Implementing an Equation of State Without Derivatives: teqp

Ian H. Bell,*,† Ulrich K. Deiters,‡ and Allan M. M. Leal¶

†*Applied Chemicals and Materials Division, National Institute of Standards and Technology, Boulder, CO 80305, USA*
‡*Institute of Physical Chemistry, University of Cologne, 50939 Köln, Germany*
¶*Geothermal Energy and Geofluids Group, Institute of Geophysics, ETH Zurich, CH-8092 Zurich, Switzerland*

E-mail: ian.bell@nist.gov

Abstract

This work uses advanced numerical techniques (complex differentiation and automatic differentiation) to efficiently and accurately compute all required thermodynamic properties of an equation of state (EOS) without any analytical derivatives – particularly without any hand-written derivatives. It avoids the tedious and error-prone process of symbolic differentiation, thus allowing for more rapid development of new thermodynamic models. The technique presented here was tested with several equations of state (van der Waals, Peng-Robinson, Soave-Redlich-Kwong, PC-SAFT, cubic-plus-association) and high-accuracy multi-fluid models. A minimal set of algorithms (critical locus tracing, vapor-liquid equilibrium tracing) were implemented in an extensible and concise open-source C++ library: teqp (for Templated EQuation of state Package). This work demonstrates that highly complicated equations of state can be implemented faster yet with minimal computational overhead and negligible loss in numerical precision compared with the traditional approach that relies on analytical derivatives. We believe the approach outlined in this work has the potential to establish a new computational standard when implementing computer codes for thermodynamic models.

1 Introduction

Thermodynamic equations of state (EOS) are challenging to implement without making programming errors, especially when it comes to composition derivatives. For instance the GERG monograph[1] painstakingly works out the derivatives of the multi-fluid GERG model over the course of more than 40 pages; the implementation of these derivatives in CoolProp[2] (just the mixture part) takes on the order of a thousand lines of code. A similar exercise for simple cubic EOS[3] results in 20 pages of hand-written derivatives. The EOS derivatives published along with ThermoPack[4,5] are a few hundred pages in total. These few examples illustrate that over the last decades, a significant amount of human endeavor has been spent on working out, implementing, and debugging derivatives of equations of state. And even with all this effort, novel calculations, for instance tracing critical loci[6] and obtaining critical points[7] require significant new work.

Even with careful unit testing, many subtle bugs are possible when implementing EOS. For instance, when finite differences are used to test analytic derivative implementations, the finite differences themselves have error caused by the precision available in double precision arithmetic, and step size selection in finite differentiation is a non-trivial matter. It there-

fore becomes difficult define the "right" value to compare against in the numerical test. Subtle bugs of this nature may reside in codebases for many years.

The most direct motivation for this work was twofold: 1) to enable tracing critical loci[6,8] for multi-fluid models which requires high-order derivatives that cannot feasibly be obtained from existing computational libraries 2) to prototype mathematical models for fitting mixture thermodynamic data.

The title of this paper is tongue-in-cheek; derivatives cannot be avoided in the application of thermodynamic models. On the other hand, the goal of this paper is to demonstrate that the human capital invested in taking derivatives of equations of state need not continue. At the very least, we show that modern numerical differentiation techniques can dramatically expedite the process, particularly as pertains to testing and debugging.

Two parallel development threads are helping high-level and low-level programming languages converge. The ubiquity of high-level languages like Python has brought many high-level user-friendly concepts into lower-level languages like C++. Coming from the other direction, compilers are getting much better at supporting the development of generic code in strongly typed languages. In C++ templated functions can be leveraged to handle arbitrary numerical types without rewriting the code. The ability to write generic implementations in C++ is heavily used in this work to calculate thermodynamic properties.

The longstanding *ThermoC* program[9] has achieved a similar goal to this work, with modular implementations of thermodynamic models in C++ and the use of numeric differentiation. The idea in the current work parallels a few contemporary activities. There is the Clapeyron.jl library for Julia.[10] Preceding Clapeyron, a set of Julia libraries is available from Andrés Riedemann (`https://github.com/longemen3000/`): `LavoisierCore.jl`, `ThermoState.jl`, and `ThermoModels.jl`. There is also a library that uses Rust with Python wrappers.[11]

The open-source C++ library developed in this work, entitled `teqp` (for Templated EQuation of state Package) is written in a compiled programming language for speed, and demonstrates performance competitive with analytical derivatives. It is furthermore flexible due to its use of automatic differentiation, and is also available as a shared library with a C-compatible interface for integration into other software tools. This library is well-suited to become the beating heart of other thermodynamic property libraries.

The outline of this paper is as follows: Section 2 describes the necessary conventional thermodynamics, Section 3 explains the use of isochoric thermodynamics, Section 4 describes the differentiation tools used, Section 5 describes the implementation in code, and Section 6 demonstrates results for speed and accuracy.

2 Conventional thermodynamics

Over the years, many different EOS formulations have been developed, some are pressure-explicit (e.g., van der Waals,[12] modified Benedict-Webb-Rubin,[13]), others are Gibbs-energy explicit,[14,15] and the most accurate equations of state available today are of the multi-parameter Helmholtz energy explicit formulation. It is usually (one exception: Dieterici[16]) possible to convert from pressure-explicit to Helmholtz-energy-explicit formulations.[3,17]

In order to develop a consistent library structure, it is necessary to decide on a single model formulation and stick with it. *So, what thermodynamic formulation should be selected?* In principle any of the fundamental potentials could be selected with no loss in generality, but the Helmholtz-energy-explicit formulation is used in all the most accurate thermophysical property libraries: NIST REFPROP,[18] CoolProp,[2] and TREND.[19] Most other EOS in use today can also be converted to this form. Therefore the molar Helmholtz energy a is selected as the fundamental potential, and it is usually expressed in the form $\alpha = a/(RT)$, with

R the molar gas constant and T the temperature. The independent variables are temperature, density, and molar composition $\vec{x}$.

Further subdividing the total reduced Helmholtz energy, it is usually expressed as the sum of ideal-gas (ig) and residual (r) contributions, such that

$$\alpha^{\mathrm{tot}} = \alpha^{\mathrm{ig}} + \alpha^{\mathrm{r}} \tag{1}$$

where `teqp` concerns itself mostly with the residual contribution α^{r}. The standard thermodynamic properties may be expressed in terms of derivatives of the Helmholtz energy with the concise nomenclature

$$\Lambda^{*}_{ij} = (1/T)^i \rho^j \left(\frac{\partial^{i+j}(\alpha^*)}{\partial(1/T)^i \partial\rho^j} \right) \tag{2}$$

where $*$ is ig (ideal gas), r (residual), or tot (total), and i and j are indices indicating the derivative order w.r.t. $(1/T)$ and ρ. A table of the standard thermodynamic properties in this format is shown in Section 8.1; some helpful relations have also been included from Lemmon et al.[20] Conversions between $(1/T)$ and T derivatives are described on page 36 of Span.[17]

In the isochoric thermodynamics formalism described below, the natural variables are temperature and molar concentrations of the components. This approach breaks down in the zero density limit needed to calculate virial coefficients because the composition is still meaningful in the zero density limit, but the molar concentrations all go to zero, which removes the information about the composition (the mole fractions all become undefined values of 0/0). Therefore, the model must be implemented in terms of temperature, density, and mole fractions, and additional transformations can be used to obtain the isochoric thermodynamic derivatives.

It should also be mentioned at this juncture that it is common to develop Gibbs-energy-explicit models, for instance in aqueous systems or in solid phases. While these models are less commonly applied to fluid phase properties for nonreactive systems, a few examples in the literature exist, especially in the industrial formulation of the properties of water.[14] Gibbs-energy-explicit models can also be differentiated in the same manner as applied in this work.

