==1.and.dvalidpt==0) then
         write(30,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              bubavedev,bubaveperdev,'NA ','NA'
         btotpts = btotpts+1
         btotapd = btotapd + bubaveperdev
         btotad = btotad + bubavedev
      elseif(bvalidpt==0.and.dvalidpt==1) then
         write(30,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              'NA ','NA ',dewavedev,dewaveperdev
         dtotpts = dtotpts+1
         dtotapd = dtotapd + dewaveperdev
         dtotad = dtotad + dewavedev
      else
         write(30,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              'NA ','NA ','NA ','NA'
      endif
```

```
        enddo
        close(30)

        ! Calculate the average errors if possible.
        if(btotpts>0)then
           bubaveperdev=btotapd/btotpts
           bubavedev=btotad/btotpts
        endif
        if(dtotpts>0)then
           dewaveperdev=dtotapd/dtotpts
           dewavedev=dtotad/dtotpts
        endif
     endif

     ! Write the kij and aveperdev values to a file so they
     ! can be analyzed later.
     open(40,file='usedkijs',status='unknown',position='append')
     if (type==0)then
        valstr = tstr
     else
        valstr = pstr
     endif
     if(btotpts>0.and.dtotpts>0)then
        write(40,fmt=*)trim(rulestr),'    ',trim(valstr), &
             k12,k21,bubaveperdev,dewaveperdev
     elseif (btotpts>0.and.dtotpts==0)then
        write(40,fmt=*)trim(rulestr),'    ',trim(valstr), &
             k12,k21,bubaveperdev,'NA'
     elseif(btotpts==0.and.dtotpts>0)then
        write(40,fmt=*)trim(rulestr),'    ',trim(valstr), &
             k12,k21,'NA  ',dewaveperdev
     else
        write(40,fmt=*)trim(rulestr),'    ',trim(valstr), &
             k12,k21,'NA  ','NA'
     endif
     close(40)

     n=n+1
  end do ruleloop

  ! Deallocate memory as needed.
  deallocate(bubarr,dewarr,zarr,xarr,yarr)
  if(datcheck) deallocate(texp,xexp,yexp,pexp,npexp)
  if(variations==1) deallocate(rulesarr)

end PROGRAM vlemain
```

I.2 *VLESOLVE.F90*

The vlesolve.f90 code contains the outer loop of the VLE calculation.

```
MODULE vlesolve

CONTAINS

  subroutine vle(T,P,zarr,bubarr,dewarr,xarr,yarr)
    use nrtype; use global; use devcalc; use vlecalcs;
    implicit none

    ! This subroutine contains the outer loop of the vle process,
    ! which corresponds to taking steps through the composition array.

    real(DP), intent(IN) :: T,P
    real(DP), dimension(:), intent(IN) :: zarr
    real(DP), dimension(:), intent(OUT) :: bubarr,dewarr,xarr,yarr
    real(DP) :: seed,xseed
    integer(I4B) :: i

    ! The zloops run the vle calculation for each step of
    ! the composition, z, from 0 to 1.  First, calculate the
    ! bubble points by calling vlebub from the vlecalcs module.
    i=1
    zloopbub: do while (i<=delta+1)
       ! Seed the VLE calculation with data from the input file
       ! for the first point.  For the following compuations,
       ! use the previous data to provide a better seed value.
       if (i==1) then
          if (type==0) then
             seed = P
          else
             seed = T
          endif
       elseif (i==2) then
          seed = bubarr(1)
       else
          seed = bubarr(i-1) + (bubarr(i-1) - bubarr(i-2))
       endif

       call vlebub(T,P,seed,zarr(i),adjust,bubarr(i),yarr(i))
       failed=0
       i=i+1
    end do zloopbub

    ! The dewcalcs require a finer adjust parameter because phiL is
  ! the main factor used in adjusting the compositions
  ! and phiL << phiV.
  adjust=adjust/100

  ! Repeat this process for the dewpoint calculations, calling
  ! vledew from the vlecalcs module.
  i=1
  zloopdew: do while(i<=delta+1)
     if (i==1) then
        seed = bubarr(1)
     elseif (i==2) then
        seed = dewarr(1)
     else
        seed = dewarr(i-1) + (dewarr(i-1)-dewarr(i-2))
     endif

     ! Seed the value of x, which is guessed in vledew using
     ! data calculated by vlebub.
     call dewseedx(i,zarr,yarr,xseed)
     call vledew(T,P,seed,zarr(i),xseed,adjust,dewarr(i),xarr(i))
     failed=0
     i=i+1
  end do zloopdew
  adjust=adjust*100
end subroutine vle



subroutine dewseedx(i,zarr,yarr,xseed)
  use nrtype; use global
  implicit none

  ! This subroutine should search the yarr created during the
  ! vlebub subroutine for a value that is close to the current
  ! yval.  This index can then be used to find a value within
  ! the previous xarr to determine a x seed value.

  integer(I4B), intent(IN) :: i
  real(DP), dimension(:), intent(IN) :: zarr,yarr
  real(DP), intent(OUT) :: xseed

  integer(I4B) :: j, minindx=1
  real(DP) :: diff, min=10000

  do j=1,size(yarr)
     diff = abs(zarr(i)-yarr(j))
```

```
         if(j==1)min=diff
         if(diff<=min) then
            min=diff
            minindx=j
         endif
      enddo
      xseed = zarr(minindx)

   end subroutine dewseedx
end MODULE vlesolve
```

APPENDIX J

CODE FOR BINARY INTERACTION PARAMETER PERTURBATIONS

The kijperturb.f90 code is the frontend for the calculations involving a perturbation of the optimal binary interaction parameters. This program also calls the vlesolve module that was shown in Appendix I.

J.1 *KIJPERTURB.F90*

```
PROGRAM kijperturb

  ! This program should be used to run a VLE calculation
  ! for the optimal kij pairs in addition to 2 other pairs
  ! that are +-10% deviation from optimal.  The optimal pairs
  ! are read in from a file called 'usedkijs' that is produced
  ! by the vlemain.f90 program.

  use nrtype; use global; use vlesolve; use devcalc;
  use kijmod; use vlecalcs; use rules;
  implicit none

  character(len=20) :: infile,inputstr,rulestr,resultsfile
  character(len=6) :: pstr,valstr
  character(len=3) :: tstr
  logical(4) :: d1,d2,d3,incheck,datcheck
  integer(I4B) :: i,m,n,bvalidpt,dvalidpt,btotpts,dtotpts,method
  integer(I4B) :: numeos,nummix,numcomb,numvars,BN,numpairs
  integer(I4B), dimension(:,:), allocatable :: rulesarr
  real(DP) :: T,P,Tadj,Padj
real(DP) :: dewavedev,dewaveperdev,dtotad,dtotapd
real(DP) :: bubavedev,bubaveperdev,btotad,btotapd
real(DP) :: maxkijpert,kijpert,perkijpert,k12center,k21center
real(DP) :: type_d, variations_d, eq_d, kijcheck_d
real(DP), dimension(2) :: Tc_2,Pc_2,w_2,kap1_2,kap2_2,kap3_2
real(DP), dimension(2) :: mixrules_d, comrules_d
real(DP), dimension(:), allocatable :: bubarr,dewarr,zarr,xarr,yarr
real(DP), dimension(:), allocatable :: texp_temp,pexp_temp, &
     xexp_temp,yexp_temp,npexp_temp

delta = 10000     ! Discretization of the composition array
conv = 0.00001d0 ! A convergence indicator for equivalence checking
Tadj = -1.0d0     ! Initial adjustment for the temperature [K]
Padj = 3844.0d0  ! Initial adjustment for the pressure [Pa]
numpairs = 3     ! Number of kijpairs to run.  This should be odd.
perkijpert = 0.1d0 ! Amount to perturb for each pair
maxkijpert = perkijpert * (numpairs - 1)/2 ! Maximum perturbation


inputstr='ethanol'
infile = 'eth.in'; datafile = 'eth.dat'; lines = 69!286


! Allocate arrays to store the initial composition and the results.
allocate(zarr(delta+1),xarr(delta+1),yarr(delta+1), &
     bubarr(delta+1),dewarr(delta+1))

! Initialize the main composition array.
do i=1,delta+1
   zarr(i) = i-1
end do
zarr = zarr/delta


! Read the input file into the correct variables
inquire(file=infile,exist=incheck)
if (incheck) then
   open(10,file=infile,access='sequential', &
        form='formatted',status='old')
   read(10,fmt="(/A1)")d1,d2,d3
   read(10,fmt="(/F18.8)")T,P,type_d,variations_d,eq_d, &
        comrules_d(1),comrules_d(2),mixrules_d(1),mixrules_d(2), &
        Tc(1),Tc(2),Pc(1),Pc(2),w(1),w(2),rho(1),rho(2), &
        kap1(1),kap1(2),kap2(1),kap2(2),kap3(1),kap3(2), &
        Tc_2(1),Tc_2(2),Pc_2(1),Pc_2(2),w_2(1),w_2(2), &
        kap1_2(1),kap1_2(2),kap2_2(1),kap2_2(2),kap3_2(1),kap3_2(2)
   close(10)
   ! Convert data into the correct numeric types
   type = int(type_d); variations = int(variations_d); eq = int(eq_d)
   mixrules(1) = int(mixrules_d(1)); mixrules(2) = int(mixrules_d(2))
   comrules(1) = int(comrules_d(1)); comrules(2) = int(comrules_d(2))
   write(tstr,fmt="(I3)")floor(T)
   write(pstr,fmt="(I6)")floor(P)
else
   print *, "ERROR: No input file available.  Terminating"
   stop
endif


! Allocate variables and store experimental data if present.
inquire(file=datafile,exist=datcheck)
if (datcheck) then
   BN = 1000 ! A big number
   allocate(texp_temp(BN),xexp_temp(BN),yexp_temp(BN), &
         pexp_temp(BN),npexp_temp(BN))
   open(20,file=datafile,status='old')
   lines = 0
   do i=1,BN
      ! Read each line of the exp data file into local variables
      read(20,fmt=*,IOSTAT=EOF)texp_temp(i),xexp_temp(i), &
            yexp_temp(i),pexp_temp(i),npexp_temp(i)
      if(EOF==-1)exit
      lines = lines+1 ! Keep track of the number of lines.
      pexp(i) = pexp(i)/1000.0d0 ! Convert pressures to kPa.
   end do
   close(20)
   ! Reset to permanent arrays of the correct size.
   allocate(texp(lines),pexp(lines),npexp(lines), &
         xexp(lines),yexp(lines))
   do i=1,lines
      texp(i) = texp_temp(i); pexp(i) = pexp_temp(i)
      xexp(i) = xexp_temp(i); yexp(i) = yexp_temp(i)
      npexp(i) = npexp_temp(i)
   enddo
   deallocate(texp_temp,pexp_temp,xexp_temp,yexp_temp,npexp_temp)
else
   print *, "ERROR: No experimental datafile included."
endif

! Set the amount to adjust the guessed pressure or temperature
! during the VLE calculation based on the type of calculation
! that is being performed.  Note: the sign is important here.
if (type==0) then
   adjust = Padj  ! Amount to change the pressure each iteration [Pa]
```

```
else
   adjust = Tadj  ! Amount to change the temp each iteration [K]
endif


! If the equation of state, combining rules, and mixing rules are
! supposed to vary, setup an array to hold all of the combinations.
! There are 3 EOS (RKS, PR, PRSV), 2 mixing (linear, quadratic),
! and 4 combining rules (arithmetic, conventional ,margules ,
! van Laar) currently implemented.  They will be stored in rulesarr
! as (eq,comb(1),comb(2),mix(1),mix(2))
if (variations == 1) then
   numeos=3; nummix=2; numcomb=4; numvars = numeos*nummix*numcomb
   allocate(rulesarr(numvars, 5))
   call varyrules(numeos,nummix,numcomb,numvars,rulesarr)
else
   numvars=1
endif

! Loop over the possible eos/cr/mr combinations.
n=1
ruleloop: do while(n<=numvars)
   ! Set the current rules from rulesarr if necessary.
   if (variations==1) then
      eq = rulesarr(n,1)
      comrules(1) = rulesarr(n,2); comrules(2) = rulesarr(n,3)
      mixrules(1) = rulesarr(n,4); mixrules(2) = rulesarr(n,5)
      comrules_fix(:)=comrules(:)
   endif
   ! If the PRSV-EOS is specified, reset to a different set of
   ! variables to keep internal consistency with the PRSV parameters.
   if (eq==2) then
      Tc = Tc_2; Pc = Pc_2; w = w_2;
      kap1 = kap1_2; kap2 = kap2_2; kap3 = kap3_2;
   endif

   ! Set the kij values based on the current rule combination.
   write(rulestr,fmt="(I1I1I1I1I1)")eq,comrules,mixrules
   if(mixrules(1)==0) then ! kij values should be 0 (no effect)
      method=0
   elseif(comrules(1)==0.or.comrules(1)==1 )then ! Use the fits file
      method=1
   else ! Use the kijpairs files
      method=2
   endif
   call setkij(method,rulestr,tstr,pstr)
   k12center = k12
```

```
k21center = k21

! Reset to the maxkijpert for each new kijloop
kijpert = maxkijpert

! Only run a perturbation if there is a kij dependence.
if(mixrules(1)==0)then
   m=numpairs+1
else
   m=1
endif

! Loop over the number of kijpairs specified.
kijloop: do while(m<=numpairs)

   ! Modify the kij values
   k12 = k12center*(1.0d0+kijpert)
   k21 = k21center*(1.0d0+kijpert)

   ! If the center values were 0, modify the perturbation method
   if(k12center==0)then
      k12 = kijpert*0.01d0
   endif
   if(k21center==0)then
      k21 = kijpert*0.01d0
   endif

   ! Update the value of kijpert
   kijpert = (kijpert-perkijpert)

   ! Initialize the results arrays and call the VLE calculations.
   bubarr(:)=0.0d0; dewarr(:)=0.0d0; xarr(:)=0.0d0; yarr(:)=0.0d0
   call vle(T,P,zarr,bubarr,dewarr,xarr,yarr)

   if(m<10)then
      write(resultsfile,fmt="(A5A1I1)")rulestr,'_',m
   else
      write(resultsfile,fmt="(A5A1I2)")rulestr,'_',m
   endif
   ! Check the Bubble point array against some experimental data.
   ! This data is stored in valid.dat so it can be plotted.
   ! Note: The .dat experimental file must exist and the length
   ! of the file must be specified above.
   !if(mixrules(1)==0)then
   !   write(resultsfile,fmt="(A5A2)")rulestr,'_1'
   if(m<10)then
      write(resultsfile,fmt="(A5A1I1)")rulestr,'_',m
```