2.1 Virial coefficients

Virial coefficients represent the information contained in dilute gas interactions. They represent one of the elements most strongly linked with rigorous theory in EOS development. As such, they are an important element of the thermodynamics of pure fluids and mixtures.[21]

The virial EOS for the compressibility factor $Z = p/(\rho RT)$ for low-density states is given by

$$Z = 1 + \sum_{i=2}^{\infty} B_i \rho^{i-1} \tag{3}$$

where B_i is the i-th virial coefficient and the reduced residual Helmholtz energy obtained from this virial EOS is given by

$$\alpha^{\mathrm{r}} = \int_0^{\rho} \frac{Z-1}{\rho} \mathrm{d}\rho = \sum_{i=2}^{N_{\max}} B_i \frac{\rho^{i-1}}{i-1} \tag{4}$$

$$= B_2\rho + B_3\frac{\rho^2}{2} + B_4\frac{\rho^3}{3} + \dots \tag{5}$$

A Maclaurin series expansion of α^{r} is written in the form

$$\alpha^{\mathrm{r}} = \sum_{i=1}^{N_{\max}-2} (\alpha^{\mathrm{r}})^{(i)} \frac{\rho^i}{i!} \tag{6}$$

$$= (\alpha^{\mathrm{r}})^{(1)}\rho + (\alpha^{\mathrm{r}})^{(2)}\frac{\rho^2}{2} + (\alpha^{\mathrm{r}})^{(3)}\frac{\rho^3}{6} + \dots \tag{7}$$

with the concise derivative

$$(\alpha^{\mathrm{r}})^{(i)} = \lim_{\rho\to 0} \left(\frac{\partial^i \alpha^{\mathrm{r}}}{\partial\rho^i} \right)_{T,\vec{x}} \tag{8}$$

thus equating terms, the i-th virial coefficient is given by the derivative

$$B_i = \frac{(\alpha^{\mathrm{r}})^{(i-1)}}{(i-2)!} \tag{9}$$

with the reminder that $0! = 1$ and $1! = 1$. In other words, the virial coefficients are related to the coefficients in the Taylor expansion of α^{r} at $\rho = 0$ for the given temperature. In the

implementation, calculating the i-th derivative of α^{r} w.r.t. ρ at $\rho = 0$ gives all the (0, 1, ... , $i-1$) intermediate derivatives at the same time.

For instance, for the van der Waals EOS given by $\alpha^{\mathrm{r}} = -\ln(1-b\rho) - a\rho/(RT)$, the virial coeficients are given by:

$$B_{i,\mathrm{vdW}} = \begin{cases} b - a/(RT) & i = 2 \\ b^{i-1} & i > 2 \end{cases} \tag{10}$$

2.2 Mixtures

Mixtures of components follow the same derivatives as for pure fluids for constant composition partial derivatives. In general, mixture models are more complicated (slower) to evaluate, but the code for pure fluids and mixtures overlaps. On the other hand, many thermodynamic properties related to mixture properties invoke composition derivatives of the Helmholtz energy, for instance to evaluate chemical potentials or fugacity coefficients. The formalism of isochoric thermodynamics allows for a concise representation of the derivatives needed to obtain these quantities. Otherwise, there is an enormous explosion of possible derivatives that can be invoked in mixture derivatives because of the new range of possible variables that can be held constant or allowed to vary in the derivative. As a brief and by no means exhaustive summary, two derivatives that appear frequently are

$$\left(\frac{\partial(n\alpha^{\mathrm{r}})}{\partial n_i}\right)_{T,V,n_j}, n\left(\frac{\partial^2(n\alpha^{\mathrm{r}})}{\partial n_j \partial n_i}\right)_{T,V} \tag{11}$$

Further partial derivatives of Eq. (11) may be required depending on the set of independent variables in use:

- Derivative w.r.t. T, with p and composition fixed
- Derivative w.r.t. p, with T and composition fixed
- Derivative w.r.t. a single composition, with all compositions treated as independent variables, and T and V held constant
- Derivative w.r.t. a single composition, with all compositions treated as independent variables, and T and p held constant
- Derivative w.r.t. a single composition, with $N-1$ compositions treated as independent variables, and T and V held constant
- Derivative w.r.t. a single composition, with $N-1$ compositions treated as independent variables, and T and p held constant
- ...

Working out, implementing, and testing all possible permutations of mixture derivatives by hand is not feasible, but can be done in a much more straightforward way in `teqp`. Some variable transformations are required, but they are not as onerous as the more general case. As a very involved example see Bell and Jäger.[7]

3 Isochoric Thermodynamics

3.1 Concept

While thermodynamic states of mixtures are usually specified by molar volume v or pressure p and mole fractions x_i, isochoric thermodynamics uses either amounts of substance, n_i, in a fixed volume V (hence the name) or, if intensive variables are preferred, molar concentrations ρ_i. The conversion between the two concepts is straightforward,

$$\begin{aligned} \rho_i &\equiv \frac{n_i}{V} = \frac{x_i}{v}, \quad \rho \equiv \sum_i^N \rho_i = \frac{1}{v} \\ x_i &\equiv \frac{n_i}{n} = \frac{\rho_i}{\rho}. \end{aligned} \tag{12}$$

In contrast to the x_i, the ρ_i are mutually independent, which greatly simplifies the use of vector algebra.

The primary thermodynamic potential of isochoric thermodynamics is the Helmholtz energy density Ψ,

$$\Psi(\boldsymbol{\rho}, T) \equiv \rho a(\boldsymbol{\rho}, T). \tag{13}$$

The ρ_i are natural variables of Ψ and the chemical potentials μ_i are obtained by

$$\boldsymbol{\mu} = \nabla_{\boldsymbol{\rho}} \Psi \ , \tag{14}$$

where the gradient operator $\nabla_{\boldsymbol{\rho}}$ indicates a differentiation with respect to all concentrations ρ_i at constant temperature.

The pressure is given by

$$p = -\Psi + \boldsymbol{\rho} \cdot \boldsymbol{\mu} = -\Psi + \boldsymbol{\rho} \cdot \nabla_{\boldsymbol{\rho}} \Psi \tag{15}$$

All relevant thermodynamic functions can be obtained from Ψ by differentiation, as is described in Section 8.2. It is advisable, however, to apply numerical differentiation techniques to its residual part only and to handle the ideal-gas part (and eventually chemical contributions) analytically.

3.2 Fluid phase equilibrium—thermodynamic conditions

The necessary and sufficient conditions for equilibrium between two phases denoted by $'$ and $''$ at a given temperature T are the equality of the chemical potentials of all components and pressures,

$$\begin{aligned} \mu_i' &= \mu_i'' , \ i = 1, \ldots, N \\ p' &= p'' \ . \end{aligned} \tag{16}$$

Substituting Eqs. (14) and (15) yields the equations

$$\begin{aligned} \nabla_{\boldsymbol{\rho}} \Psi'' - \nabla_{\boldsymbol{\rho}} \Psi' &= 0 \\ -(\Psi'' - \Psi') + (\boldsymbol{\rho}'' - \boldsymbol{\rho}') \cdot \nabla_{\boldsymbol{\rho}} \Psi' &= 0 \ . \end{aligned} \tag{17}$$

The first one of these equations is a vector equation involving N-component vectors. Hence there are $N + 1$ equations for $2N$ concentrations ρ_i', ρ_i''. One can therefore, for instance, set the mole fractions $\boldsymbol{x}$ and solve the system of equations (17) for ρ' (a scalar)and $\boldsymbol{\rho''}$ (a vector) using a nonlinear root finder.

Eq. (17) is particularly suitable for implementation in computer programs if a programming language is used that accommodates vector arithmetic, e.g., C++.

3.3 Fluid phase equilibria—curve tracing

The conditions of phase equilibrium can be formulated as algebraic equations (as in the previous section) or as differential equations. This is well known for pure compounds; most thermodynamics textbooks mention the Maxwell criterion for the determination of vapor pressures (algebraic equation) as well as the Clausius–Clapeyron equation for vapor pressure curves (differential equation). The corresponding differential equations for mixtures, the so-called Gibbs–Konovalov rules[22,23] (modern formulation, see Section 5.5.1 of Ref. 24), never became as popular, because their application to the computation of phase envelopes is rather complicated.

It is possible, however, to formulate differential equations in the framework of isochoric thermodynamics, which are particularly well suited for machine computation. For example the differential equations for the isothermal vapor–liquid phase envelope of a binary mixture can be expressed as

$$\begin{aligned} \begin{pmatrix} \boldsymbol{H}_1' \cdot \boldsymbol{\rho}'' & \boldsymbol{H}_2' \cdot \boldsymbol{\rho}'' \\ \boldsymbol{H}_1' \cdot \boldsymbol{\rho}' & \boldsymbol{H}_2' \cdot \boldsymbol{\rho}' \end{pmatrix} \frac{\mathrm{d}\boldsymbol{\rho}'}{\mathrm{d}p} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \boldsymbol{H}'' \frac{\mathrm{d}\boldsymbol{\rho}''}{\mathrm{d}p} &= \boldsymbol{H}' \frac{\mathrm{d}\boldsymbol{\rho}'}{\mathrm{d}p} \ , \end{aligned} \tag{18}$$

where the $\boldsymbol{H}_i$ are the row vectors of the Hessian matrix of Ψ,

$$\boldsymbol{H} = \begin{pmatrix} \left(\frac{\partial^2 \Psi}{\partial \rho_1^2}\right) & \left(\frac{\partial^2 \Psi}{\partial \rho_1 \, \partial \rho_2}\right) \\ \left(\frac{\partial^2 \Psi}{\partial \rho_1 \, \partial \rho_2}\right) & \left(\frac{\partial^2 \Psi}{\partial \rho_2^2}\right) \end{pmatrix} \ . \tag{19}$$

Both equations are systems of linear equations, which can be solved for the derivatives $\mathrm{d}\rho_i'$ or $\mathrm{d}\rho_i''$, respectively. Eq. (18) cannot be evaluated directly if one of the concentrations is zero; in this case the continuous extension can be used.[25] Similar differential equations have been derived for isobaric and isoplethic cross-sections of the phase envelope.[26,27] Moreover, all these equations can be extended to multicomponent mixtures, as explained in the original literature.

The phase envelopes are obtained by integrating the differential equations, preferably with

an adaptive integrator (e.g., the Runge–Kutta–Fehlberg[28] or the Cash–Karp[29] method). The integration requires an initial state. This can, for instance, be a pure-fluid vapor-liquid equilibrium state, which can be obtained either by a separate calculation (for which very robust algorithms exist) or from "superancillary equations".[30,31] It should be highlighted that it is not necessary, at any point of the tracing, to invert the EOS (i.e., determine the density for a given pressure). Instead, the pressure is computed from the phase concentrations at the end of the computation.

At this point one might wonder why differential equations are proposed when there is already a set of much simpler algebraic equations (Eq. (17)) for the problem of phase equilibrium calculations. These algebraic equations, however, are nonlinear and can only be numerically solved by iterative methods. These methods need initial values and are prone to fail if these initial values are not good enough. In contrast to the use of the algebraic equations, the differential equations do not require non-linear rootfinding from potentially insufficiently accurate initial guesses. On the other hand, during the integration of differential equations numerical errors can accumulate. The method proposed here combines both approaches: Starting from a pure-fluid state, a new equilibrium state on the phase envelope is obtained by integrating the differential equation. The result is then used as initial value for the solver of the algebraic equations, which eliminates any accumulated errors and thus "polishes" the integration result. As the results of the integration step are usually good to 6 or more decimal places already, the algebraic solver has no convergence problems. In most cases, a single step of a Newton–Raphson rootfinder is sufficient to achieve convergence within machine precision.

3.4 Critical curves—algebraic equations

The vapor–liquid two–phase region of a pure compound ends at a critical point. Because of Gibbs' phase rule, binary mixtures have critical curves. These curves, however, are not the boundaries of vapor-liquid coexistence regions only, but also of liquid–liquid two-phase regions. The patterns of the critical curves can be partitioned into several categories, the so-called phase diagram classes. Understanding these classes and, in particular, knowing the number and locations of the critical curves of a mixture is essential for the correct prediction of phase equilibria. It is easy to overlook a liquid–liquid phase split that one does not know to exist!

This section as well as the next one summarize methods to calculate critical states of mixtures that have been explained in more detail elsewhere.[6]

For a single-phase state of a mixture that is stable against splitting into two phases the Hessian matrix $\boldsymbol{H}$ (see Eq. (19)) is positive-definite. This is equivalent to stating that all eigenvalues of λ_k of $\boldsymbol{H}$ are positive. The boundary of stability is reached when (at least) one eigenvalue becomes zero. If the eigenvalues are in ascending order, $\lambda_1 < \lambda_2 < \cdots$, the first criterion for a critical state, the so-called spinodal criterion, can therefore be expressed as

$$\lambda_1(\boldsymbol{\rho}) = 0 \; . \tag{20}$$

The second criterion reflects the fact that λ_1 has a local minimum in the direction indicated by the eigenvector $\boldsymbol{u}_1$,

$$\frac{\mathrm{d}\lambda_1(\boldsymbol{\rho})}{\mathrm{d}\sigma_1} = 0 \text{ with } \boldsymbol{\rho} = \boldsymbol{\rho}^{\mathrm{c}} + \sigma_1 \boldsymbol{u}_1 \; , \tag{21}$$

where σ_1 is a scalar.

The following presentation of the mathematics of critical states is considerably simplified if the ρ_i coordinates are replaced by transformed coordinate system whose axes are aligned with the eigenvectors $\boldsymbol{u}_k$. The transformation is given by

$$\begin{aligned} \boldsymbol{\rho} &= \boldsymbol{\rho}^{\mathrm{c}} + \sum_{k=1}^{N} \boldsymbol{u}_k \sigma_k = \boldsymbol{\rho}^{\mathrm{c}} + \boldsymbol{U}\boldsymbol{\sigma} \\ \boldsymbol{\sigma} &= \boldsymbol{U}^{\mathrm{T}}(\boldsymbol{\rho} - \boldsymbol{\rho}^{\mathrm{c}}) \; . \end{aligned} \tag{22}$$

Here $\boldsymbol{U}$ denotes a matrix whose row vectors are the eigenvectors $\boldsymbol{u}_k, k = 1, \ldots, N$.

In the transformed coordinate system, the Hessian matrix of Ψ is a diagonal matrix,

$$\boldsymbol{H} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} , \tag{23}$$

and the eigenvalues can be expressed as

$$\lambda_k = \left(\frac{\partial^2 \Psi}{\partial \sigma_k^2} \right) . \tag{24}$$

Then the two critical conditions can be expressed as

$$\begin{aligned} \lambda_1 &= \left(\frac{\partial^2 \Psi}{\partial \sigma_1^2} \right) = 0 \\ \left(\frac{\partial \lambda_1}{\partial \sigma_1} \right)_{\sigma_{j>1}} &= \left(\frac{\partial^3 \Psi}{\partial \sigma_1^3} \right) = 0 . \end{aligned} \tag{25}$$

They constitute two nonlinear algebraic equations, from which the concentrations ρ_1^c and ρ_2^c can be determined for a given temperature. Eq. (25) is valid for pure compounds and multicomponent mixtures.

3.5 Critical curves—differential equations

The system of equations (Eq. (25)) is notorious for its difficult convergence. It is therefore worth-while—in analogy to two-phase envelopes—to develop differential equations for critical curves. The formalism of isochoric thermodynamics yields a rather compact set of equations,[6]

$$\begin{aligned} [\boldsymbol{U}^{\mathrm{T}} (\nabla_{\boldsymbol{\sigma}} \lambda_1)] \left. \frac{\mathrm{d}\boldsymbol{\rho}}{\mathrm{d}T} \right|_{\mathrm{c}} &= -\frac{\partial'}{\partial T} \left(\frac{\partial^2 \Psi}{\partial \sigma_1^2} \right) \\ [\boldsymbol{U}^{\mathrm{T}} (\nabla'_{\boldsymbol{\sigma}} ((\nabla_{\boldsymbol{\sigma}} \lambda_1) \cdot \boldsymbol{u}_1))] \left. \frac{\mathrm{d}\boldsymbol{\rho}}{\mathrm{d}T} \right|_{\mathrm{c}} &= -\frac{\partial'}{\partial T} \left(\frac{\partial^3 \Psi}{\partial \sigma_1^3} \right) \end{aligned} \tag{26}$$

Together, these two equations constitute a system of linear equations, which can be solved for the concentrations derivatives along the critical curve. The gradients of λ_1 are

$$\begin{aligned} \nabla_{\boldsymbol{\sigma}} \lambda_1 &= \begin{pmatrix} \frac{\partial'}{\partial \sigma_1} \left(\frac{\partial^2 \Psi}{\partial \sigma_1^2} \right) \\ \frac{\partial'}{\partial \sigma_2} \left(\frac{\partial^2 \Psi}{\partial \sigma_1^2} \right) \end{pmatrix} \\ \nabla'_{\boldsymbol{\sigma}} ((\nabla_{\boldsymbol{\sigma}} \lambda_1) \cdot \boldsymbol{u}_1) &= \begin{pmatrix} \frac{\partial'}{\partial \sigma_1} \left(\frac{\partial^3 \Psi}{\partial \sigma_1^3} \right) \\ \frac{\partial'}{\partial \sigma_2} \left(\frac{\partial^3 \Psi}{\partial \sigma_1^3} \right) \end{pmatrix} . \end{aligned} \tag{27}$$

The primes on the outer differentiation operators indicate that here the eigenvectors change during the differentiation. Such differentiations can easily be carried out with a finite-step method. The numerical inaccuracy introduced is eliminated when the integration step is followed by a "polishing step", i.e., when the result of the integrator is used as initial value for a nonlinear rootfinder that solves Eq. (25).