```
else
   write(resultsfile,fmt="(A5A1I2)")rulestr,'_',m
endif

! Check the Bubble point array against some experimental
! data and calculate both the percent deviations and total
! deviations.  Store this in a file so it can be plotted.
! The .dat experimental file must exist in order to do this.
if (datcheck) then
   btotpts = 0; btotapd = 0.0d0; btotad = 0.0d0
   dtotpts = 0; dtotapd = 0.0d0; dtotad = 0.0d0
   open(30,file=resultsfile,status='unknown',position='append')

   ! Calculate the deviations by calling the devcalc module.
   do i = 1,delta+1
      call bubdev(m,T,P,zarr(i),bubarr(i),bvalidpt, &
           bubavedev,bubaveperdev)
      call dewdev(m,T,P,zarr(i),dewarr(i),dvalidpt, &
           dewavedev,dewaveperdev)
      if(type==0)then
         bubarr(i) = bubarr(i)/1000.0d0
         dewarr(i) = dewarr(i)/1000.0d0
      endif

      ! Write the calculated errors to a file when they are
      ! found.  If no experimental data exists at this point,
      ! just write a 'NA' to the file.
      if(bvalidpt==1.and.dvalidpt==1) then
         write(320,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              bubavedev,bubaveperdev,dewavedev,dewaveperdev
         btotpts = btotpts+1; dtotpts = dtotpts+1
         btotapd = btotapd + bubaveperdev
         btotad = btotad + bubavedev
         dtotapd = dtotapd + dewaveperdev
         dtotad = dtotad + dewavedev
      elseif(bvalidpt==1.and.dvalidpt/=1) then
         write(320,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              bubavedev,bubaveperdev,'NA  ','NA  '
         btotpts = btotpts+1
         btotapd = btotapd + bubaveperdev
         btotad = btotad + bubavedev
      elseif(bvalidpt/=1.and.dvalidpt==1) then
         write(320,fmt=*)zarr(i),bubarr(i),dewarr(i), &
              'NA  ','NA  ',dewavedev,dewaveperdev
         dtotpts = dtotpts+1
         dtotapd = dtotapd + dewaveperdev
         dtotad = dtotad + dewavedev
```

```
            else
               write(320,fmt=*)zarr(i),bubarr(i),dewarr(i), &
                     'NA  ','NA  ','NA  ','NA  '
            endif
         enddo
         close(30)

         ! Calculate the average errors if possible.
         if(btotpts>0)then
            bubaveperdev=btotapd/btotpts
            bubavedev=btotad/btotpts
         endif
         if(dtotpts>0)then
            dewaveperdev=dtotapd/dtotpts
            dewavedev=dtotad/dtotpts
         endif
      endif

      ! Write the kij and aveperdev values to a file so they
      ! can be analyzed later.
      open(40,file='usedkijs',status='unknown',position='append')
      if (type==0)then
         valstr = tstr
      else
         valstr = pstr
      endif

      if(btotpts>0.and.dtotpts>0)then
         write(40,fmt=*)trim(resultsfile),'    ',trim(valstr), &
               k12,k21,bubaveperdev,dewaveperdev
      elseif (btotpts>0.and.dtotpts==0)then
         write(40,fmt=*)trim(resultsfile),'    ',trim(valstr), &
               k12,k21,bubaveperdev,'NA'
      elseif(btotpts==0.and.dtotpts>0)then
         write(40,fmt=*)trim(resultsfile),'    ',trim(valstr), &
               k12,k21,'NA',dewaveperdev
      else
         write(40,fmt=*)trim(resultsfile),'    ',trim(valstr), &
               k12,k21,'NA  ','NA'
      endif
      close(40)

      m=m+1
   end do kijloop

   n=n+1
end do ruleloop
```

```
  ! Deallocate memory as needed.
  deallocate(bubarr,dewarr,zarr,xarr,yarr)
  if(datcheck) deallocate(texp,xexp,yexp,pexp,npexp)
  if(variations==1) deallocate(rulesarr)

end PROGRAM kijperturb
```

APPENDIX K

CODE FOR FULL ANALYSIS OF BINARY INTERACTION PARAMETERS

K.1 *VLEDEV.F90*

The vledev.f90 code is the frontend for the program that calculates the bubble and dew points for a full range of binary interaction parameters at every available experimental data point. It should be noted that this program relies on the MPI module for parallelization.

```
PROGRAM vledev

   ! This program should be used to run a kij deviation calculation.
   ! A mesh of values will be created and kijpairs will be used to
   ! calculate VLE data at available experimental data points.
   ! The deviation between simulated and experimental values
   ! will be determined and store for later analysis.  This program
   ! implements MPI for parallelization across multiple CPUs.

   use nrtype; use global; use kijmod;
   use rules; use pvlesolve; use mpi
   implicit none

   character(len=20) :: infile,inputstr
   logical(4) :: d1,d2,d3,incheck,datcheck
   integer(I4B) :: h,i,j,k,beg,BN,numeos,nummix,numcomb,myid,rc
   integer(I4B), dimension(:), allocatable :: jvalid_long,jvalid
   integer(I4B), dimension(:,:), allocatable :: rulesarr
   real(DP) :: T,P,Tadj,Padj,start_time,end_time
   real(DP) :: type_d, variations_d, eq_d, kijcheck_d
   real(DP), dimension(2) :: mixrules_d, comrules_d
   real(DP), dimension(:), allocatable :: k12arr,k21arr
   real(DP), dimension(:), allocatable :: texp_temp,pexp_temp, &
       xexp_temp,yexp_temp,npexp_temp
   real(DP), dimension(:,:), allocatable :: pairarr, pairarr_temp
```

```
! Setup all the required MPI components
call MPI_INIT(ierr)
if (ierr.ne.MPI_SUCCESS) then
   print *, 'Error starting MPI program. Terminating.'
   call MPI_ABORT(MPI_COMM_WORLD, rc, ierr); stop
endif
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
print *, 'Process ', myid, ' of ', numprocs-1, ' is alive'
if(myid==0)call cpu_time(start_time)


! Run these steps for every process as each one needs this data.

front=1      ! Current position in pairarr
Tadj = -1.0d0 ! Initial adjustment for the temperature [K]
Padj = 3844.0d0 ! Initial adjustment for the pressure [Pa]

! Setup the parameters for the kij mesh.
k12center=-0.0d0; k21center=0.0d0
k12lenparam=200; k21lenparam=k12lenparam
k12chparam=0.01d0; k21chparam=k12chparam

k12 = k12center; k21 = k21center
k12len = k12lenparam; k21len = k21lenparam
k12change = k12chparam; k21change = k21chparam
pairarrlen = (k12len+1)*(k21len+1)

! Set the name of the input file with substance specific data
inputstr='ethanol'
infile = 'eth.in'
datafile = 'eth.dat'

! Read the input file into the correct variables
inquire(file=infile,exist=incheck)

if(incheck) then
   open(10,file=infile,access='sequential', &
         form='formatted',status='old')
   read(10,fmt="(/A1)")d1,d2,d3
   read(10,fmt="(/F18.8)")T,P,type_d,variations_d,eq_d, &
         comrules_d(1),comrules_d(2),mixrules_d(1),mixrules_d(2), &
         Tc(1),Tc(2),Pc(1),Pc(2),w(1),w(2),rho(1),rho(2), &
         kap1(1),kap1(2),kap2(1),kap2(2),kap3(1),kap3(2), &
         Tc_2(1),Tc_2(2),Pc_2(1),Pc_2(2),w_2(1),w_2(2), &
         kap1_2(1),kap1_2(2),kap2_2(1),kap2_2(2),kap3_2(1),kap3_2(2)
   close(10)
   ! Convert data into the correct numeric types
   type = int(type_d); variations = int(variations_d); eq = int(eq_d)
   mixrules(1) = int(mixrules_d(1)); mixrules(2) = int(mixrules_d(2))
   comrules(1) = int(comrules_d(1)); comrules(2) = int(comrules_d(2))
   write(tstr,fmt="(I3)")floor(T)
   write(pstr,fmt="(I6)")floor(P)
else
   print *, "ERROR: No input file available.  Terminating"
   stop
endif

! Allocate variables and store experimental data if present.
inquire(file=datafile,exist=datcheck)
if (datcheck) then
   BN = 1000 ! A big number
   allocate(texp_temp(BN),xexp_temp(BN),yexp_temp(BN), &
         pexp_temp(BN),npexp_temp(BN))
   open(20,file=datafile,status='old')
   lines = 0
   do i=1,BN
      ! Read each line of the exp data file into local variables
      read(20,fmt=*,IOSTAT=EOF)texp_temp(i),xexp_temp(i), &
            yexp_temp(i),pexp_temp(i),npexp_temp(i)
      if(EOF==-1)exit
      lines = lines+1 ! Keep track of the number of lines.
      pexp(i) = pexp(i)/1000.0d0 ! Convert pressures to kPa.
   end do
   close(20)
   ! Reset to permanent arrays of the correct size.
   allocate(texp(lines),pexp(lines),npexp(lines), &
         xexp(lines),yexp(lines))
   do i=1,lines
      texp(i) = texp_temp(i); pexp(i) = pexp_temp(i)
      xexp(i) = xexp_temp(i); yexp(i) = yexp_temp(i)
      npexp(i) = npexp_temp(i)
   enddo
   deallocate(texp_temp,pexp_temp,xexp_temp,yexp_temp,npexp_temp)
else
   print *, "ERROR: No experimental datafile included."
endif

! Setup some arrays to store relevant exp data lines from *.dat
allocate(jvalid_long(lines)); jvalid_long(:) = 0; validcount = 0
exp: do j=1,lines ! Check arrays. For P-x, check T; For T-x, P.
   if (type == 0) then
      if (abs(texp(j) - T) > conv) cycle exp
   elseif (type==1) then
      if (abs(pexp(j)*1000.0d0 - P) > conv) cycle exp
   endif
   validcount=validcount+1
   jvalid_long(validcount) = j
end do exp
! Create jvalid with the line numbers of valid data points.
allocate(jvalid(validcount))
do j=1,validcount
   jvalid(j) = jvalid_long(j)
enddo
deallocate(jvalid_long)

! Set the amount to adjust the guessed pressure or temperature
! during the VLE calculation based on the type of calculation
! that is being performed.  Note: the sign is important here.
if (type==0) then
   adjust = Padj  ! Amount to change the pressure each iteration [Pa]
else
   adjust = Tadj  ! Amount to change the temp each iteration [K]
endif

! If the equation of state, combining rules, and mixing rules are
! supposed to vary, setup an array to hold all of the combinations.
! There are 3 EOS (RKS, PR, PRSV), 2 mixing (linear, quadratic),
! and 4 combining rules (arithmetic, conventional ,margules ,
! van Laar) currently implemented.  They will be stored in rulesarr
! as (eq,comb(1),comb(2),mix(1),mix(2))
if (variations == 1) then
   numeos=3; nummix=1; numcomb=4; numvars = numeos*nummix*numcomb
   allocate(rulesarr(numvars, 5))
   call varyrules(numeos,nummix,numcomb,rulesarr)
else
   numvars=1
   write(errorfile,fmt="(I1I1I1I1I1A6)")eq,comrules,mixrules,'errors'
endif

! Now, setup the kij mesh only on the root process
if(myid==0)then
   ! Create and initialize the kij arrays.
   allocate(k12arr(k12len+1)); call setk12arr(k12arr)
   allocate(k21arr(k21len+1)); call setk21arr(k21arr)

   ! Create an array of duples where each duple is a kij pair
   allocate(pairarr_temp(pairarrlen,2),pairarr(pairarrlen,2))

   ! In order to use optimal chunksize while keeping an even
```

```
        ! distribution of kij pairs on each proc, setup the pairarr
        ! based on the numprocs available.
        ! First, set all the pairs together as duples.
        j=0
        do k=1,k12len+1
           do h=1,k21len+1
              j=j+1
              pairarr_temp(j,1) = k12arr(k)
              pairarr_temp(j,2) = k21arr(h)
           enddo
        enddo

        ! Then, reorganize the pairs based on numprocs.
        beg = 1
        j=0
        do k=1,numprocs
           do h=beg,pairarrlen,numprocs
              j=j+1
              pairarr(j,1) = pairarr_temp(h,1)
              pairarr(j,2) = pairarr_temp(h,2)
           enddo
           beg=beg+1
        enddo
        deallocate(pairarr_temp)
     endif

     ! Call the VLE calculation on every processor.
     call pvle(T,P,myid,pairarr,jvalid,rulesarr)



     ! Finish up a few things in the root process only.
     if(myid==0)then
        call cpu_time(end_time)
        ! Deallocate root process specific memory
        deallocate(k12arr,k21arr,pairarr)
     endif

     ! Deallocate memory on all processes as needed.
     if(datcheck) deallocate(texp,xexp,yexp,pexp,npexp)
     if(variations==1) deallocate(rulesarr)
     deallocate(jvalid)

     ! Finalize the MPI process.
     call MPI_FINALIZE(ierr)
  end PROGRAM vledev
```

K.2 *VLEDEV.F90*

The pvlesolve.f90 code is the parallelized equivalent of the vlesolve.f90 module shown in Appendix I. In includes subroutines to divide the binary interaction parameters arrays among different processors, communicate with those processors to determine the status of the work, and send jobs to those processors.

```
MODULE pvlesolve
  ! This module is carried out for every processor using MPI.
  ! It loops through a chunk of the kij pairs as well as
  ! any matching experimental data and over EOS/CR/MR combos.
  ! Average errors over the exp data sets are calculated for each
  ! kij pair and rule combination.

CONTAINS

  subroutine pvle(T,P,myid,pairarr,jvalid,rulesarr)
    use nrtype; use global; use pkijmod; use devcalc
    use vlesolve; use vlecalcs; use mpi
    implicit none

    real(DP), intent(IN) :: T,P
    integer(I4B), intent(IN) :: myid
    real(DP), dimension(:,:), intent(IN) :: pairarr
    integer(I4B), dimension(:), intent(IN) :: jvalid
    integer(I4B), dimension(:,:), intent(IN) :: rulesarr
    logical(4) :: stillworking=.true.,work_avail,ALL_PROCS_DONE=.false.
    integer(I4B) :: j, h, i,chunk_loc,ktag=0,ctag=1, &
         status(MPI_STATUS_SIZE),dest,req=0,n
    real(DP) :: deviation,perdev,BubTotErr,DewTotErr, &
         BubAveErr,DewAveErr,bubval,yval,seed,dewval,xval,xseed
    real(DP), dimension(:,:), allocatable :: pairarr_loc,pairarr_temp

    ! Assume that there is enough work for all procs initially.
    work_avail=.true.
    ALL_PROCS_DONE=.false.