As for the calculation of phase envelopes, some special measures must be taken if one of the concentrations becomes zero. But this is not a difficult obstacle, because the limiting values of all terms involved can be computed accurately (the reader is referred to the original publication[6]). It is therefore possible to start the calculation of a critical curve at one of the pure-fluid critical points.

4 Differentiation

As has been seen, derivatives of the Helmholtz energy expressions are essential to calculating thermodynamic properties from EOS. This differentiation can be done analytically or using numerical methods.

4.1 Finite differentiation

The classical numerical analysis texts propose finite differentiation to take numerical derivatives of real-valued functions. For instance, the centered first finite difference of truncation order two is given by

$$f' \approx \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \tag{28}$$

This approach results in a number of downsides: 1) Not easy to know what step size h to

take. Heuristics like $h \approx \sqrt{\varepsilon}x$, with ε the epsilon of the numerical type, are acceptable, but not ideal (*What about at $x = 0$ in our example?*) 2) Even when the optimal step size has been selected, the result still deviates from the exact solution. Figure 1 demonstrates this for the function $\cos(x)\sin(x)$.

The deficiencies of finite differentiation can be avoided in many cases, as we will show in this work. An alternative (and not recommended because it incurs a severe computational speed penalty[31]) approach to improve the accuracy of finite differentiation is to use arithmetic with more digits of precision, for instance with the `boost::multiprecision` library in C++, or `mpmath` in Python.

4.2 Into the complex plane

Years from now, it is possible that the blog post of Higham[32] will be seen as a landmark in the application of complex mathematics to the practical differentiation of real functions. While the ideas in the blog post are not new, they are explained in a very accessible way. To summarize: if a function normally takes real arguments, but you can instead provide complex arguments, that will allow you to get the derivatives of the function output with respect to the input argument to numerical precision. Mathematically, this is

$$f' \approx \frac{\Im(f(x + hj))}{h} \tag{29}$$

where $\Im(z)$ gets the imaginary component of a complex number z and j is the imaginary unit ($\sqrt{-1}$). An example in Python demonstrates the approach; the derivative of the function $\cos(x)\sin(x)$ is equal to $\cos(2x)$:

```
import cmath
# Value and step
x = 8.3
h = 1e-100
# Function
f = lambda x: cmath.cos(x)*cmath.sin(x)
# Exact derivative
fprime_exact = cmath.cos(2.0*x).real
# Complex step derivative
fprime_csd = f(x+1j*h).imag/h
err = fprime_csd - fprime_exact
print(f'Error: {err:18.16e}' )
""" Output:
Error: 0.0000000000000000e+00
"""
```

There are two remarkable features of this approach: 1) the step size `h` is 10^{-100} (not a typo) 2) the error is zero (also not a typo). The difference compared with the evaluation of $\cos(2x)$ in infinite precision is not quite zero, but the error rounds to zero in the example because the difference used to obtain `err` is carried out in double precision. Therefore, with complex step derivatives it is possible to calculate numerical derivatives to all of the digits of double precision arithmetic (with some manageable caveats)! The downsides of the complex step derivative approach are twofold: a) it only allows for first derivatives and b) it is not extremely computationally efficient (as compared with hand-written derivatives), although it is still very computationally efficient (a few times slower than the evaluation of the function itself, in general; still much slower than automatic differentiation).

The generalization of complex step derivatives to higher-order derivatives is the multicomplex approach; multicomplex numbers are generalizations of complex numbers. In analogy to complex numbers, which comprise two real numbers,

$$z = r_0 + jr_1 \ , \tag{30}$$

a multicomplex number of level l comprises two multicomplex numbers of level $l - 1$,

$$m^{(l)} = m_0^{(l-1)} + j^{(l)} m_1^{(l-1)} \ , \tag{31}$$

where $j^{(l)}$ represents the imaginary unit of the l-th level. A multicomplex number of level 1 is equivalent to a complex number, and therefore $j^{(1)} \equiv j$. A multicomplex number of level l

comprises 2^l real components. For example, a multicomplex number of level 2 (also called a bicomplex number) is given by

$$\begin{aligned} m^{(2)} &= m_0^{(1)} + j^{(2)} m_1^{(1)} \\ &= r_{00} + j^{(1)} r_{01} + j^{(2)} r_{10} + j^{(1)} j^{(2)} r_{11} \ , \end{aligned} \tag{32}$$

where the r_{ij} are real numbers. It can be shown that a derivative of arbitrary order of a function $f(x)$ can be computed as

$$\frac{\mathrm{d}^n f(x)}{\mathrm{d}x^n} \approx \frac{1}{h^n} \mathfrak{C}_{2^n-1} \left\{ f(x + h j^{(1)} + \ldots h j^{(n)}) \right\} \ , \tag{33}$$

where $\mathfrak{C}_{2^n-1}\{\}$ denotes an operator that returns the last, "most imaginary" component of a multicomplex number. The exact definition of this operator is given in another publication.[33] As for the complex step method, the increment h can be made very small, so that the derivative is obtained with almost machine precision.

The multicomplex step method can also be applied to functions of more than one variable, and then can also compute mixed derivatives. Moreover, the multicomplex step method yields not only the desired derivative, but all derivatives of lower order, too.[33] Deiters and Bell[33] also discuss alternative differentiation techniques, for instance techniques based on Cauchy's integral theorem. Hyperduals are a notionally similar construct to multicomplex algebra that allow for exact second partial derivatives like for complex step derivatives[34,35]

Contemporary compiler implementations do not contain multicomplex algebra (yet), but there are multicomplex program libraries available for Python and C++[33] which make it easy to implement multicomplex differentiation in existing programs.

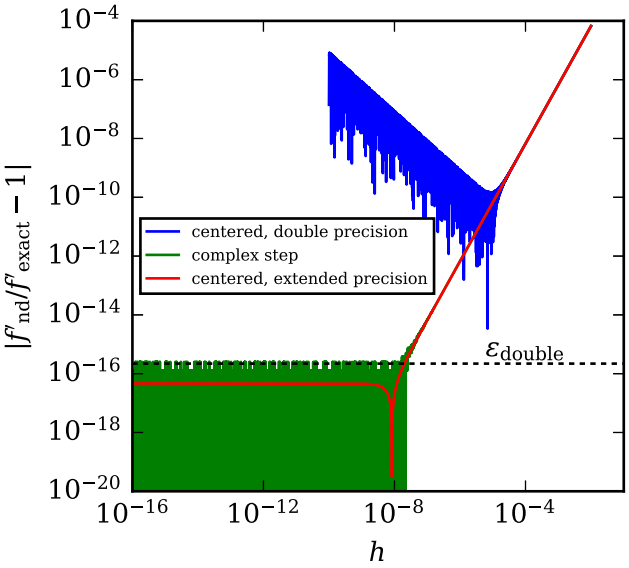


Figure 1: Absolute value of relative error in derivatives of $\cos(x)\sin(x)$ at $x = 8.1$ with finite differentiation in double precision via Eq. (28), complex step derivatives via Eq. (29), and finite differentiation with 100 digits of working precision, as a function of step size h. The subscript nd is for the numerical derivative, and exact is for the exact solution. The indicated value of $\varepsilon_{\mathrm{double}}$ is the machine precision in double precision arithmetic.

4.3 Automatic differentiation

Automatic differentiation is yet another computational technique that allows us to compute function derivatives.[36] For a review of automatic differentiation techniques and its misconceptions, we refer to Güneş Baydin et al..[37] It is a powerful and simpler alternative to numerical and analytical derivatives that offers performance, precision, and convenience when taking derivatives of functions. It is in general a superior approach to finite differences, which often lacks accuracy due to amplified round-off errors.

There are many variations of automatic differentiation techniques. Usually they can be classified in two groups: source code generation/transformation and operator overloading. Several computer codes exists in which all or some of these techniques are implemented. They include, for example, TAPENADE,[38] Stan Math Library,[39] CppAD,[40] CasADi,[41] ADOL-C,[42] Clad,[43] Adept,[44] `autodiff`.[45] We

focus below on the description of automatic differentiation techniques in which operator overloading is used and the derivatives are computed in a forward-mode approach instead of reverse-mode.

In a forward-mode automatic differentiation algorithm, the usual floating point type `double` in C/C++ is replaced by a special type (commonly named `dual`) that implements, via operator and function overloading, all necessary arithmetic and mathematical operations on that type. Thus, variables of this `dual` type can be, like variables of `double` type, added, subtracted, multiplied, divided, and used in mathematical functions such as sin, cos, log, exp, etc. However, this type has also been implemented to propagate derivatives using the chain rule of calculus, which yields accurate results up to machine precision. The output of an automatic differentiation evaluation is a numerical result, not a mathematical expression. We list below an example of a program using the `autodiff`[45] C++ library that illustrates what we explain here, on a higher level and more practical description, since the chain rule operations are hidden away from the user.

```cpp
#include <autodiff/forward/dual.hpp>
#include <iostream>
using namespace autodiff;
using namespace std;

dual f(dual x, dual y){
    return sin(x)*cos(y);
}

int main(){
    dual x = 0.2;
    dual y = 0.3;

    dual u; // used as u = f(x, y)

    seed(x);
    u = f(x, y);
    unseed(x);

    cout << "du/dx = " << u.grad << std::endl;

    seed(y);
    u = f(x, y);
    unseed(y);

    cout << "du/dy = " << u.grad << std::endl;

    return 0;
}

// Output:
// du/dx =  0.936293363584199234 1
// du/dy = -0.058710801693826517 0
```

The above program implements a function named `f` using type `dual` for its arguments and return value instead of `double` and shows how its derivatives with respect to the arguments are obtained. The `seed` operation indicates which variable we should compute derivatives with respect to (we do this for both `x` and `y` individually). By doing this, the chain rule operations that happen in the background during the function evaluation `f(x, y)` propagates the derivatives of all sub-expressions with respect to the seeded variable.

An equivalent, higher-level and simplified program can be obtained in `autodiff` as shown next:

```cpp
#include <autodiff/forward/dual.hpp>
#include <iostream>
using namespace autodiff;
using namespace std;

dual f(dual x, dual y){
    return sin(x)*cos(y);
}

int main(){
    dual x = 0.2;
    dual y = 0.3;

    double dudx = derivative(f, wrt(x), at(x, y));
    double dudy = derivative(f, wrt(y), at(x, y));

    cout << "du/dx = " << dudx << std::endl;
    cout << "du/dy = " << dudy << std::endl;

    return 0;
}

// Output:
// du/dx =  0.93629336358419923414
// du/dy = -0.058710801693826517O3
```

`autodiff` also offers many other convenience methods for derivative computations such as:

- `autodiff::gradient`,
- `autodiff::jacobian`,
- `autodiff::hessian`,
- `autodiff::taylorseries`.

`autodiff`[45] uses advanced template metaprogramming techniques to optimize, at compile-time, expression evaluations to greatly reduce the overhead of employing a numeric type other than native floating point types (e.g., `float`, `double`).

We show in the next section the usage of `autodiff` for computation of thermodynamic derivatives that are in general derived by hand and implemented manually, resulting in thermodynamic computer codes that are both complicated and prone to errors that are difficult to identify when the EOS itself is complicated.

Note: `autodiff` is also used in Reaktoro[46] for the computation of derivatives of thermodynamic properties with respect to temperature, pressure and composition for the sake of computing exact Jacobian matrices in its chemical equilibrium and kinetics solvers as detailed in Ref. 47. For example, when using Newton's method to solve chemical equilibrium, every iteration requires derivatives such as $(\partial \boldsymbol{\mu}/\partial \boldsymbol{n})_{T,p}$, $(\partial \boldsymbol{\mu}/\partial T)_{p,\boldsymbol{n}}$, $(\partial \boldsymbol{\mu}/\partial p)_{T,\boldsymbol{n}}$, where $\boldsymbol{\mu}$ and $\boldsymbol{n}$ are the vector of chemical potentials and amounts of all species across all phases in the multiphase chemical system. These chemical potentials are computed from equations of state and activity models for each phase in the system. While $(\partial \boldsymbol{\mu}/\partial \boldsymbol{n})_{T,p}$ is always needed, because the vector $\boldsymbol{n}$ is always unknown in every chemical equilibrium problem, $(\partial \boldsymbol{\mu}/\partial T)_{p,\boldsymbol{n}}$ and/or $(\partial \boldsymbol{\mu}/\partial p)_{T,\boldsymbol{n}}$ are only evaluated when T and/or p are also unknown in the problem (e.g., chemical equilibrium with specified internal energy and volume). `autodiff` has been a reliable library to compute these derivatives with machine precision and performance, and is used in `teqp` as the default differentiation approach.

We highlight, however, that the goal in this work is to use higher-order derivatives, rather than just first-order derivatives for Jacobian computations, to fully resolve all thermodynamic properties that can be extracted from the residual Helmholtz energy function.

5 Implementation

Having laid out the mathematics and thermodynamics, we now pivot to the computational aspects. In this section we describe the manner in which the library is practically implemented.

In high level languages without strong typing (e.g., Python), writing generic models in the form $\alpha^{\mathrm{r}}(T, \rho, x)$ and taking derivatives with the numerical tools we describe above requires very little additional code. Here for instance is a complete example of calculating $\partial^2 p/\partial v \partial T$ for the van der Waals EOS with the multicomplex package for Python:

```python
import multicomplex

# Definition of model
a = 0.13617565223879216
b = 3.2204437295436385e-05
R = 8.31446261815324 # J/mol/K
def p(T, v):
    return R*T/(v-b) - a/v**2

# State point
T = 300 # K
v = 1.2 # m^3/mol

# Second cross derivative
d2pdTdv = multicomplex.diff_mcxN(
    lambda Tv: p(Tv[0],Tv[1]),
    x=[T, v],
    orders=[1, 1]
)
print(d2pdTdv, -R/(v-b)**2) # approx, exact
"""output:
-5.7742422965987155 -5.7742422965987155
"""
```

So, why not develop the entire library in Python? In short, speed. While developing in Python would indeed be very convenient, it would be too slow to be used in any sort of production code. Besides, if you have existing code written in Fortran (for instance), interfacing with Python is a non-starter, but interfacing with a shared library with a C interface is viable. The goal of this library is to be computationally efficient enough that it could replace existing optimized Fortran (REFPROP) and C++ (CoolProp) implementations. As we'll see, it comes quite close to this goal. As a result, the decision was made from the outset to develop in C++.

5.1 C++ Implementation

This section is intended for readers already well versed in C++, and explains some of the low-level details. Readers may want to proceed to the next section to return to the thermodynamics, but before doing so, might want to quickly peruse the examples.

The generic function for the residual reduced Helmholtz energy to be implemented in `teqp` has the specification

```
template<typename T1,
         typename T2,
         typename T3>
auto alphar(const T1& T,
            const T2& rho,
            const T3& molefrac) {
    ...
}
```

for which a few notes are needed:

- the implementation is a pure function with no side-effects and all arguments are immutable (are `const`) and passed by reference
- no specification of the input and output types is provided; the method must be able to handle all different permutations of numerical types. The vector of mole fractions is also implemented in a generic fashion. Even though there are no formal constraints on the types provided to the function, they must adhere to the expected interface, i.e. implement arithmetic operators `+`, `-`, `*`, `/`, `exp`, and so on.
- because the function is generic, all functions called by this function must also be, and so too any datatypes that store any of the intermediate calculations, all the way to the bottom of the call stack.

It is a common paradigm in C++ to define an abstract base class (ABC) and implementation classes deriving from this base class, and store instances of derived classes in smart containers (e.g., `std::vector<std::shared_ptr<ABC>>`, with `ABC` being the base class). Unfortunately, once templated arguments are used in any virtual function, the polymorphism (deriving from ABC) approach is no longer possible. For instance this definition of the base class would NOT be allowed in C++:

```
class ABC{
    template<typename T1,
             typename T2,
             typename T3>
    virtual auto alphar(
      const T1& T,
      const T2& rho,
      const T3& molefrac) = 0;
};
```

Thus, if the specification defines that the model has a templated method `alphar`, an alternative approach is needed to store instances of the models in a dynamic container (as is needed for the C interface). Thus, smart pointers (`std::shared_ptr`, `std::unique_ptr`, etc.) cannot be used. Instead, each model instance may be specified in a `std::variant`, and the variants stored in a dynamic container (like a `std::vector` or a `std::unordered_map`). The list of possible model implementations allowed in the `std::variant` must be enumerated at compile time, and in this case is hardcoded. One implementation downside of this approach is that the `std::variant` does not allow for direct access to the class instance it contains, rather the compiler needs to use a visitor model to determine which model is contained in the variant, and as such, cannot simply return the model it holds without already knowing what is to be returned. This is why the templated function `std::get<T>()` in the C++ standard library needs to know the template type `T` at compile time. The cost to be paid here is only in terms of lines of code; the compiler nearly completely optimizes away the visitor model functions.