    ! Each pair could take a very long time, so limit how
    ! large chunksize can be, otherwise basing chunksize on numprocs.
    ! in order to minimize communcation overhead.
    chunksize = pairarrlen/numprocs
    if(chunksize>3)chunksize=3

    ! Store chunksize so it can be used as an upperbound for MPI calls
    chunk_loc = chunksize

    ! Setup the local array for each processor
```

```
allocate(pairarr_loc(chunksize,2))

! Send out initial set of work to procs from root.
if(myid==0)then
   ! Temp array to send to other procs
   allocate(pairarr_temp(chunksize,2))
   front = 1 ! Start at the beginning of pairarr
   do i=0,numprocs-1
      dest = i
      req = i
      if(i==0)then ! Don't self-send, just set the local array
         pairarr_loc(:,1) = pairarr(front:front+chunksize-1,1)
         pairarr_loc(:,2) = pairarr(front:front+chunksize-1,2)
      else
         pairarr_temp(:,1) = pairarr(front:front+chunksize-1,1)
         pairarr_temp(:,2) = pairarr(front:front+chunksize-1,2)
         call MPI_SEND(pairarr_temp,chunksize*2, &
              MPI_DOUBLE_PRECISION,dest,ktag,MPI_COMM_WORLD,ierr)
      endif
      front = front + chunksize
   enddo
else
   call MPI_RECV(pairarr_loc,chunksize*2, &
        MPI_DOUBLE_PRECISION,0,ktag,MPI_COMM_WORLD,status,ierr)
endif


! Start a loop that continues until all work is done.
do while (work_avail)
   ! Cycle through the current chunk
   do h=1,chunk_loc
      k12 = pairarr_loc(h,1)
      k21 = pairarr_loc(h,2)

      ! Loop over the possible eos/cr/mr combinations.
      n=1
      ruleloop: do while(n<=numvars)
         if (variations==1) then
            eq = rulesarr(n,1)
            comrules(1) = rulesarr(n,2)
            comrules(2) = rulesarr(n,3)
            mixrules(1) = 1 ! Force the quadratic MR for a
            mixrules(2) = rulesarr(n,5)
            write(errorfile,fmt="(I1I1I1I1I1A6)")eq,comrules,&
                 mixrules,'errors'
            open(40,file='errorfilelist',status='unknown', &
                 position='append')
```

```
    write(40,fmt=*)errorfile
    close(40)
 endif

 ! If the PRSV-EOS is specified, reset to a
 ! different set of variables
 if (eq==2) then
    Tc = Tc_2; Pc = Pc_2; w = w_2;
    kap1 = kap1_2; kap2 = kap2_2; kap3 = kap3_2;
 endif


BubTotErr=0.0d0; DewTotErr=0.0d0 ! Initialize the Errors
exp2: do i=1,validcount ! Step through the data points
    j = jvalid(i)
    ! Seed the calcuation depending on the type
    if(type==) then
       seed=pexp(j)*1000.0d0
    else
       seed=texp(j)
    endif
    ! Call the bubble point calculation
    call vlebub(T,P,seed,xexp(j),adjsuc,bubval,yval)
    ! Calculate the errors
    call dev(j,bubval,deviation,perdev)
    BubTotErr = BubTotErr + perdev

    ! Set the xseed to improve simulation speed and
    ! modify adjust and maxiters for the dewpoint routine.
    xseed = xexp(j)
    adjust = adjust/100.0d0
    maxiters = maxiters * 10
    call vledew(T,P,seed,yexp(j),xseed,adjust,dewval,xval)
    call dev(j,dewval,deviation,perdev)
    DewTotErr = DewTotErr + perdev
    adjust = adjust*100.0d0
    maxiters = maxiters/10
end do exp2

! Determine the average errors
BubAveErr = BubTotErr/validcount
DewAveErr = DewTotErr/validcount

! Write the results to a file
if(myid<10)then
    write(errorfile,fmt="(A11I1)")errorfile,myid
elseif(myid<100)then
```

```
            write(errorfile,fmt="(A11I2)") errorfile,myid
         endif
         open(30,file=errorfile,status='unknown',position='append')
         write(30,fmt=*)BubAveErr,DewAveErr,k12,k21
         close(30)


         n=n+1
      enddo ruleloop

      ! If this is the root proc, check the status of the work.
      if(myid==0)then
         ! Check if the last pair on the root proc is done.
         if(h==chunk_loc)stillworking=.false.
         ! Check if all the other procs are done
         if(ALL_PROCS_DONE)rootworkcheck(1)=-1
         ! Call queue_work if some procs are waiting.
         if(rootworkcheck(1).ne.-1)call queue_work(pairarr, &
               pairarr_loc,pairarr_temp,stillworking,ALL_PROCS_DONE)
      endif

   enddo

   ! If this is the root proc, check to see if all the work is done
   if(myid==0)then
      if(rootworkcheck(1)==-1)then
         work_avail = .false.; cycle ! End root process
      elseif(rootworkcheck(1)==0)then
         chunk_loc=workcheck(2)
      endif
   else
      ! If this is not root, this work is finished.
      ! Send a signal root
      call MPI_SEND(workcheck,2,MPI_INTEGER,0,ctag, &
            MPI_COMM_WORLD,ierr)
      ! Receive workcheck and the local kij array from root.
      call MPI_RECV(workcheck,2,MPI_INTEGER,0,ctag, &
            MPI_COMM_WORLD,status,ierr)
      call MPI_RECV(pairarr_loc,chunksize*2, &
            MPI_DOUBLE_PRECISION,0,ktag,MPI_COMM_WORLD,status,ierr)

      ! If root says all work is done, end. Otherwise, continue.
      if(workcheck(1)==-1)then
         work_avail=.false.; cycle ! End current proc
      elseif(workcheck(1)==0)then
         chunk_loc = workcheck(2)
      endif
      endif
   end do

   deallocate(pairarr_loc)
   if(myid==0)deallocate(pairarr_temp)

end subroutine pvle


subroutine queue_work(pairarr,pairarr_loc,pairarr_temp, &
      stillworking,ALL_PROCS_DONE)
   use nrtype; use global; use mpi
   real(DP), dimension(:,:), intent(IN) :: pairarr
   real(DP), dimension(:,:), intent(INOUT) :: pairarr_loc,pairarr_temp
   logical(4), intent(INOUT) :: stillworking, ALL_PROCS_DONE
   logical(4) :: flag
   integer(I4B) :: i,src,dest,ktag=0,ctag=1, &
        status(MPI_STATUS_SIZE),newchunk=1, arraycheck=1

   do i=0,numprocs-1
      src=i
      dest=i
      ! Find out where we are in the pairarr
      if(front>pairarrlen)then ! Already done
         arraycheck = -1
      elseif(front+chunksize-1>pairarrlen) then
         ! Going to run off the end of the array at the next chunk
         arraycheck = 0
      else ! Still good
         arraycheck = 1
      endif

      ! If this is the root proc
      if(i==0)then
         if(stillworking)then
            cycle ! We are still in the middle of a chunk
         else ! We need a new chunk on the root proc
            if(arraycheck==-1)then ! Already done
               rootworkcheck(1)=-1
               cycle
            elseif(arraycheck==0) then ! Going to run off end
               newchunk = pairarrlen - front + 1
               rootworkcheck(1) = 0; rootworkcheck(2) = newchunk
            else ! Still good
               newchunk=chunksize
            endif
            pairarr_loc(1:newchunk,1) = &
```

```
              pairarr(front:front+newchunk-1,1)
         pairarr_loc(1:newchunk,2) = &
              pairarr(front:front+newchunk-1,2)
         front=front+chunksize
         stillworking=.true.
      endif



   else
      ! If the array was finished during this loop, wait for
      ! all the prev procs to finish and send workcheck=-1 to
      ! everything.  NOTE: It is possible that root still has
      ! a last chunk to finish here.
      if(ALL_PROCS_DONE)cycle
      if(arraycheck==-1)then
         do j=1,numprocs-1
            src=j
            dest=j
            call MPI_RECV(workcheck,2,MPI_INTEGER,src,ctag, &
                 MPI_COMM_WORLD,status,ierr)
            workcheck(1)=-1
            call MPI_SEND(workcheck,2,MPI_INTEGER,dest,ctag, &
                 MPI_COMM_WORLD,ierr)
            call MPI_SEND(pairarr_temp,chunksize*2, &
                 MPI_DOUBLE_PRECISION,dest,ktag, &
                 MPI_COMM_WORLD,ierr)
            ALL_PROCS_DONE=.true.
         enddo
      else
         call MPI_IPROBE(src,ctag,MPI_COMM_WORLD,flag,status,ierr)
         if(flag)then ! src is done working
            call MPI_RECV(workcheck,2,MPI_INTEGER,src,ctag, &
                 MPI_COMM_WORLD,status,ierr)
            if(arraycheck==0)then ! Going to run off
               newchunk = pairarrlen - front + 1
               workcheck(1) = 0; workcheck(2) = newchunk
            else ! Still good
               newchunk = chunksize
            endif
            pairarr_temp(1:newchunk,1) = &
                 pairarr(front:front+newchunk-1,1)
            pairarr_temp(1:newchunk,2) = &
                 pairarr(front:front+newchunk-1,2)
            ! Use original chunksize here as it needs to match
            ! the other procs and it will always be the
            ! upperbound of newchunk which is allowable.
            call MPI_SEND(workcheck,2,MPI_INTEGER,dest,ctag, &
                    MPI_COMM_WORLD, ierr)
                call MPI_SEND(pairarr_temp,chunksize*2, &
                    MPI_DOUBLE_PRECISION,dest,ktag, &
                    MPI_COMM_WORLD, ierr)
                front=front+chunksize
             endif
          endif
       endif
    enddo
  end subroutine queue_work

end MODULE pvlesolve
```

APPENDIX L

MODULES USED BY MULTIPLE PROGRAMS

The code files shown below are the body of the VLE calculations. All the modules to loop through a numerical step, set the equations of state, mixing rules, and combining rules, determine fugacity coefficients, and calculated deviations are included here. Not included are the modules used for root finding as the code for those can be found in *Numerical Recipes* [45].

L.1 *VLECALCS.F90*

The vlecalcs.f90 code contains the bulk of the VLE routine. It corresponds to one numerical step, as described in Chapter 4 and is called by all of the main programs shown above.

```
MODULE vlecalcs

CONTAINS
  subroutine vlebub(T,P,seed,zval,adjust,bubval,yval)
    use nrtype; use global; use eosmod; use convfail
    implicit none

    ! This module calculates the bubble points of the
    ! vapor-liquid equilibrium of a mixture using the Phi-Phi
    ! method is used with the Equation of State, Mixing Rules
    ! and Combining Rules that are specified.

    ! The stages of the VLE algorithm are labeled throughout as well.
    ! Stage 1 and 2 involve fixing x_i and P (or T) and assuming
    ! values for T (or P).  This is already done before this
! subroutine is called.

real(DP), intent(IN) :: T,P
real(DP), intent(IN) :: seed,zval
real(DP), intent(INOUT) :: adjust
real(DP), intent(OUT) :: bubval,yval
integer(I4B) :: phase,totalitersin=0,totalitersout=0, &
     checker=0, skip1=0, skip2=0, bigval=1000000
real(DP) :: ysum=0,sum_new=0,change=0,val=0,lcladj
real(DP), dimension(comp) :: phi,x,y,phiL,phiV,K

! If the previous iteration failed, don't use its value
! to seed the next run, instead just use the value that
! was provided in the input file.  Also, reset adjust.
if (failed==1) then
   if (type==0) then
      val = P
   else
      val = T
   endif
   call resetadj(lcladj)
else
   val = seed
   lcladj=adjust
endif

! Reset all values for the new iteration
failed=0;checker=0;totalitersin=0;totalitersout=0;
numadj=0;jump=0;case=0;skip1=0; skip2=0

! zloop runs the vle calculation for a single step of z
infloop: do while (.TRUE.)

  ! skip1 = 0 for a new z step or if sum didn't change but /= 1
  if (skip1==0) then
     ! skip2 = 0 only if we are beginning a new z step
      if (skip2==0) then
         ! Initialize everything for a new iteration step.
         phiL(:)=0.0d0; phiV(:)=0.0d0; phi(:)=0.0d0
         ! x(i)=z(i) bc this is BUBBLEPT and L=1,V=0
         ! z(1)+z(2)=1 ==> x(1)+x(2)=1
         x(1) = zval; x(2) = 1.0d0 - x(1)

         ! STAGE 3: ASSUME VALUES FOR y_i
         y(1)=x(1); y(2)=x(2);
         ! Call divchecker with div=1 to check for poles of x
         call divchecker(1,x,y)
   ! Call divchecker with div=2 to check for poles of y
   call divchecker(2,x,y)
endif

! The loop returns here when skip2=1 and skip1=0.
! This corresponds to a constant value of ysum that
! does not equal 1.  val has just been adjusted using
! lcladj if this isn't the first iteration of
! the infloop.

! If totalitersin exceeds a maximum value, modify adjust
! to increase chance of convergence.
! If this fails, terminate the iteration properly.
! Skip this for the first value to reduce the risk
! of exiting due to a poor seed value.
! A poor seed value should instead be handled by taking
! large iteration steps as used in "outer loop divergence"
if (totalitersin > maxiters.and.zval.ne.0.0d0) then
   ! Call the noconv subroutine with case=0. Exit if
   ! failed.  Otherwise, continue with new adjust value.
   case=0; call noconv(x,lcladj,val)
   if (failed==1) then
      bubval=bigval; exit infloop
   else
      totalitersin=0;totalitersout=0;checker=0
   endif
endif

! STAGE 4: Calculate the Liquid Fugacity Coeffs,
! phiL(1) and phiL(2)
phase=1;
call eos(phase,T,P,val,x,phi)
! If phi values are invalid, exit iteration with case=4.
if (phi(1)/=phi(1).or.phi(2)/=phi(2)) then
   case=4; call noconv(x,lcladj,val)
   bubval=bigval;exit infloop
endif

! If roots did not converge, exit iteration properly
! with case=2.
if (failed==1) then
   case=2; call noconv(x,lcladj,val);
   bubval=bigval; exit infloop
endif
phiL=phi

! STAGE 4: Calculate the Vapor Fugacity Coeffs,
    ! phiV(1) and phiV(2)
    phase=2;
    call eos(phase,T,P,val,y,phi)
    ! If phi values are invalid, exit iteration with case=4.
    if (phi(1)/=phi(1).or.phi(2)/=phi(2)) then
       case=4; call noconv(x,lcladj,val)
       bubval=bigval;exit infloop
    endif

    ! If roots did not converge, exit iteration properly
    ! with case=2.
    if (failed==1) then
       case=2; call noconv(x,lcladj,val);
       bubval=bigval; exit infloop
    endif
    phiV=phi

    ! STAGE 5: Calculate phiL/phiV and a new guess for y
    ysum=0
    K(1)=phiL(1)/phiV(1); K(2)=phiL(2)/phiV(2)
    call newcomposition(K,x,y)
    ysum=y(1)+y(2)

    ! Call divchecker to check the new values of y
    ! and perturb if required.
    call divchecker(2,x,y)

endif


! If skip1=1 the loop picks up here.
! This occurs during the continuation of a previous
! z step when the sum is still changing.