In practice, while this all sounds complicated, and there are indeed a lot of important and frustrating details to become familiar with, the implementation of a given model is still nothing more than filling in the function `alphar`. Other details are the responsibility of the library developer. For simple models, implementing the function in C++ is trivial; for more involved models, more effort is required. As an example, to implement the cubic EOS (Peng-Robinson and SRK) in their canonical form took the primary developer a few hours after already learning how to avoid common pitfalls in model implementation in `teqp`, and having more than a decade of daily use of C++. That stands in marked contrast to the several months spent working out the hand-written derivatives.[3]

The complete implementation of the van der Waals EOS model for a pure fluid could read

```
#pragma once
#include "teqp/types.hpp"

class vdWEOS1 {
private:
double a, b;

public:
vdWEOS1(double a, double b) : a(a), b(b) {};

// Exact value for R, given by k_B*N_A
const double R = 1.380649e-23*6.02214076e23;

template<typename T1, typename T2, typename T3>
auto alphar(const T1& T,
            const T2& rho,
            const T3& molefrac) const {
  auto Psiminus = -log(1.0 - b*rho);
  auto val = Psiminus - a/(R*T)*rho;
  return forceeval(val);
}
};
```

Note how no header other than the `types.hpp` header (used only to provide the `forceeval` function) is needed. There is no inheritance, and the model implementation need not implement a plethora of functions to satisfy the requirements of the ABC (as is common in conventional polymorphism).

A comment about `forceeval`: When `autodiff` is used to evaluate expressions, unevaluated expressions are obtained from a statement like `a+b` when `a` and/or `b` are autodiff duals. These unevaluated expressions store lightweight references to local memory locations of the variables `a` and `b`. It is invalid behavior to return these expressions from the function because the memory references become dangling. The call to `forceeval` flattens the expression (resolving all the references) before the expression falls out of scope and is returned. Therefore, as is the case for Eigen expressions, it is necessary to be careful about the use of `auto`. The multicomplex implementation does not have this limitation (the `forceeval` function has no effect), so confirming that the same results are returned with multicomplex and automatic differentiation can serve as a powerful check in the debugging workflow.

One limitation of `teqp` is the rather slow compilation time; this is a consequence of the generality of the models. The compiler must work out template instantiations of each combination of possible arguments to `alphar`. As an intro-

duction to the problem, if the only thing that you would like to do with the model is to calculate α^r and its partial derivative with respect to density with complex step derivatives, that invokes the templates `alphar(double, double, Eigen::ArrayXd)` and `alphar(double, std::complex<double>, Eigen::ArrayXd)`, and recursively invokes all templated functions down the call stack. As more models and higher derivatives enter into the compilation process, the number of template instantiations increases and compilation speed decreases. Models like the multi-fluid model with nested templated classes are especially computationally costly to compile. If the C-language interface is used in C++, with the JSON interface for model construction, the compilation cost can be amortized, although a small runtime cost is incurred, and limited functionality is exposed.

The steps to add a new model are as follows:

1. A class is defined that exposes the `alphar` function and any other methods needed to implement the model
2. An entry is added in the JSON-builder function in `include/teqp/json_builder.hpp` (optional)
3. A wrapper of the class is added in the Python interface in the `interface` directory (optional)

5.2 Python wrapper

Although modern C++ is much more user friendly than its predecessors, it remains the case that operations like file input/output and plotting are more convenient in a high-level language like Python. Thus, a C++ $\leftrightarrow$ Python interface was written with the pybind11 library.[48] This Python interface allows for the use of the models in a close to one-to-one mapping with the low-level C++ code while not incurring much computational overhead. The C++ details are hidden from the user, and the time of compiling the library must only be paid once. Metadata interchange, where needed, is in the JSON format. The binary wheels of `teqp` are available in the Python package index (pypi).

The Python wrapper allows for use of this library in modern languages that support calling Python, including MATLAB, Julia, R, Scilab, etc.

5.3 C wrapper

C++ does not interface conveniently with legacy tools like Fortran or Microsoft Excel. In order to broaden the range of end users, a C interface was written that accepts a JSON representation of the model parameters as inputs, as null-terminated strings. All inputs and outputs are C data types: `int`, `char*`, `double`, `double*`, etc. The caller must be very careful to ensure that memory is managed properly on the calling side, as memory checking safeguards are not possible inside the C functions. This interface allows for calling from any language supporting calling shared libraries. The models are managed with a `std::variant` as described above. Unique keys allow the caller to know which model is being operated upon. When no error is produced, evaluation of the model is approximately as fast as the full C++ interface (example: 0.9 μs/call with PC-SAFT for Λ_{01}^r). Examples demonstrating how to call the shared library from Python and C++ are in the source file `interface/C`.

6 Results and Examples

6.1 Models

As described in the abstract, a number of types of models are implemented in `teqp`, sorted roughly in terms of likely accuracy:

- van der Waals: One of the simplest thermodynamic models, primarily useful for testing purposes. Implementations for pure fluids and mixtures (with standard mixing rules).
- canonical cubics (Peng-Robinson and SRK): As described in Ref. 3, the canonical cubics are benchmark models still used heavily in industry, in spite of their weaknesses.

- PC-SAFT: The original version of PC-SAFT as published by Gross and Sadowski,[49] *without* any association contribution. An important error from their paper is fixed [1], as noted in the `teqp` source code.

- CPA: Cubic-plus-association. Allows for a variety of association models, and base thermodynamic models. Currently only implemented for pure fluids because mixtures require iteration for the association part, while pure fluids are non-iterative.

- multi-fluid: This is the model used in NIST REFPROP and CoolProp. It is precisely the same corresponding states formulation used in the development of the GERG-2004 model.[1,50] It is based upon highly accurate multiparameter pure fluid EOS in concert with departure functions and reducing functions to account for mixing behaviors.

6.2 Speed

The more derivatives desired, the more computational cost will be incurred. This holds true both for hand-written derivatives and algorithmic differentiation. The question then is how competitive these algorithmic differentiation speeds are with the optimized computational codes.

Each of the timing tests in this section are carried out in a docker container running ubuntu 20.04 running on a Windows host machine. The compilers used are `gfortran` for REFPROP 10.0, and `clang++` for `teqp` to yield a fair comparison. Note that the docker containers must be run in privileged mode to yield the same speedup on all host machines.

Figure 2 shows the timing for propane, for the multi-fluid model and other models implemented in `teqp` for the function $\Lambda^{\mathrm{r}}_{0n}$ (defined in Eq. (2)). The timing baseline is REFPROP and all calculations are done in C++ to eliminate any additional overhead. 10^6 calls are done, and the average of the five fastest repeats of the 10^6 calls are kept as the time for a particular data point. For the multi-fluid model, the same models and reducing parameters are used in both `teqp` and REFPROP, and the same numerical results are obtained in both libraries to all 16 digits. The other models (vdW, PR, PC-SAFT) are shown for comparison. The timing for REFPROP 10.0[18] of the multi-fluid model is on the order of 0.5 μs/call. REFPROP calculates all of the derivatives up to the third order in temperature and density with all cross terms; this represents unnecessary work when only first derivatives are needed, but is worthwhile when higher derivatives are required. The PC-SAFT model, even when optimized, is much slower than the multiparameter model. As a test of overhead, the van der Waals and Peng-Robinson implementations show that very simple models are very fast to evaluate (but are far less accurate than the multiparameter models). The timing in `teqp` is competitive with REFPROP, 1% slower for $\Lambda^{\mathrm{r}}_{02}$, and 38% faster for evaluation of α^{r} for the same model formulation. As described above, many of the most important thermodynamic properties (see Section 8.1) require only first derivatives, but more advanced calculations require additional derivatives.

[1] Eqn. A.11 should use the reciprocal of the right-hand-side

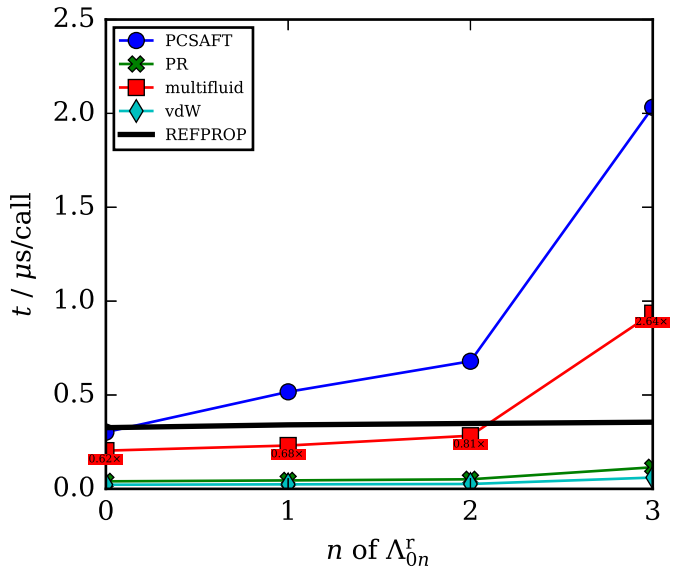


Figure 2: Timing of the $\Lambda^{\mathrm{r}}_{0n}$ function for models for propane with `autodiff` differentiation. The multiparameter (and REFPROP function `PHIXdll`) results are for the EOS of Lemmon et al..[51] The labels on each multifluid data point are the ratio relative to the REFPROP timing.

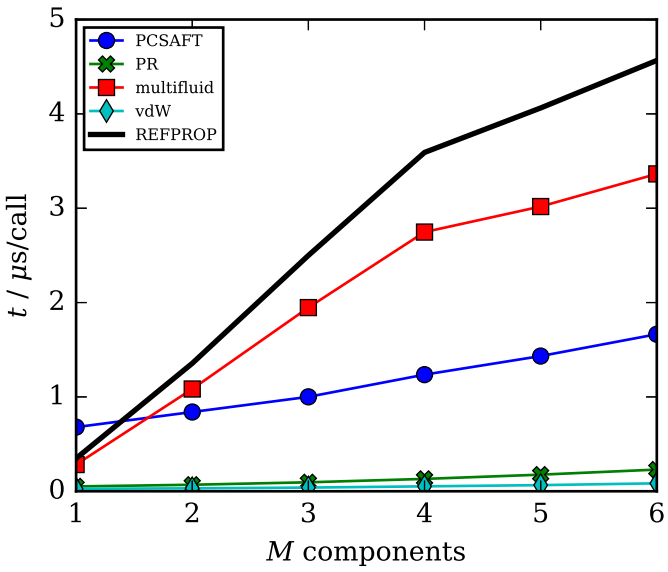


Figure 3: Timing of the $\Lambda^{\mathrm{r}}_{02}$ function for mixtures with M alkanes for models from `teqp` and REFPROP.

Once additional components are included in a mixture, the computational cost of algorithmic differentiation increases. To demonstrate this, mixtures with the first M n-alkanes are modeled, and the $\Lambda^{\mathrm{r}}_{02}$ derivative is calculated for each of the mixtures, as presented in Fig. 3. As before, the timing comparison is against that of REFPROP with the multi-fluid model. Across the number of components considered, `teqp` is always slightly faster than REFPROP, although the computational penalty is modest, and the scaling is qualitatively similar.

Many common mixture calculations require composition derivatives. For instance, the chemical potential of the i-th component is obtained from $\mu_i = (\partial\Psi/\partial\rho_i)_{T,\rho_{j\neq i}}$, which is the molar concentration gradient of Ψ with the temperature fixed. The residual portion of the chemical potential (the most important part for chemical equilibrium calculations) is given by $\mu^{\mathrm{r}}_i = (\partial\Psi^{\mathrm{r}}/\partial\rho_i)_{T,\rho_{j\neq i}}$. The `gradient` method of `autodiff` allows for a straightforward evaluation of the gradient. A wrapper function (in C++, a lambda function) is written for the implementation of `Psir` and then passed to `autodiff::gradient`. REFPROP does not have an identical function for the residual portion of the chemical potential, but it does allow for calculation of the fugacity coefficient, a closely related quantity, obtained from the `FUGCOFdll` function. Calculation of the fugacity coefficient involves two parts: the gradient of Ψ^{r} and $\Lambda^{\mathrm{r}}_{01}$ (see Eq. (59)). Figure 4 shows the results of the timing. An initially surprising result, the fugacity coefficient calculations from `teqp` are approximately two times faster than those in REFPROP.

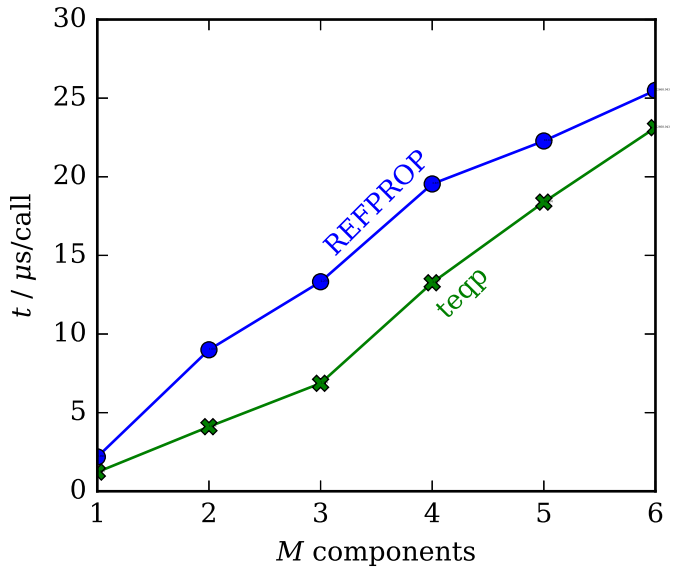


Figure 4: Timing of the fugacity coefficient function with the multi-fluid model from `teqp` and REFPROP.

In summary, in a fair apples-to-apples comparison, `teqp` is faster than REFPROP for calculations of fugacity coefficients and up to first derivatives of α^{r}.

6.3 Accuracy

Assessing the accuracy of the calculated derivatives is not straightforward. The ground truth for the comparison is the derivative carried out in infinite precision arithmetic. Infinite precision is impossible to achieve in practice, so instead calculations (and derivatives) carried out with a very large number of digits of precision are the stand-in for the ground truth. In this context, accuracy does not mean the difference between the model and experiments, rather between the double precision result and the quasi-infinite-precision result. The comparisons are further complicated by the fact that some of the conditions we are testing in this section represent extremely small numbers, and therefore results are sensitive to precise details of the implementations, even to the level of the order in which terms are added together.

As a first test of the `teqp` approach, the values of the compressibility factor Z were calculated at the saturated liquid and vapor densities for propane from the EoS of Lemmon et al..[51] The saturated liquid and vapor densities were obtained from REFPROP's saturation routine (which uses double precision arithmetic throughout), and for each phase, the compressibility factor Z was obtained from the given densities. For orientation, Fig. 5 shows the temperature and density values, showing that at temperatures near that of the triple point, the ratio of liquid to vapor density is greater than 10^{10}! Numerical peril awaits those who venture into this region of thermodynamic space.

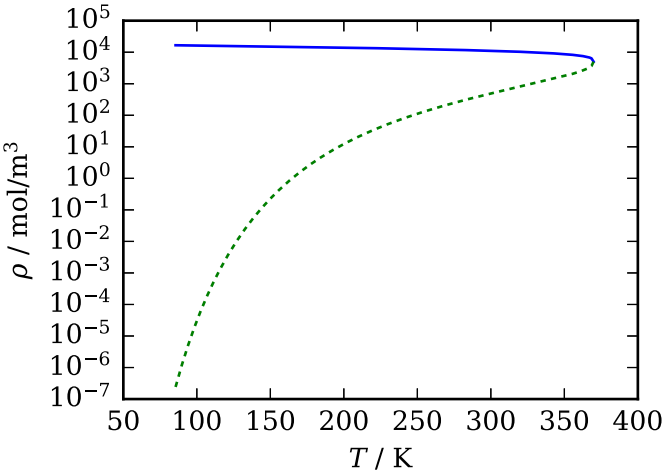


Figure 5: Density of the saturated liquid (solid curve) and vapor phases (dashed curve) as calculated by REFPROP.

After a phase equilibrium calculation the pressures should be identical in both phases, but they will not be because of the loss in numerical precision. The `teqp` results were calculated with autodiff differentiation. The "exact" values were taken with 200 digits of working precision with a standalone implementation given in the source file `src/sat_Z_accuracy.cpp`. Calculations of the derivatives were carried out with centered finite differences of sixth truncation order[52] with the numerical type provided by `boost::multiprecision`. This combination required the least reworking of `teqp`. We may take these extended precision derivatives as the ground truth for the derivatives, and comparisons may be made against them. While finite differentiation is not recommended in double precision, it is acceptable in extended precision due to the larger amount of working precision

available (see for instance Fig. 1). The results are plotted in Fig. 6. In the vapor phase the deviations from the exact value are so small that most round to zero in double precision ($\log(0) = -\infty$) and are not visible in the scale of the plot. The large densities in the liquid phase result in severe truncation and significant errors in Z. REFPROP and `teqp` are in excellent agreement with each other, and suffer the same loss in precision at low temperature (near the maximum density of the EOS). This analysis is in agreement with similar issues identified in Ref. 31 when developing superancillary equations.