! Check if the outer loop has exceeded maxiters and
! handle accordingly.
if (totalitersout > maxiters) then
    case=0; call noconv(x,lcladj,val)
    if (failed==1) then
       bubval=bigval; exit infloop
    endif
    totalitersout=0; totalitersin=0; checker=0
    skip1=1; skip2=0; cycle infloop
end if

! STAGE4-5: Recalculate Phi with the new y value
skip1=0; skip2=0;
```

```
phase=2;
call eos(phase,T,P,val,y,phi)
! If phi values are invalid, exit iteration with case=4.
if (phi(1)/=phi(1).or.phi(2)/=phi(2)) then
   case=4; call noconv(x,lcladj,val)
   bubval=bigval;exit infloop
endif

! If roots did not converge, exit iteration properly
! with case=2.
if (failed==1) then
   case=2; call noconv(x,lcladj,val);
   bubval=bigval; exit infloop
endif
phiV=phi

K(1)=phiL(1)/phiV(1); K(2)=phiL(2)/phiV(2)
call newcomposition(K,x,y)
sum_new=y(1)+y(2)
change = sum_new - ysum

! Call divchecker with div=2 to check for y poles
! and perturb if required.
call divchecker(2,x,y)

! STAGE 6: IS YSUM STAYING CONSTANT?
! Now we can check if the sum is changing and if it is = 1.
! If the sum hasn't changed and is 1, we store this P value.
! If the sum hasn't changed but isn't 1, we will adjust P.
! We will use checker in this step to make sure we don't get
! stuck in a loop, continually incr then decr P.
! If the sum has changed, we will adjust our y values.
if (abs(change) < conv) then ! If sum has not changed

   ! STAGE 7: IS YSUM=1?
   if (abs(sum_new-1.0d0) <= conv) then
      ! STAGE 8: OUTPUT VALUES - DONE!
      bubval = val; yval = y(1); adjust=lcladj;
      exit infloop

   else ! If sum_new doesn't converge to 1
      ! STAGE 10: ITERATE T OR P AND RETURN TO STAGE 3
      if (sum_new < 1) then ! if sum < 1, decr P, incr T
         if(checker == 20) then
            ! We just incr P/decr T, decrease adjust
            case=5; call noconv(x,lcladj,val)
         end if
```

```
            checker=10
            ! Decrease P or Incr T, preventing negative values
            if(val-lcladj <0)then
               val=val/2.0d0
            else
               val = val - lcladj
            endif
            totalitersin=totalitersin+1
            skip1=0;skip2=1; cycle infloop

         else ! if sum > 1, incr P, decr T
            if(checker == 10) then
               ! We just decr P/incr T, decrease adjust
               case=5; call noconv(x,lcladj,val)
            end if
            checker=20
            ! Increase P or Decr T, preventing negative values
            if(val+lcladj <0)then
               val=val/2.0d0
            else
               val = val + lcladj
            endif
            totalitersin=totalitersin+1
            skip1=0;skip2=1; cycle infloop
         end if
      end if

   else ! If sum has changed, find new y
      ! STAGE 9: SUM IS CHANGING, NORMALIZE Y
      totalitersout=totalitersout+1
      ysum=sum_new
      call newcomposition(K,x,y)
      y(1) = y(1)/ysum
      y(2) = y(2)/ysum

      ! Call divchecker with div=2 to check new y values
      ! for poles and perturb.
      call divchecker(2,x,y)

      ! If the method is not converging it could be because
      ! the adjust parameter is just too large.
      ! Only decrease lcladj if it remains larger than conv.
      ! Otherwise, val may be forced to jump before maxiters.
      if(totalitersout==maxiters/100.and.abs(lcladj)>=conv)then
         case=0; call noconv(x,lcladj,val)
      endif
      ! Try decreasing adjust twice.
```

```
        if(totalitersout==maxiters/10.and.abs(lcladj)>=conv)then
           case=0; call noconv(x,lcladj,val)
           if(abs(lcladj)>=conv)call noconv(x,lcladj,val)
        endif
        ! Decrease multiple times as maxiters is approached to
        ! avoid reaching that point.  This will ensure non-
        ! convergence is not due to a bad adjust value before
        ! a large jump of val is taken.
        if (totalitersout>(maxiters/1)-5.and.abs(lcladj)>=conv)then
           case=0; call noconv(x,lcladj,val)
           if(abs(lcladj)>=conv)call noconv(x,lcladj,val)
           if(abs(lcladj)>=conv)call noconv(x,lcladj,val)
        endif
        checker = 0; skip1=1; cycle infloop;
     end if
  end do infloop
end subroutine vlebub


subroutine vledew(T,P,seed,zval,xseed,adjust,dewval,xval)
  use nrtype; use global; use eosmod; use convfail
  implicit none

  ! This module calculates the dew points of the
  ! vapor-liquid equilibrium of a mixture using the Phi-Phi
  ! method is used with the Equation of State, Mixing Rules
  ! and Combining Rules that are specified.

  real(DP), intent(IN) :: T,P
  real(DP), intent(IN) :: seed,zval,xseed
  real(DP), intent(INOUT) :: adjust
  real(DP), intent(OUT) :: dewval,xval
  integer(I4B) :: i,phase,checker=0,totalitersin=0, &
       totalitersout=0,bigval=1000000,skip1=0,skip2=0
  real(DP) :: xsum=0,sum_new=0,change=0,val=0,lcladj
  real(DP), dimension(comp) :: phi,x,x_new,y,phiL,phiV,K

  if (failed==1) then
     if (type==0) then
        val = P
     else
        val = T
     endif
     call resetadj(lcladj)
  else
     val = seed
     lcladj=adjust
```

```
endif

failed=0;checker=0;totalitersin=0;totalitersout=0;
numadj=0;jump=0;case=0; skip1=0; skip2=0

infloop: do while (.TRUE.)
  if (skip1==0) then
      if (skip2==0) then
         phiL(:)=0.0d0; phiV(:)=0.0d0; phi(:)=0.0d0
         ! y(i)=z(i) bc this is DEWPT and L=0,V=1
         ! z(1)+z(2)=1, y(1)+y(2)=1
         y(1) = zval; y(2) = 1.0d0 - y(1)
         x(1)=xseed; x(2)=1.0d0-x(1);  ! Guess values for x(i)
         ! Call divchecker with div=3 to check for poles of y
         call divchecker(3,x,y)
         ! Call divchecker with div=4 to check for poles of x
         call divchecker(4,x,y)
      endif

      if (totalitersin > maxiters.and.zval.ne.0.0d0) then
         case=0; call noconv(y,lcladj,val)
         if (failed==1) then
            dewval=bigval; exit infloop
         else
            totalitersin=0;totalitersout=0;checker=0
         endif
      endif

      ! STAGE 4: Calculate the Liquid Fugacity Coeffs,
      ! phiL(1) and phiL(2)
      phase=1
      call eos(phase,T,P,val,x,phi)
      if (phi(1)/=phi(1).or.phi(2)/=phi(2)) then
         case=4; call noconv(y,lcladj,val)
         dewval=bigval;exit infloop
      endif
      if (failed==1) then
         case=2; call noconv(y,lcladj,val);
         dewval=bigval; exit infloop
      endif
      phiL=phi

      ! STAGE 4: Calculate the Vapor Fugacity Coefficients,
      ! phiV(1) and phiV(2)
      phase=2
      call eos(phase,T,P,val,y,phi)
      if (phi(1)/=phi(1).or.phi(2)/=phi(2)) then
```

```
            case=4; call noconv(y,lcladj,val)
            dewval=bigval;exit infloop
         endif

         if (failed==1) then
            case=2; call noconv(y,lcladj,val);
            dewval=bigval; exit infloop
         endif
         phiV=phi
         if (DEBUG) print *, 'phiV1', phiV

         ! STAGE 5: Calculate phiL/phiV and a new guess for x
         xsum=0
         kloop1: do i=1,comp,1
            K(i) = phiL(i)/phiV(i)
            x(i) = y(i)/K(i)
            xsum=xsum+x(i)
         end do kloop1
         ! Call divchecker with div=4 to check for x poles
         ! and perturb
         call divchecker(4,x,y)
      endif

      if (totalitersout > maxiters) then
         case=0; call noconv(y,lcladj,val)
         if (failed==1) then
            dewval=bigval; exit infloop
         else
            totalitersout=0; totalitersin=0; checker=0
            skip1=1; skip2=0; cycle infloop
         endif
      end if

      ! STAGE4-5: Recalculate Phi with new x value
      skip1=0; skip2=0;
      phase=1
      call eos(phase,T,P,val,x,phi)

      if (phi(1)/=phi(1).or.phi(2)/=phi(2)) then
         case=4; call noconv(y,lcladj,val)
         dewval=bigval;exit infloop
      endif

      if (failed==1) then
         case=2; call noconv(x,lcladj,val);
         dewval=bigval; exit infloop
      endif
```

```
phiL=phi

sum_new=0
kloop2: do i=1,comp,1
   K(i) = phiL(i)/phiV(i)
   x_new(i) = y(i)/K(i)
   sum_new=sum_new+x_new(i)
end do kloop2
change = sum_new - xsum
call divchecker(4,x,y)

! STAGE 6: IS XSUM STAYING CONSTANT?
! Now we can check if the sum is changing and if it is = 1.
! If the sum hasn't changed and is 1, we store this P value.
! If the sum hasn't changed but isn't 1, we will adjust P.
! We will use checker in this step to make sure we don't get
! stuck in a loop, continually incr then decr P.
! If the sum has changed, we will adjust our x values.
if (abs(change) < conv) then ! If sum has not changed

   ! STAGE 7: IS XSUM=1?
   if (abs(sum_new-1.0d0) <= conv) then
      ! STAGE 8: OUTPUT VALUES - DONE!
      dewval = val; xval = x_new(1); adjust=lcladj;
      exit infloop

   else ! If sum_new doesn't converge to 1
      ! STAGE 10: ITERATE T OR P AND RETURN TO STAGE 3
      if (sum_new < 1) then ! if sum < 1, incr P/decr T
         if(checker == 20) then
            case=5; call noconv(y,lcladj,val)
         end if
         checker = 10
         ! Increase P or Decr T, preventing negative values
         if(val+lcladj <0)then
            val=val/2.0d0
         else
            val = val + lcladj
         endif
         totalitersin=totalitersin+1
         skip1=0;skip2=1; cycle infloop

      else ! If sum > 1, decr P/incr T
         if(checker == 10) then
            case=5; call noconv(y,lcladj,val)
         end if
         checker = 20
```

```
          ! Decrease P or Incr T, preventing negative values
          if(val-lcladj <0)then
             val=val/2.0d0
          else
             val = val - lcladj
          endif
          totalitersin=totalitersin+1
          skip1=0;skip2=1; cycle infloop
       end if
    end if

 else ! If sum has changed, find new x
    ! STAGE 9: SUM IS CHANGING, NORMALIZE X
    totalitersout=totalitersout+1
    xsum=sum_new
    x(1)=x_new(1)
    x(2)=x_new(2)

    ! Call divchecker with div=4 to check for x poles
    ! and perturb.
    call divchecker(4,x,y)

    ! If the method is not converging it could be because
    ! the adjust parameter is just too large.
    if(totalitersout==maxiters/100)then
       case=0; call noconv(y,lcladj,val)
    endif
    ! Try decreasing adjust twice.
    if(totalitersout==maxiters/10)then
       case=0; call noconv(y,lcladj,val)
       case=0; call noconv(y,lcladj,val)
    endif
    ! Decrease multiple times as maxiters is approached to
    ! avoid reaching that point.  This will ensure non-
    ! convergence is not due to a bad adjust value before
    ! a large jump of val is taken.
    if (totalitersout>maxiters/1-5)then
       case=0; call noconv(y,lcladj,val)
       case=0; call noconv(y,lcladj,val)
       case=0; call noconv(y,lcladj,val)
    endif

    if (failed==1) then
       dewval=bigval; exit infloop
    endif

    skip1=1; checker=0; cycle infloop
```

```
      end if
    end do infloop

  end subroutine vledew



  subroutine divchecker(div,x,y)
    use nrtype; use global
    implicit none

    ! This subroutine will perform sanity checks to make sure
    ! the values in the vlecalcs are not diverging.  If they
    ! are, data is stored and corrections are made to prevent
    ! the breakdown of the calculations.

    integer(I4B), intent(IN) :: div
    real(DP), dimension(:), intent(INOUT) :: x,y

    if(mixrules(1)>0.or.mixrules(2)>0)then
       if(comrules(1)==3) then

          if (div==1) then
             ! Check for a pole with the liquid compositions
             ! in bubcalc.  If one is found here, the combining
             ! rule should be reset.
             if (abs(x(1)*k12+x(2)*k21)<=abs(pole))then
                comrules(1)=newcomrule
             endif
          elseif (div==2) then
             ! Check for a pole with the vapor compositions in
             ! bubcalc.  Also, check if the fugacity coefficient
             ! is diverging.  If either is found here, a small
             ! perturbation of y is needed.
             if (abs(y(1)*k12+y(2)*k21)<=abs(pole)) then
                y(1)=y(1)*perturb; y(2)=y(2)*perturb
             elseif (logdiv==1) then
                y(1)=y(1)*perturb; y(2)=y(2)*perturb
             endif
          elseif (div==3) then
             ! Check for a pole with the vapor compositions in
             ! dewcalc.  If one is found here, the combining rule
             ! should be reset.
             if (abs(y(1)*k12+y(2)*k21)<=abs(pole)) then
                comrules(1)=newcomrule
             endif
          elseif (div==4) then
```

```
         ! Check for a pole with the liquid compositions
         ! in dewcalc.  If one is found here, a small
         ! perturbation of x is needed.
         if (abs(x(1)*k12+x(2)*k21)<=abs(pole)) then
            x(1)=x(1)*perturb; x(2)=x(2)*perturb
         endif

      endif
    endif
  endif

  ! Check the final values
  if (div==1.or.div==2) then
    if (y(1)<conv.and.y(2)<conv) then
      y(1)=0.1d0
      y(2)=0.1d0
    endif
  elseif (div==3.or.div==4) then
    if (x(1)<conv.and.x(2)<conv) then
      x(1)=0.1d0
      x(2)=0.1d0
    endif
  endif

end subroutine divchecker



subroutine newcomposition(K,fc,cc)
  use nrtype; use global;
  implicit none

  real(DP), dimension(:), intent(IN) :: K
  real(DP), dimension(:), intent(INOUT) :: fc, cc

  integer(I4B) :: kdiverg
  kdiverg=10**9

  if (K(1)>kdiverg.and.K(2)<kdiverg)then
    cc(2)=fc(2)*K(2); cc(1)=1.0d0-cc(2)
  elseif (K(1)<kdiverg.and.K(2)>kdiverg) then
    cc(1)=fc(1)*K(1); cc(2)=1.0d0-cc(1)
  elseif (K(1)>kdiverg.and.K(2)>kdiverg) then
    cc(1)=fc(1); cc(2)=fc(2)
  elseif (K(1)<kdiverg.and.K(2)<kdiverg) then
    cc(1)=fc(1)*K(1); cc(2)=fc(2)*K(2)
  endif
  end subroutine newcomposition


end MODULE vlecalcs
```

L.2 *EOSMOD.F90*

```
MODULE eosmod

  ! This module contains all the subroutines necessary to set an
  ! equation of state, and its parameters.

CONTAINS

  subroutine eos(phase,T,P,val,x,phi)
    use nrtype; use nrutil, only : nrerror; use global; use mixing;
    use combining; use solver; use kijmod;
    implicit none

    ! This subroutine checks the value of eq and uses it to set
    ! the equation of state.

    integer(I4B), intent(IN) :: phase
    real(DP), intent(IN) :: T,P,val
    real(DP), dimension(:), intent(IN) :: x
    real(DP), dimension(:), intent(OUT) :: phi

    integer(I4B) :: i=0
    real(DP) :: temp, press,A_st,B_st
    real(DP), dimension(8) :: aijs(8), bijs(8), alphaijs(8), &
         a_alphaijs(8),aa_st1(8),aa_st2(8)

    logdiv=0
    ! Set the local pressure and temperature values.
    if (type==0) then
       press = val
       temp = T
    else
       press = P
       temp = val
    endif

    ! Initialize the parameter arrays
```

```
aijs(:)=0.0d0; bijs(:)=0.0d0
alphaijs(:)=0.0d0; a_alphaijs(:)=0.0d0

! Set the values of a11, a22, b11, b22 based on the specified EOS
! where pijs(1) = p11, pijs(2) = p22, pijs(3) = p12, pijs(4) = p21
if (eq==1) then
   call prparams(T,P,val,aijs,bijs,alphaijs)
elseif (eq==0) then
   call rksparams(T,P,val,aijs,bijs,alphaijs)
elseif (eq==2) then
   call prsvparams(T,P,val,aijs,bijs,alphaijs)
elseif (eq==3) then
   call tbsparams(T,P,val,aijs,bijs,alphaijs)
endif


! Determine parameter values using the specified combining rules.
! Do this for a_alpha, and b values separately,
! where valijs(1) = val11, (2) = 22, (3) = 12, (4) = 21
! (5) = n* d(val12)/dn1, (6) n* d(val21)/dn1
! (7) = n* d(val12)/dn2, (8) n* d(val21)/dn2
! NOTE: The derivatives returned are all multiplied by n
a_alphaijs(1) = aijs(1)*alphaijs(1)
a_alphaijs(2) = aijs(2)*alphaijs(2)
do i=1,8
   aa_st1(i) = a_alphaijs(i)
end do
call combine(comrules(1), x, a_alphaijs)
call combine(comrules(2), x, bijs)
do i=1,8
   aa_st2(i) = a_alphaijs(i)
end do

! Use the mixing rules to determine the mixture parameters.
! mixvals(1) = pmix
! mixvals(2) = d(np)/dn1
! mixvals(3) = d(np)/dn2
! mixvals(4) = 1/n* d(n^2 p)/dn1
! mixvals(5) = 1/n* d(n^2 p)/dn2
! mixvals(6) = 1/n* d(n^2 p^2)/dn1
! mixvals(7) = 1/n* d(n^2 p^2)/dn2
! NOTE: The derivatives returned are multiplied by 1/n
call mixrule(mixrules(1), x, a_alphaijs, a_alphavals)
call mixrule(mixrules(2), x, bijs, bvals)

! Define the parameters for the compressibility equation
! coefficients
```

```
    A = a_alphavals(1) * press / (R_D * temp)**2
    B = bvals(1) * press / (R_D * temp)

    ! Call the specified Equation of State
    if (eq==1) then
       call pengrob(phase,T,P,val,phi)
    elseif (eq==0) then
       call rks(phase,T,P,val,phi)
    elseif (eq==2) then
       call prsv(phase,T,P,val,x,phi)
    elseif (eq==3) then
       call tbs(phase,T,P,val,phi)
    endif

  end subroutine eos


  subroutine prparams(T,P,val,aijs,bijs,alphaijs)
    use nrtype; use global;
    implicit none

    ! Program to calculate the parameters for
    ! the Peng-Robinson Equation of State.

    real(DP), intent(IN) :: T,P,val
    real(DP), dimension(:), intent(INOUT) :: aijs, bijs, alphaijs

    integer(I4B) :: i=0
    real(DP) :: temp, press
    real(DP), allocatable :: n(:),Pr(:),Tr(:)

    allocate(n(comp),Pr(comp),Tr(comp))
    if (type==0) then
       press = val
       temp = T
    else
       press = P
       temp = val
    endif

    ! Set  the parameters for PR EOS
    ! pijs(1) = p11, pijs(2) = p22
    params: do i=1,comp,1
       Pr(i) = press/(Pc(i)); Tr(i) = temp/(Tc(i))
       n(i) = 0.37464d0 + w(i)*(1.54226d0 - w(i)*0.26992d0)
       alphaijs(i) = (1.0d0 + n(i) * (1.0d0 - sqrt(Tr(i))))**2
       aijs(i) = 0.45724d0 * ((R_D*Tc(i))**2)/(Pc(i))
```

```
      bijs(i) = 0.07780d0 * R_D*Tc(i)/(Pc(i))
    end do params

    deallocate(n,Pr,Tr)
  end subroutine prparams





  subroutine pengrob(phase,T,P,val,phi)
    use nrtype; use nrutil, only : nrerror; use global; use solver
    implicit none

    ! Program to calculate the partial fugacity coefficient
    ! using the Peng-Robinson Equation of State.

    integer(I4B), intent(IN) :: phase
    real(DP), intent(IN) :: T,P,val
    real(DP), dimension(:), intent(OUT) :: phi
    real(DP) :: temp, press, Z, fln=0
    complex(DPC), dimension(eosorder) :: roots
    complex(DPC), dimension(eosorder+1) :: eoscoeffs
    logical :: polish=.true.

    if (type==0) then
      press = val
      temp = T
    else
      press = P
      temp = val
    endif

    ! Use Laguerre's Method with polishing to determine the roots
    ! of the Peng-Robinson compressibility equation.
    !
    ! eqn = Z**3 - (1-Bmix)*Z**2 + (Amix-3*Bmix**2-2*Bmix)*Z -
    ! (Amix*Bmix-Bmix**2-Bmix**3)

    eoscoeffs(1) = -B*(A-B*(1.0d0 + B))
    eoscoeffs(2) = A-B*(2.0d0 + 3.0d0 * B)
    eoscoeffs(3) = B - 1.0d0
    eoscoeffs(4) = 1.0d0

    call zroots(eoscoeffs,roots,polish)
    call pickrealpos(phase,roots,Z)
    ! If the convergence of the roots failed,
    ! exit this iteration reasonably.
    if (failed==1)return

    ! Definition of mixvals array for parameter p
    ! mixvals(1) = pmix
    ! mixvals(2) = d(np)/dn1, mixvals(3) = d(np)/dn2
    ! mixvals(4) = 1/n* d(n^2 p)/dn1
    ! mixvals(5) = 1/n* d(n^2 p)/dn2
    ! mixvals(6) = 1/n* d(n^2 p^2)/dn1
    ! mixvals(7) = 1/n* d(n^2 p^2)/dn2
    ! NOTE: The derivatives returned are multiplied by 1/n

    ! Define fln as the repeated log term in the fug coeff to
    ! simplify phi expressions
    fln = LOG((Z + B*(1.0d0+sqrt(2.0d0))) / (Z+B*(1.0d0-sqrt(2.0d0))))

    phi(1) = EXP(-LOG(Z-B) + bvals(2) * (B/(bvals(1)*(Z-B))+ &
          A/(4.0d0*B*bvals(1)*sqrt(2.0d0))*fln + &
          A*(B-Z)/(2.0d0*bvals(1)*(Z**2+2.0d0*B*Z-B**2))) - &
          a_alphavals(4)*(fln/(2.0d0*bvals(1)*R_D*temp*sqrt(2.0d0)))+ &
          bvals(6) * (A*fln/(8.0d0*B*bvals(1)**2*sqrt(2.0d0))-
          A*(Z+B)/(4.0d0*bvals(1)**2*(Z**2+2.0d0*B*Z-B**2))))

    phi(2) = EXP(-LOG(Z-B) + bvals(3) * (B/(bvals(1)*(Z-B))+ &
          A/(4.0d0*B*bvals(1)*sqrt(2.0d0))*fln+ &
          A*(B-Z)/(2.0d0*bvals(1)*(Z**2+2.0d0*B*Z-B**2))) - &
          a_alphavals(5)*(fln/(2.0d0*bvals(1)*R_D*temp*sqrt(2.0d0)))+ &
          bvals(7) * (A*fln/(8.0d0*B*bvals(1)**2*sqrt(2.0d0))- &
          A*(Z+B)/(4.0d0*bvals(1)**2*(Z**2+2.0d0*B*Z-B**2))))

end subroutine pengrob


subroutine rksparams(T,P,val,aijs,bijs,alphaijs)
   use nrtype; use global;
   implicit none

   ! Program to calculate the parameters for
   ! the Redlich-Kwong-Soave Equation of State.

   real(DP), intent(IN) :: T,P,val
   real(DP), dimension(:), intent(INOUT) :: aijs, bijs, alphaijs

   integer(I4B) :: i=0
   real(DP) :: temp, press
   real(DP), allocatable :: n(:),Pr(:),Tr(:)
```

```
  allocate(n(comp),Pr(comp),Tr(comp))
  temp=T
  press=P
  if (type==0) then
     press = val
  else
     temp = val
  endif

  ! Set Parameters for RKS EOS
  ! pijs(1) = p11, pijs(2) = p22
  params: do i=1,comp,1
     Pr(i) = press/(Pc(i)); Tr(i) = temp/(Tc(i))
     n(i) = 0.48508d0 + w(i) * (1.55171d0 - w(i) * 0.15613d0)
     alphaijs(i) = (1.0d0 + n(i) * (1.0d0 - sqrt(Tr(i))))**2
     aijs(i) = 0.42747d0 * ((R_D*Tc(i))**2)/(Pc(i))
     bijs(i) = 0.08664d0 * R_D*Tc(i)/(Pc(i))
  end do params
  deallocate(n,Pr,Tr)
end subroutine rksparams


subroutine rks(phase,T,P,val,phi)
  use nrtype; use nrutil, only : nrerror; use global; use solver;
  implicit none


  ! Program to calculate the partial fugacity coefficient
  ! using the RKS Equation of State.

  integer(I4B), intent(IN) :: phase
  real(DP), intent(IN) :: T,P,val
  real(DP), dimension(:), intent(OUT) :: phi

  real(DP) :: temp,press,Z
  complex(DPC), dimension(eosorder) :: roots
  complex(DPC), dimension(eosorder+1) :: eoscoeffs
  logical :: polish=.true.

  temp=T
  press=P
  if (type==0) then
     press = val
  else
     temp = val
  endif
  ! Use Laguerre's Method with polishing to determine the roots
  ! of the RKS compressibility equation.
  !
  ! eqn = Z**3 - (1.0d0)*Z**2 + (Amix-3*Bmix**2-2*Bmix)*Z -
  ! (Amix*Bmix-Bmix**2-Bmix**3)

  eoscoeffs(1) = -A*B
  eoscoeffs(2) = A - B*(1.0d0 + B)
  eoscoeffs(3) = -1.0d0
  eoscoeffs(4) = 1.0d0

  call zroots(eoscoeffs,roots,polish)
  call pickrealpos(phase, roots, Z)
  ! If the convergence of the roots failed,
  ! exit this iteration reasonably.
  if (failed==1)return

  ! Definition of mixvals array for parameter p
  ! mixvals(1) = pmix
  ! mixvals(2) = d(np)/dn1, mixvals(3) = d(np)/dn2
  ! mixvals(4) = 1/n* d(n^2 p)/dn1
  ! mixvals(5) = 1/n* d(n^2 p)/dn2
  ! mixvals(6) = 1/n* d(n^2 p^2)/dn1
  ! mixvals(7) = 1/n* d(n^2 p^2)/dn2
  ! NOTE: The derivatives returned are multiplied by 1/n

  phi(1) = EXP(-LOG(Z-B) + bvals(2) * (B/(bvals(1)*(Z-B))- &
       A/(bvals(1)*(Z+B)) + (A/(B*bvals(1)))*LOG(1+B/Z))- &
       1/(R_D*temp*bvals(1))*LOG(1+B/Z) * a_alphavals(4))
  phi(2) = exp(-LOG(Z-B) + bvals(3) * (B/(bvals(1)*(Z-B))- &
       A/(bvals(1)*(Z+B)) + A/(bvals(1)*B)*LOG(1+B/Z))- &
       1/(R_D*temp*bvals(1))*LOG(1+B/Z) * a_alphavals(5))

end subroutine rks


subroutine prsvparams(T,P,val,aijs,bijs,alphaijs)
  use nrtype; use global;
  implicit none

  ! Program to calculate the parameters for
  ! the PRSV Equation of State.

  real(DP), intent(IN) :: T,P,val
  real(DP), dimension(:), intent(INOUT) :: aijs, bijs, alphaijs
  integer(I4B) :: i=0
  real(DP) :: temp, press
  real(DP), allocatable :: kap0(:),kappa(:),Pr(:),Tr(:)

  allocate(kap0(comp),kappa(comp),Pr(comp),Tr(comp))
  if (type==0) then
     press = val
     temp = T
  else
     press = P
     temp = val
  endif

  ! Set Parameters for PRSV EOS
  ! pijs(1) = p11, pijs(2) = p22
  params: do i=1,comp,1
     Pr(i) = press/(Pc(i)); Tr(i) = temp/(Tc(i))
     kap0(i) =  0.378893d0 + w(i)*(1.4897153d0 - &
          w(i)*(0.17131848d0 - w(i)*0.0196544d0))
     kappa(i) = kap0(i) + (kap1(i) + kap2(i)*(kap3(i)- &
          Tr(i))*(1.0d0-sqrt(Tr(i))))*(1.0d0+ &
          sqrt(Tr(i)))*(0.7d0-Tr(i))
     alphaijs(i) = (1.0d0 + kappa(i) * (1.0d0 - sqrt(Tr(i))))**2
     aijs(i) = 0.457235d0 * ((R_D*Tc(i))**2)/(Pc(i))
     bijs(i) = 0.077796d0 * R_D*Tc(i)/(Pc(i))
  end do params

  deallocate(kap0,kappa,Pr,Tr)

end subroutine prsvparams



subroutine prsv(phase,T,P,val,x,phi)
  use nrtype; use nrutil, only : nrerror; use global; use solver;
  implicit none

  ! Program to calculate the partial fugacity coefficient
  ! using the Peng-Robinson-Stryjek-Vera Equation of State.

  integer(I4B), intent(IN) :: phase
  real(DP), intent(IN) :: T,P,val
  real(DP), dimension(:), intent(IN) :: x
  real(DP), dimension(:), intent(OUT) :: phi
  real(DP) :: temp,press,Z,fln=0
  complex(DPC), dimension(eosorder) :: roots
  complex(DPC), dimension(eosorder+1) :: eoscoeffs
  logical :: polish=.true.
  real(DP), dimension(7) :: term
```

```
if (type==0) then
   press = val
   temp = T
else
   press = P
   temp = val
endif

! Use Laguerre's Method with polishing to determine the roots
! of the PRSV compressibility equation.
!
! eqn = Z**3 - (1-Bmix)*Z**2 + (Amix-3*Bmix**2-2*Bmix)*Z -
! (Amix*Bmix-Bmix**2-Bmix**3)
eoscoeffs(1) = -B*(A-B*(1.0d0+B))
eoscoeffs(2) = A-B*(2.0d0 + 3.0d0*B)
eoscoeffs(3) = B - 1.0d0
eoscoeffs(4) = 1.0d0
call zroots(eoscoeffs,roots,polish)
call pickrealpos(phase,roots,Z)
! If the convergence of the roots failed,
! exit this iteration reasonably.
if (failed==1)return

! Definition of mixvals array for parameter p
! mixvals(1) = pmix
! mixvals(2) = d(np)/dn1, mixvals(3) = d(np)/dn2
! mixvals(4) = 1/n* d(n^2 p)/dn1
! mixvals(5) = 1/n* d(n^2 p)/dn2
! mixvals(6) = 1/n* d(n^2 p^2)/dn1
! mixvals(7) = 1/n* d(n^2 p^2)/dn2
! NOTE: The derivatives returned are multiplied by 1/n

! Define fln as a repeated log term in the fug coeff to
! simplify phi expressions
fln = LOG(abs((Z + B*(1.0d0+sqrt(2.0d0)))/(Z + &
      B*(1.0d0-sqrt(2.0d0)))))

phi(1) = EXP(-LOG(abs(Z-B)) + bvals(2) * (B/(bvals(1)*(Z-B))+ &
      A/(4.0d0*B*bvals(1)*sqrt(2.0d0))*fln + &
      A*(B-Z)/(2.0d0*bvals(1)*(Z**2+2.0d0*B*Z-B**2)))- &
      a_alphavals(4)*(fln/(2.0d0*bvals(1)*R_D*temp*sqrt(2.0d0)))+ &
      bvals(6) * (A*fln/(8.0d0*B*bvals(1)**2*sqrt(2.0d0))- &
      A*(Z+B)/(4.0d0*bvals(1)**2*(Z**2+2.0d0*B*Z-B**2))))

phi(2) = EXP(-LOG(abs(Z-B)) + bvals(3) * (B/(bvals(1)*(Z-B))+ &
      A/(4.0d0*B*bvals(1)*sqrt(2.0d0))*fln + &
```

```
        A*(B-Z)/(2.0d0*bvals(1)*(Z**2+2.0d0*B*Z-B**2)))- &
        a_alphavals(5)*(fln/(2.0d0*bvals(1)*R_D*temp*sqrt(2.0d0)))+ &
        bvals(7) * (A*fln/(8.0d0*B*bvals(1)**2*sqrt(2.0d0))- &
        A*(Z+B)/(4.0d0*bvals(1)**2*(Z**2+2.0d0*B*Z-B**2))))

  end subroutine prsv

end MODULE eosmod
```

L.3 *MIXING.F90*

```
MODULE mixing

   ! This module contains the subroutines used to determine
   ! the mixture parameters.

CONTAINS

  subroutine mixrule(mixtype, x, mijs, mixvals)
    use nrtype; use global
    implicit none

    integer(I4B), intent(IN) :: mixtype
    real(DP), dimension(:), intent(IN) :: x
    real(DP), dimension(:), intent(IN) :: mijs
    real(DP), dimension(:), intent(OUT) :: mixvals

    ! Call the specified Mixing Rules where
    ! mijs(1) = val11, mijs(2) = val22
    ! mijs(3) = val12, mijs(4) = val21
    ! mijs(5) = n* d(val12)/dn1, n* mijs(6) = d(val21)/dn1
    ! mijs(7) = n* d(val12)/dn2, n* mijs(8) = d(val21)/dn2
    if (mixtype==0) then
       call linmix(x, mijs, mixvals)
    elseif (mixtype==1) then
       call quadmix(x, mijs, mixvals)
    endif
  end subroutine mixrule


  subroutine linmix(x, mijs,  mixvals)
    use nrtype; use global;
    implicit none

    ! mijs(1) = val11, mijs(2) = val22
    ! mijs(3) = val12, mijs(4) = val21
  ! mijs(5) = n* d(val12)/dn1, mijs(6) = n* d(val21)/dn1
  ! mijs(7) = n* d(val12)/dn2, mijs(8) = n* d(val21)/dn2
  real(DP), dimension(:), intent(IN) :: x
  real(DP), dimension(:), intent(IN) :: mijs
  real(DP), dimension(:), intent(OUT) :: mixvals

  ! Definition of mixvals array:
  ! mixvals(1) = pmix
  ! mixvals(2) = d(np)/dn1, mixvals(3) = d(np)/dn2
  ! mixvals(4) = 1/n * d(n^2 p)/dn1
  ! mixvals(5) = 1/n * d(n^2 p)/dn2
  ! mixvals(6) = 1/n * d(n^2 p^2)/dn1
  ! mixvals(7) = 1/n * d(n^2 p^2)/dn2
  mixvals(1) = x(1)*mijs(1) + x(2)*mijs(2)
  mixvals(2) = mijs(1)
  mixvals(3) = mijs(2)
  mixvals(4) = mijs(1)*(x(1)+1.0d0) + mijs(2)*x(2) ! *n
  mixvals(5) = mijs(1)*x(1) + mijs(2)*(x(2)+1.0d0) ! *n
  mixvals(6) = 2.0d0*x(1)*mijs(1)**2+ &
       2.0d0*x(2)*mijs(1)*mijs(2) ! *n
  mixvals(7) = 2.0d0*x(1)*mijs(1)*mijs(2)+ &
       2.0d0*x(2)*mijs(2)**2 ! *n
end subroutine linmix


subroutine quadmix(x, mijs, mixvals)
  use nrtype; use global;
  implicit none
  ! mijs(1) = val11, mijs(2) = val22
  ! mijs(3) = val12, mijs(4) = val21
  ! mijs(5) = n* d(val12)/dn1, mijs(6) = n* d(val21)/dn1
  ! mijs(7) = n* d(val12)/dn2, mijs(8) = n* d(val21)/dn2
  real(DP), dimension(:), intent(IN) :: x
  real(DP), dimension(:), intent(IN) :: mijs
  real(DP), dimension(:), intent(OUT) :: mixvals

  ! Definition of mixvals array for parameter p
  ! mixvals(1) = pmix
  ! mixvals(2) = d(np)/dn1, mixvals(3) = d(np)/dn2
  ! mixvals(4) = 1/n* d(n^2 p)/dn1
  ! mixvals(5) = 1/n* d(n^2 p)/dn2
  ! mixvals(6) = 1/n* d(n^2 p^2)/dn1
  ! mixvals(7) = 1/n* d(n^2 p^2)/dn2
  mixvals(1) = x(1)**2*mijs(1)+x(1)*x(2)*mijs(3)+ &
       x(1)*x(2)*mijs(4)+x(2)**2*mijs(2)

  mixvals(2) = x(1)*mijs(1)*(2.0d0-x(1))+ &
```

```
          (x(2)**2*mijs(3)+x(1)*x(2)*mijs(5))+ &
          (x(2)**2*mijs(4)+x(1)*x(2)*mijs(6))- &
          x(2)**2*mijs(2)

     mixvals(3) = x(2)*mijs(2)*(2.0d0-x(2))+ &
          (x(1)**2*mijs(3)+x(1)*x(2)*mijs(7))+ &
          (x(1)**2*mijs(4)+x(1)*x(2)*mijs(8))- &
          x(1)**2*mijs(1)

     mixvals(4) = 2.0d0*x(1)*mijs(1)+x(2)*mijs(3)+ &
          x(1)*x(2)*mijs(5)+x(2)*mijs(4)+x(1)*x(2)*mijs(6) !*n

     mixvals(5) = 2.0d0*x(2)*mijs(2)+x(1)*mijs(3)+ &
          x(1)*x(2)*mijs(7)+x(1)*mijs(4)+x(1)*x(2)*mijs(8) !*n

     mixvals(6) = x(1)**3*(4.0d0-2.0d0*x(1))*mijs(1)**2- &
          2.0d0*x(2)**4*mijs(2)**2+4.0d0*x(1)*x(2)**2*(1.0d0- &
          x(1))*mijs(1)*mijs(2)+2.0d0*x(1)*x(2)**2*(1.0d0- &
          x(1))*(mijs(3)**2+2.0d0*mijs(3)*mijs(4)+mijs(4)**2)+ &
          x(1)**2*x(2)**2*(2.0d0*mijs(3)*mijs(5)+ &
          2.0d0*mijs(3)*mijs(6) + 2.0d0*mijs(4)*mijs(5)+ &
          2.0d0*mijs(4)*mijs(6))+2.0d0*mijs(1)*((mijs(3)+ &
          mijs(4))*x(1)**2*x(2)*(3.0d0 - 2.0d0*x(1)) + &
          x(1)**3*x(2)*(mijs(5) + mijs(6)))+ &
          2.0d0*mijs(2)*((mijs(3)+mijs(4))*x(2)**3*(1.0d0 - &
          2.0d0*x(1))+x(1)*x(2)**3*(mijs(5) + mijs(6)))


     mixvals(7) = x(2)**3*(4.0d0-2.0d0*x(2))*mijs(2)**2- &
          2.0d0*x(1)**4*mijs(1)**2 + 4.0d0*x(1)**2*x(2)*(1.0d0- &
          x(2))*mijs(1)*mijs(2)+2.0d0*x(1)**2*x(2)*(1.0d0- &
          x(2))*(mijs(3)**2+2.0d0*mijs(3)*mijs(4)+mijs(4)**2)+ &
          x(1)**2*x(2)**2*(2.0d0*mijs(3)*mijs(7)+ &
          2.0d0*mijs(3)*mijs(8) + 2.0d0*mijs(4)*mijs(7)+ &
          2.0d0*mijs(4)*mijs(8))+2.0d0*mijs(1)*((mijs(3) + &
          mijs(4))*x(1)**3*(1.0d0 - 2.0d0*x(2)) + &
          x(1)**3*x(2)*(mijs(7) + mijs(8)))+&
          2.0d0*mijs(2)*((mijs(3) + mijs(4))*x(1)*x(2)**2*(3.0d0- &
          2.0d0*x(2)) + x(1)*x(2)**3*(mijs(7) + mijs(8)))

   end subroutine quadmix

end MODULE mixing
```

L.4 *COMBINING.F90*

```
MODULE combining

  ! This module contains the subroutines used to select the
  ! correct combining rule and determine the parameters
  ! based on that rule.

CONTAINS

  subroutine combine(combtype, x, cijs)
    use nrtype; use global
    implicit none

    integer(I4B), intent(IN) :: combtype
    real(DP), dimension(:), intent(IN) :: x
    real(DP), dimension(:), intent(OUT) :: cijs

    ! Call the specified Combining Rules
    ! where cijs(1) = c11, (2) = 22, (3) = 12, (4) = 21
    ! (5) = d(c12)/dn1 *n, (6) d(c21)/dn1 *n
    ! (7) = d(c12)/dn2 *n, (8) d(c21)/dn2 *n
    if (combtype==0) then
       call arthcomb(cijs)
    elseif (combtype==1) then
       call convcomb(cijs)
    elseif (combtype==2) then
       call margcomb(x, cijs)
    elseif (combtype==3) then
       call vanlaarcomb(x, cijs)
    endif
  end subroutine combine


  subroutine arthcomb(cijs)
    use nrtype; use global;
    implicit none
    real(DP), dimension(:), intent(INOUT) :: cijs

    ! 12 and 21
    cijs(3) = (1.0d0 - k12) * 0.5d0 * (cijs(1) + cijs(2))
    cijs(4) = (1.0d0 - k21) * 0.5d0 * (cijs(1) + cijs(2))
    ! n1 derivatives of 12 and 21
    cijs(5) = 0.0d0
    cijs(6) = 0.0d0
    ! n2 derivatives of 12 and 21
    cijs(7) = 0.0d0
    cijs(8) = 0.0d0
  end subroutine arthcomb
```

```
subroutine convcomb(cijs)
  use nrtype; use global;
  implicit none
  real(DP), dimension(:), intent(INOUT) :: cijs

  ! 12 and 21
  cijs(3) = sqrt(cijs(1)*cijs(2)) * (1.0d0 - k12)
  cijs(4) = sqrt(cijs(1)*cijs(2)) * (1.0d0 - k21)
  ! n1 derivatives of 12 and 21
  cijs(5) = 0.0d0
  cijs(6) = 0.0d0
  ! n2 derivatives of 12 and 21
  cijs(7) = 0.0d0
  cijs(8) = 0.0d0
end subroutine convcomb


subroutine margcomb(x, cijs)
  use nrtype; use global;
  implicit none
  real(DP), dimension(:), intent(IN) :: x
  real(DP), dimension(:), intent(INOUT) :: cijs

  ! 12 and 21
  cijs(3) = sqrt(cijs(1)*cijs(2)) * (1.0d0 - x(1)*k12 - x(2)*k21)
  cijs(4) = cijs(3)
  ! n1 derivatives of 12 and 21
  cijs(5) = sqrt(cijs(1)*cijs(2)) * x(2) * (k21 - k12) ! /n
  cijs(6) = cijs(5)
  ! n2 derivatives of 12 and 21
  cijs(7) = sqrt(cijs(1)*cijs(2)) * x(1) * (k12 - k21) ! /n
  cijs(8) = cijs(7)
end subroutine margcomb


subroutine vanlaarcomb(x, cijs)
  use nrtype; use global;
  implicit none
  real(DP), dimension(:), intent(IN) :: x
  real(DP), dimension(:), intent(INOUT) :: cijs

  ! 12 and 21
  cijs(3) = sqrt(cijs(1)*cijs(2)) * (1.0d0 - &
       (k12*k21/(x(1)*k12 + x(2)*k21)))
  cijs(4) = cijs(3)
```

```
    ! n1 derivatives of 12 and 21
    cijs(5) = sqrt(cijs(1)*cijs(2))*k12*k21/((x(1)*k12+ &
         x(2)*k21)**2)*x(2)*(k12-k21) !/n
    cijs(6) = cijs(5)
    ! n2 derivatives of 12 and 21
    cijs(7) = sqrt(cijs(1)*cijs(2))*k12*k21/((x(1)*k12+ &
         x(2)*k21)**2)*x(1)*(k21-k12) !/n
    cijs(8) = cijs(7)
  end subroutine vanlaarcomb

end MODULE combining
```

L.5 *CONVFAIL.F90*

```
MODULE convfail

  ! This module contains subroutines that will deal with issues
  ! related to divergence encountered in the vlecalcs module.

CONTAINS

  subroutine noconv(x, lcladj, val)
    use nrtype; use global;
    implicit none
    ! This module is called when there are convergence problems in
    ! vlecalcs. This can refer to an inner or outer loop issue or even
    ! the lack of convergence to specific root values.  The subroutine
    ! should manually adjust the iteration step size if the inner loop
    ! is failing, or take a large iteration step if the outer loop is
    ! failing.  If the root convergence is the problem, the iteration
    ! step should be considered invalid for the given k12, k21
    ! parameters and the iteration step should be terminated.
    ! This should also occur after multiple attempts
    ! at adjusting for inner and outer loop divergence.
    real(DP), dimension(:), intent(IN) :: x
    real(DP), intent(INOUT) :: lcladj, val
    if (case==0) then
       ! The vle calculation is not converging, reduce stepsize.
       ! Or if lcladj<conv, try taking large jumps.
       if (abs(lcladj)<conv) then
          call resetadj(lcladj)
          call valjump(x,lcladj,val)
       else
          call redstep(lcladj)
       endif
    elseif (case==1) then
```

```
      ! Divergence is occuring in the outer loop, take a large jump
      ! Reset adjust whenever this occurs.
      call resetadj(lcladj)
      call valjump(x,lcladj,val)
   elseif (case==2) then
      ! Divergence is occuring in the rootfinder, terminate.
      call termiter(lcladj,x,val)
   elseif (case==3) then
      ! Reset the stepsize for a new set of iterations
      call resetadj(lcladj)
   elseif (case==4) then
      ! Terminate the current iteration as it is not converging.
      call termiter(lcladj,x,val)
   elseif (case==5) then
      ! Non-convergence is being caused by the inner loop stepping
      ! past the optimal point.  Reduce the stepsize as many times
      ! as is necessary here.
      call redstep(lcladj)
   endif
end subroutine noconv

subroutine valjump(x,lcladj,val)
   ! This subroutine should take a big jump up or down from the
   ! current value val.
   use nrtype; use global;
   implicit none
   real(DP), dimension(:), intent(IN) :: x
   real(DP), intent(INOUT) :: lcladj,val
   if (jump==0) then
      val = bubblevalue
      jump=1
   elseif (jump==1) then
      val = val/2.0d0
      jump=2
   elseif (jump==2) then
      val = val * 4.0d0
      jump=3
   elseif (jump==3) then
      case=1; call termiter(lcladj,x,val)
   endif
end subroutine valjump

subroutine redstep(lcladj)
   use nrtype; use global;
   implicit none
   ! This subroutine will be used to reduce the local adjust stepsize
   real(DP), intent(INOUT) :: lcladj
```

```
    lcladj = lcladj / 2.0d0
    if (VERBOSE) then
       if(INNER)print *, 'adjust decrease to:', lcladj
    endif
  end subroutine redstep


  subroutine resetadj(lcladj)
    use nrtype; use global;
    implicit none
    ! This subroutine will reset the local adjust stepsize to the
    ! global adjust value.
    real(DP), intent(INOUT) :: lcladj
    lcladj = adjust
  end subroutine resetadj


  subroutine termiter(lcladj, x, val)
    use nrtype; use global;
    implicit none
    ! This subroutine will terminate the current iteration, setting
    ! values to alert that this iteration failed and saving the kij
    ! data to a file to keep track of which combinations failed.
    real(DP), intent(IN) :: lcladj
    real(DP), dimension(:), intent(IN) :: x
    real(DP), intent(OUT) :: val

    if (case==0) then
       ! Divergence is occuring in the inner loop.
       if (VERBOSE) then
          print *, 'Inner loop did not converge with adjust:', lcladj
          print *, 'Terminating iteration.'
       endif
       open(260,file="innerloop",status='unknown',position='append')
       write(260,fmt=*)k12,k21,x(1)*k12+x(2)*k21
       close(260)
    elseif (case==1) then
       ! Divergence is occuring in the outer loop.
       if (VERBOSE) then
          print *, 'Outer loop convergence failure.'
          print *, 'Terminating iteration.'
       endif
       open(270,file="outerloop",status='unknown',position='append')
       write(270,fmt=*)k12,k21,x(1)*k12+x(2)*k21
       close(270)
       jump=0
    elseif (case==2) then
```

```
         ! Divergence is occuring in the rootfinder
         if (VERBOSE) then
            print *, 'Root finder convergence failure.'
            print *, 'Terminating iteration.'
         endif
         open(280,file="root",status='unknown',position='append')
         write(280,fmt=*)k12,k21,x(1)*k12+x(2)*k21
         close(280)
      elseif (case==4) then
         ! Divergence is occuring because phi is diverging.
         if (VERBOSE) then
            print *, 'Fugacity Coefficients are diverging'
            print *, 'Terminating iteration.'
         endif
         open(290,file="fugacity",status='unknown',position='append')
         write(290,fmt=*)k12,k21,x(1)*k12+x(2)*k21
         close(290)
      endif

      ! Set global failed to 1 to alert vlecalcs.f90 that this is the
      ! termination of a iteration.
      failed=1

      ! Store the kij values that caused this to fail.
      if (VLEDEV) then
         open(140, file='kijconvfail',status='unknown',position='append')
         write(140,fmt=*)k12,k21
         close(140)
      endif
   end subroutine termiter

end MODULE convfail
```

L.6 *KIJMOD.F90*

```
MODULE kijmod
   ! This module contains the subroutines necessary to setup
   ! the kij values used for VLE calculations.

CONTAINS

   subroutine setkij(method,rulestr,tstr,pstr)
    use nrtype; use global;
      implicit none

      ! Set kij values from the kijpairs file which was creating
! using VLEDEV by finding the min deviation over a wide
! range of kij values.

integer(I4B), intent(IN) :: method
character(len=20), intent(IN) :: rulestr
character(len=3), intent(IN) :: tstr
character(len=6), intent(IN) :: pstr
character(len=20), dimension(:), allocatable :: rulearr,strarr
integer(I4B) :: i,success=0,maxnumvars
real(DP), dimension(:), allocatable :: k12arr,k21arr,coef1,coef2

! Set maxnumvars based on the length of kijpairs and fits
maxnumvars = 30

! method=0 when mixrules(1)=0 and kij values have no effect
if(method==0)then
   k12=0.0d0; k21=0.0d0
   success=1

! method=1 when file fits should be used to set kij data
elseif(method==1)then
   allocate(rulearr(maxnumvars),strarr(maxnumvars),&
         coef1(maxnumvars),coef2(maxnumvars))
   ! Read in the fits from the file 'fits'
   open(12,file='fits',status='old',access='sequential')
   do i=1,maxnumvars
      read(12,fmt=*)rulearr(i),strarr(i),coef1(i),coef2(i)
   enddo
   close(12)

   ! Calculate kij values using the fits based on the rule
   ! combination currently being used.
   do i=1,maxnumvars
      if(rulearr(i)==rulestr)then
         if(type==0)then
            if(strarr(i)==tstr)then
               k12 = -0.1d0
               k21 = coef1(i)*k12+coef2(i)
               success = 1
            endif
         else
            if(strarr(i)==pstr)then
               k12 = -0.1d0
               k21 = coef1(i)*k12+coef2(i)
               success = 1
            endif
         endif
```

```
        endif
      enddo
      deallocate(rulearr,strarr,coef1,coef2)

   ! method=2 when specific kij pairs are in the kijpairs file.
   else
      allocate(rulearr(maxnumvars),strarr(maxnumvars),&
           k12arr(maxnumvars),k21arr(maxnumvars))
      ! Read in the values from 'kijpairs'
      open(11,file='kijpairs',status='old',access='sequential')
      do i=1,maxnumvars
         read(11,fmt=*)rulearr(i),strarr(i),k12arr(i),k21arr(i)
      enddo
      close(11)

      ! Set the correct kijvalues based on the rule combination.
      do i=1,maxnumvars
         if(rulearr(i)==rulestr)then
            if(type==0)then
               if(strarr(i)==tstr)then
                  k12 = k12arr(i); k21 = k21arr(i)
                  success = 1
               endif
            else
               if(strarr(i)==pstr)then
                  k12 = k12arr(i); k21 = k21arr(i)
                  success = 1
               endif
            endif
         endif
      enddo
      deallocate(rulearr,strarr,k12arr,k21arr)
   endif

   if(success==0)then
      print*, 'Error: kij values not set!  Terminating.'
      stop
   endif

end subroutine setkij



subroutine setk12arr(k12arr)
   use nrtype; use global
   implicit none
   ! Set the k12 array based on specified length,
    ! stepsize, and k12 value
    real(DP), dimension(:), intent(OUT) :: k12arr
    integer(I4B) :: k12pos=0, i=0
    k12pos = (k12len/2) + 1
    k12arr(k12pos)=k12
    if (k12len/=0) then
       do i=k12pos,k12len,1
          k12arr(i+1) = k12arr(i) + k12change
       enddo
       do i=k12pos,2,-1
          k12arr(i-1) = k12arr(i) - k12change
       enddo
    endif
  end subroutine setk12arr


  subroutine setk21arr(k21arr)
    use nrtype; use global
    implicit none
    ! Set the k21 array based on specified length,
    ! stepsize, and k21 value
    real(DP), dimension(:), intent(OUT) :: k21arr
    integer(I4B) :: k21pos=0, i=0
    k21pos = (k21len/2) + 1
    k21arr(k21pos)=k21
    if (k21len/=0) then
       do i=k21pos,k21len,1
          k21arr(i+1) = k21arr(i) + k21change
       enddo
       do i=k21pos,2,-1
          k21arr(i-1) = k21arr(i) - k21change
       enddo
    endif
  end subroutine setk21arr

end MODULE kijmod
```

L.7 *RULES.F90*

```
MODULE rules

  ! This module contains any subroutines that have to do
  ! with setting or varying the equation of state, the mixing
  ! rules, and the combining rules.

CONTAINS
```

```
subroutine varyrules(numeos,nummix,numcomb,numvars,rulesarr)
  use nrtype; use global;
  implicit none

  ! This subroutine will create an array called rulesarr that
  ! will contain all the possible variations of EOS, mixing
  ! and combining rules that can be used to solve the VLE.

  integer(I4B), intent(IN) :: numeos, nummix, numcomb, numvars
  integer(I4B), dimension(:,:), intent(OUT) :: rulesarr
  integer(I4B) :: i=1,n=0,o=0,q=0,r=0,s=0

  do while (i<=numvars)
     n=0;o=0;q=0;r=0;s=0
     do while (n < numeos)
        eq = n
        o = 0
        do while (o < numcomb)
           comrules(1) = o
           q=0
           do while (q < 1)
              ! Force only the arithmetic combining rule for
              ! the b parameter
              comrules(2)=q
              r=0
              do while (r < nummix)
                 mixrules(1)=r
                 s=0
                 do while (s < 1)
                    ! Force only linear mixing rules for
                    ! b parameter.
                    mixrules(2)=s
                    rulesarr(i,1) = eq
                    rulesarr(i,2) = comrules(1)
                    rulesarr(i,3) = comrules(2)
                    rulesarr(i,4) = mixrules(1)
                    rulesarr(i,5) = mixrules(2)
                    i=i+1
                    s=s+1
                 enddo
                 r=r+1
              enddo
              q=q+1
           enddo
           o=o+1
        enddo
```

```
          n=n+1
        enddo
      end do
    end subroutine varyrules

  end MODULE rules
```

L.8 *DEVCALC.F90*

```
MODULE devcalc

    ! This module contains the subroutines that will be used
    ! to calculate the deviations and percent deviations between
    ! calculated values and experimental data points.

CONTAINS

    subroutine bubdev(T,P,zval,bubval,validpt,avedev,aveperdev)
      use nrtype; use global;
      implicit none

      ! This subroutine is used by vlemain to calculate the
      ! deviations between experimental and calculated errors
      ! for the bubble point routine.

      real(DP), INTENT(IN) :: T,P
      real(DP), intent(IN) :: zval, bubval
      integer(I4B), intent(OUT) :: validpt
      real(DP), intent(OUT) :: avedev,aveperdev
      integer(I4B) :: j, points=0
      real(DP), allocatable :: diff(:),perdev(:)

      allocate(diff(lines),perdev(lines))
      diff(:)=0.0d0; perdev(:) = 0.0d0
      points = 0; validpt = 0

      ! Open a file to store the calculated data.
      open(150,file='validb.dat',status='unknown', position='append')

      ! If P-x was calculated:
      if (type == 0) then
         ! Go through the experimental data arrays and check for
         ! lines of matching T and x_1.
         do j=1,lines
            if (abs(texp(j) - T) < conv) then
               ! Compare the zval passed in to the x1 values
         ! in the experimental data for bubble point pressures
         if (abs(zval-xexp(j))<conv) then
            diff(j) = abs(bubval/1000.0d0 - pexp(j))
            points = points+int(npexp(j))
            perdev(j) = diff(j)/pexp(j)*100.0d0
            validpt=1
            write(150,fmt=*)texp(j),xexp(j),yexp(j),&
                 pexp(j),diff(j),perdev(j)
         endif
      endif
   end do

 ! If T-x was calculated:
 else
   do j=1,lines
      if (abs(pexp(j) - P/1000.0d0) < conv) then
         if (abs(zval-xexp(j))<conv) then
            diff(j) = abs(bubval-texp(j))
            points = points+int(npexp(j))
            perdev(j) = diff(j)/texp(j)*100.0d0
            validpt=1
            write(150,fmt=*)pexp(j),xexp(j),yexp(j),&
                 texp(j),diff(j),perdev(j)
         endif
      endif
   enddo
 end if

 if(validpt==1)then
   ! Calculate the average deviation and average percent deviation
   avedev = sum(diff)/points
   aveperdev = sum(perdev)/points
 endif
 close(150)
 deallocate(diff,perdev)
end subroutine bubdev

subroutine dewdev(T,P,zval,dewval,validpt,avedev,aveperdev)
 use nrtype; use global;
 implicit none
 ! This subroutine is used by vlemain to calculate the
 ! deviations between experimental and calculated errors
 ! for the dew point routine.
 real(DP), INTENT(IN) :: T,P
 real(DP), intent(IN) :: zval, dewval
 integer(I4B), intent(OUT) :: validpt
 real(DP), intent(OUT) :: avedev,aveperdev
```

```
  integer(I4B) :: j, points=0
  real(DP), allocatable :: diff(:),perdev(:)

  allocate(diff(lines),perdev(lines))
  diff(:)=0.0d0; perdev(:) = 0.0d0
  points = 0; validpt = 0

  open(150,file='validd.dat',status='unknown', position='append')

  if (type == 0) then
     do j=1,lines
        if (abs(texp(j) - T) < conv) then
           ! Compare the zval passed in to the y1 values
           ! in the experimental data for dew point pressures
           if (abs(zval-yexp(j))<conv) then
              diff(j) = abs(dewval/1000.0d0 - pexp(j))
              points = points+int(npexp(j))
              perdev(j) = diff(j)/pexp(j)*100.0d0
              validpt=1
              write(150,fmt=*)texp(j),xexp(j),yexp(j),&
                   pexp(j),diff(j),perdev(j)
           endif
        endif
     end do
  else
     do j=1,lines
        if (abs(pexp(j) - P/1000.0d0) < conv) then
           if (abs(zval-yexp(j))<conv) then
              diff(j) = abs(dewval-texp(j))
              points = points+int(npexp(j))
              perdev(j) = diff(j)/texp(j)*100.0d0
              validpt=1
              write(150,fmt=*)pexp(j),xexp(j),yexp(j),&
                   texp(j),diff(j),perdev(j)
           endif
        endif
     enddo
  end if

  if(validpt==1)then
     avedev = sum(diff)/points
     aveperdev = sum(perdev)/points
  endif
  close(150)
  deallocate(diff,perdev)
end subroutine dewdev
```

```
  subroutine dev(j,bubval,deviation,perdev)
    use nrtype; use global;
    implicit none

    ! This subroutine is used by the vledev program to calculate
    ! deviation and percent deviation.

    integer(I4B), intent(IN) :: j
    real(DP), intent(IN) :: bubval
    real(DP), intent(OUT) :: deviation,perdev

    open(150,file='valid.dat',status='unknown', position='append')
    if (type == 0) then
       deviation = abs(bubval/1000.0d0 - pexp(j))
       perdev = deviation/pexp(j)*100.0d0
       write(150,fmt=*)texp(j),xexp(j),pexp(j),deviation,perdev
    else
       deviation = abs(bubval-texp(j))
       perdev = deviation/texp(j)*100.0d0
       write(150,fmt=*)pexp(j),xexp(j),texp(j),deviation,perdev
    endif
    close(150)
  end subroutine dev

end MODULE devcalc
```

BIBLIOGRAPHY

[1] P. Wankat, *Equilibrium Staged Separations*, Elsevier Science Publishing (1988).

[2] *M & M Metals: Heat Sinks - Custom Fabricated Aluminum Extrusions - Precision Machining*, www.mmmetals.com (2009).

[3] L. Schaefer, *Single Pressure Absorption Heat Pump Analysis*, Ph.D. Thesis, Georgia Institute of Technology (2000).

[4] E. Pawlikowski, J. Newman, J. Prausnitz, *Phase Equilibriums for Aqueous Solutions of Ammonia and Carbon Dioxide*, Industrial and Engineering Chemistry Process Design and Development, **Vol. 21** pp. 764–770 (1982).

[5] R. Weir, T. Loos, *Measurement of the Thermodynamic Properties of Multiple Phases*, Gulf Professional Publishing (2005).

[6] P. Richet, *The Physical Basis of Thermodynamics with Applications to Chemistry*, Kluwer Academic Plenum Publishers (2001).

[7] E. Weisstein, *Eric Weisstein's World of Physics*, www.scienceworld.wolfram.com (2010).

[8] S. Walas, *Phase Equilibria in Chemical Engineering*, Butterworth Publishers (1985).

[9] Y. Wei, R. Sadus, *Equations of State for the Calculation of Fluid-Phase Equilibria*, Thermodynamics, **Vol. 46** pp. 169–196 (2000).

[10] M. Thiesen, *Investigations of the Equation of State*, Annalen der Physik und Chemie, **Vol. 24** pp. 467–492 (1885).

[11] H. Kamerlingh-Onnes, *Expression of the Equation of State of Gases and Liquids by Means of Series*, Communications from the Physical Laboratory at the University of Leiden, **Vol. 71** (1901).

[12] M. Assael, J. Trusler, T. Tsolakis, *Thermophysical Properties of Fluids*, Imperial College Press (1996).

[13] S. Ornstein, *Application of the Statistical Mechanics of Gibbs on Molecular-Theoretical Problems*, Communications from the Physical Laboratory at the University of Leiden (1908).

[14] J. Prausnitz, R. Lichtenthaler, E. Azevedo, *Molecular Thermodynamics of Fluid-Phase Equilibria*, Prentice Hall (1986).

[15] T. Hill, *An Introduction to Statistical Thermodynamics*, Dover Publications (1986).

[16] G. Ikonomou, M. Donohue, *Thermodynamics of Hydrogen-Bonded Molecules: The Associated Perturbed Anisotropic Chain Theory*, American Institute of Chemical Engineers Journal, **Vol. 32** p. 1716 (1986).

[17] G. Ikonomou, M. Donohue, *Extension of the Associated Perturbed Anisotropic Chain Theory to Mixtures and More Than One Associating Component*, Fluid Phase Equilibria, **Vol. 39** p. 129 (1988).

[18] I. Economou, M. Donohue, *Chemical, Quasi-Chemical and Perturbation Theories for Associating Fluids*, American Institute of Chemical Engineers Journal, **Vol. 37** p. 1875 (1991).

[19] I. Economou, M. Donohue, *Equation of State with Multiple Associating Sites for Water and Water-Hydrogen Mixtures*, Industrial and Engineering Chemistry Research, **Vol. 31** p. 2388 (1992).

[20] P. Vilmalchand, M. Donohue, *Thermodynamics of Quadropolar Molecules: The Perturbed-Anisotropic-Chain Theory*, Industrial and Engineering Chemistry Fundamentals, **Vol. 24** p. 246 (1985).

[21] P. Vilmalchand, I. Celmins, M. Donohue, *VLE Calculations for Mixtures Containing Multipolar Compounds Using the Perturbed Anisotropic Chain Theory*, American Institute of Chemical Engineers Journal, **Vol. 32** p. 1735 (1986).

[22] I. Economou, C. Peters, J. de Swaan Arons, *Water-Salt Phase Equlibria at Elevated Temperatures and Pressures: Model Development and Mixture Predictions*, Journal of Physical Chemistry, **Vol. 99** p. 6182 (1995).

[23] W. Chapman, G. Jackson, K. Gubbins, *Phase Equilibria of Associating Fluids: Chain Molecules with Multiple Bonding Sites*, Molecular Physics, **Vol. 65** p. 1057 (1988).

[24] W. Chapman, K. Gubbins, G. Jackson, M. Radosz, *New Reference Equation of State for Associating Liquids*, Industrial and Engineering Chemistry Research, **Vol. 29** p. 1709 (1990).

[25] S. Huang, M. Radosz, *Equation of State for Small, Large, Polydisperse, and Associating Molecules*, Industrial and Engineering Chemistry Research, **Vol. 29** p. 2284 (1990).

[26] M. Abbott, H. Van Ness, *Schaum's Outline: Theory and Problems of Thermodynamics*, McGraw Hill Inc. (1989).

[27] M. Trebble, P. Bishnoi, *Development of a New Four-Parameter Cubic Equation of State*, Fluid Phase Equilibria, **Vol. 35** pp. 1–18 (1987).

[28] P. Salim, M. Trebble, *A Modified Trebble-Bishnoi Equation of State: Thermodynamic Consistency Revisited*, Fluid Phase Equilibria, **Vol. 65** pp. 59–71 (1991).

[29] I. Polishuk, J. Wisniak, H. Segura, *A Novel Approach for Defining Parameters in a Four-Parameter Equation of State*, Chemical Engineering Science, **Vol. 55** pp. 5705–5720 (2000).

[30] J. Hu, R. Wang, S. Mao, *Some Useful Expressions for Deriving Component Fugacity Coefficients from Mixture Fugacity Coefficients*, Fluid Phase Equilibria, **Vol. 268** pp. 7–13 (2008).

[31] E. M. Hendriks, *Reduction theorem for phase equilibrium problems*, Industrial & Engineering Chemistry Research, **Vol. 27 (9)** pp. 1728–1732, URL http://pubs.acs.org/doi/abs/10.1021/ie00081a027 (1988).

[32] E. M. Hendriks, A. R. D. van Bergen, *Application of a reduction method to phase equilibria calculations*, Fluid Phase Equilibria, **Vol. 74** pp. 17 – 34, URL http://www.sciencedirect.com/science/article/B6TG2-43PSB3P-6V/2/324ad134dc84e81e16d5b469a32e5645 (1992).

[33] D. V. Nichita, C. F. Leibovici, *An analytical consistent pseudo-component delumping procedure for equations of state with non-zero binary interaction parameters*, Fluid Phase Equilibria, **Vol. 245 (1)** pp. 71 – 82, URL http://www.sciencedirect.com/science/article/B6TG2-4JTR93T-1/2/8f1638f248ead0a9b8986591c086eab5, proceedings of the Seventeenth European Conference on Thermophysical Properties (2006).

[34] J. Elliott, C. Lira, *Introductory Chemical Engineering Thermodynamics*, Prentice Hall (1999).

[35] H. Orbey, S. Sandler, *Modeling Vapor-Liquid Equilibria*, Cambridge University Press (1998).

[36] J. Thomas, *Carnegie Mellon University Nanoscale Transport Phenomena Wiki*, www.ntpl.me.cmu.edu (2007).

[37] D. Chrisman, J. Leach, *Intermolecular Parameters and Combining Rules for the Square Well Potential*, Industrial and Engineering Chemistry Fundamentals, **Vol. 12** pp. 423–431 (1973).

[38] J. Maxwell, *On the Dynamical Evidence of the Molecular Constitution of Bodies*, Journal of the Chemical Society, **Vol. 28** pp. 493–508 (1875).

[39] E. Guggenheim, *Thermodynamics - An Advanced Treatment for Chemists and Physicists*, Elsevier Science Publishers (1986).

[40] J. Tester, M. Modell, *Thermodynamics and Its Applications*, Prentice Hall (1997).

[41] S. Sandler, *Chemical and Engineering Thermodynamics*, John Wiley & Sons (1989).

[42] R. Stryjek, J. Vera, *PRSV: An Improved Peng-Robinson Equation of State for Pure Compounds and Mixtures*, The Canadian Journal of Chemical Engineering, **Vol. 64** pp. 323–333 (1986).

[43] R. Stryjek, J. Vera, *PRSV2: A Cubic Equation of State for Accurate Vapor-Liquid Equilibria Calculations*, The Canadian Journal of Chemical Engineering, **Vol. 64** pp. 820–827 (1986).

[44] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI - The Complete Reference: Volume 1, The MPI Core*, The MIT Press (1998).

[45] *Numerical Recipes in FORTRAN, 2nd Edition* (1992).

[46] P. Coutsikos, N. Kalospiros, D. Tassios, *Capabilities and Limitations of the Wong-Sandler Mixing Rules*, Fluid Phase Equilibria, **Vol. 108** pp. 69–78 (1998).

[47] J. Gmehling, U. Onken, W. Arlt, *Vapor Liquid Equilibrium Data Collection: Aqueous Organic Systems, Supplement 1*, Dechema (1996).

[48] J. Gmehling, U. Onken, W. Arlt, *Vapor Liquid Equilibrium Data Collection: Aqueous Organic Systems*, Dechema (1996).

[49] G. Soave, *Equilibrium Constants from a Modified Redlich-Kwong Equation of State*, Chemical Engineering Science, **Vol. 27** pp. 1197–1203 (1972).

[50] C. Torres-Marchal, A. Cantalino, R. Brito, *Prediction of Vapor-Liquid Equilibria (VLE) from Dilute Systems Data Using the SRK Equation of State: Industrial Applications*, Fluid Phase Equilibria, **Vol. 52** pp. 111–117 (1989).

[51] V. Harismiadis, N. Koutras, D. Tassios, A. Panagiotopoulos, *How good is conformal solutions theory for phase equilibrium predictions?Gibbs ensemble simulations of binary Lennard-Jones mixtures*, Fluid Phase Equilibria, **Vol. 65** pp. 1–18 (1991).

[52] *Pittsburgh Supercomputing Center*, www.psc.edu/machines/hp/c3000/warhol.php.