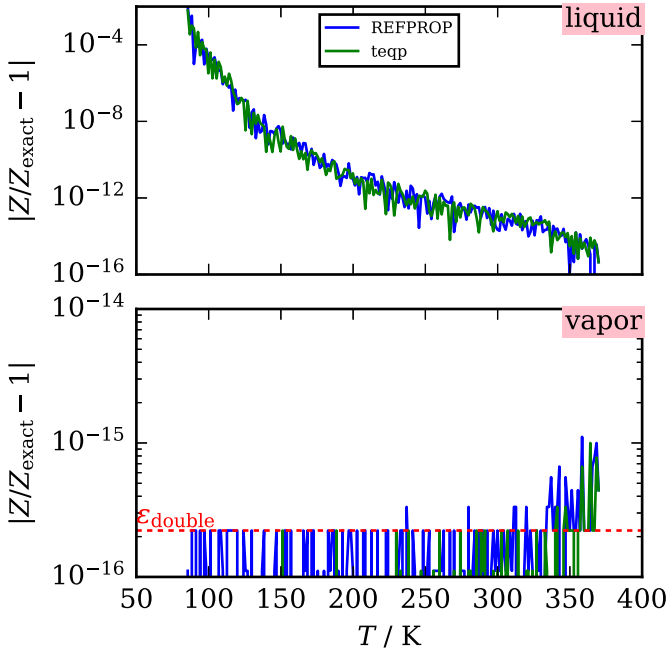


Figure 6: Relative deviations between values of $Z = p/(\rho RT)$ calculated by `teqp` and REFPROP versus those calculated by an extended precision calculation in `teqp` for propane.[51] The indicated value of $\varepsilon_{\text{double}}$ is the machine precision in double precision arithmetic.

The deviations for Λ_{01}^{r} are shown in Fig. 7. The results for Λ_{01}^{r} show a much closer agreement with the exact solution than do Z (reminder: $Z = 1 + \Lambda_{01}^{\text{r}}$), which is somewhat surprising since they are related by only the addition of 1. The deviations for Λ_{01}^{r} are mostly less than a part in 10^{14}. *So how can we reconcile the tiny deviations for Λ_{01}^{r} with the proportionally larger deviations for Z?* The explanation can be made mathematically. Suppose we have a quantity Y that has some exact value Y_{exact} and we know that the actual value is within some error band of ϵ (in a relative sense). Therefore if we consider the new quantity $Y+1$, the relative deviation in $Y + 1$ would be

$$\Delta = \frac{Y_{\text{exact}}(1+\epsilon)+1}{Y_{\text{exact}}+1} - 1 \tag{34}$$

which factors to

$$\Delta = \frac{\epsilon}{1 + \dfrac{1}{Y_{\text{exact}}}} \tag{35}$$

and if Y_{exact} is approximately $-1+10^{-9}$, even if ϵ is on the order of 10^{-15}, the deviations in $Y+1$ are still on the order of 10^{-4}. This example demonstrates how even if the results for Λ_{01}^{r} are small, the deviations for other properties may not be. This limitation is one of mathematics in general, not of `teqp` in particular.

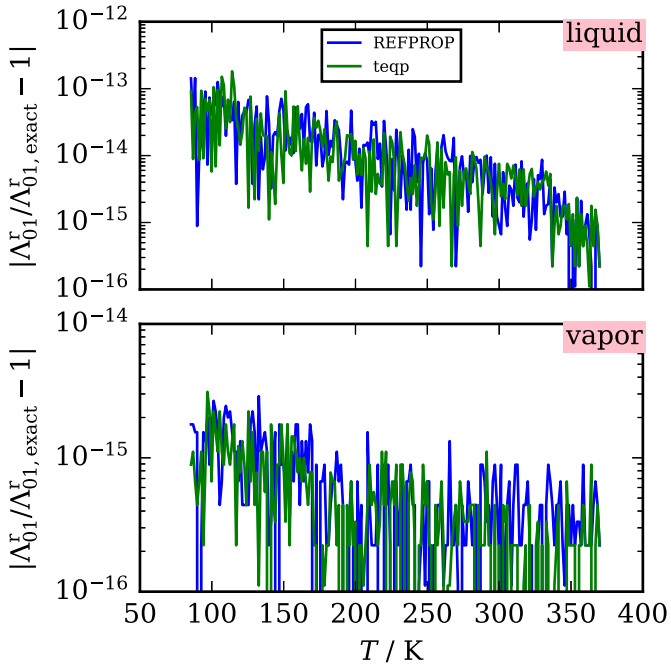


Figure 7: Relative deviations between values of Λ_{01}^{r} calculated by `teqp` and REFPROP versus those calculated by an extended precision calculation in `teqp` for propane.[51]

The second density derivative Λ_{02}^{r} represents an intermediate case, as shown in Fig. 8. The results from REFPROP are still mostly in

agreement with the exact solution to within approximately numerical precision. This is good to see because the calculated thermodynamic properties (see the equations in Section 8.1) invoke no derivatives higher than $\Lambda^{\mathrm{r}}_{02}$. On the other hand, the derivatives from `teqp` are starting to deviate significantly in the vapor phase at extremely low densities, especially so at low temperature. Admittedly we are pushing double precision arithmetic to its breaking point; subtle implementation details in the EOS (and in `autodiff`) might be responsible. In fact, the tests of $\Lambda^{\mathrm{r}}_{02}$ in this section ultimately identified a deficiency in CoolProp version 6.4.2 caused by numerical precision lost by intermediate rounding for very small density values. Simply rearranging the terms eliminated the problem, highlighting the utility of checking derivatives with extended precision calculations, even if the extended precision calculations are not used in production code.

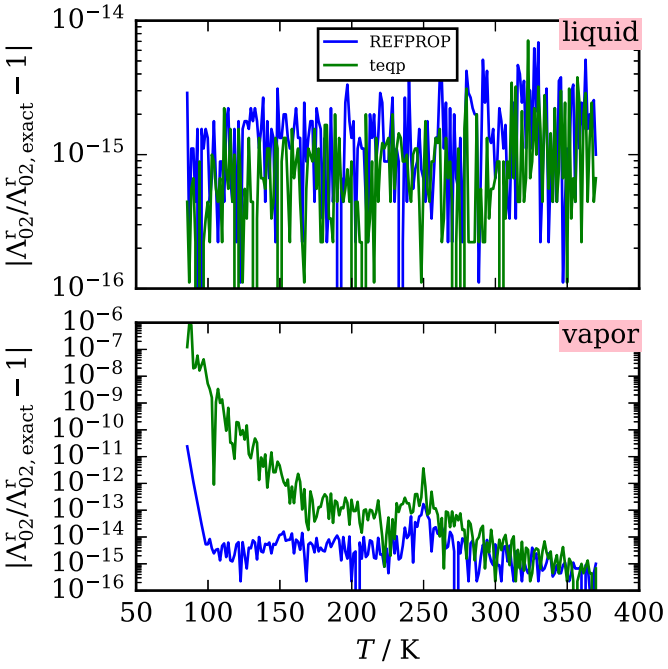


Figure 8: Relative deviations between values of $\Lambda^{\mathrm{r}}_{02}$ calculated by `teqp` and REFPROP versus those calculated by an extended precision calculation in `teqp` for propane.[51]

REFPROP implements up to $\Lambda^{\mathrm{r}}_{03}$ so it is possible to check the obtained values for the third density derivative in the same manner as for $\Lambda^{\mathrm{r}}_{01}$. Figure 9 shows the results of this comparison. For the third derivative, points along the saturated liquid phase again show values in excellent agreement with the exact solution. On the other hand, the saturated vapor points correspond to proportionately enormous errors, especially at low temperatures (corresponding to minuscule densities). It is well known that taking numerical derivatives is a destructive operation from the standpoint of numerical precision. These results also help to explain why moving to higher-order rootfinding methods like Halley's method (or more generally the higher Householder methods[53]) for density rootfinding in the gas phase (as was attempted in CoolProp[2]) is not necessarily an improvement on the naïve Newton's method for rootfinding if the derivatives themselves deviate significantly from their exact solution. CoolProp implements up to the fourth derivatives, which appears to be potentially problematic in the gas phase because the derivative error increases quickly with the derivative order.

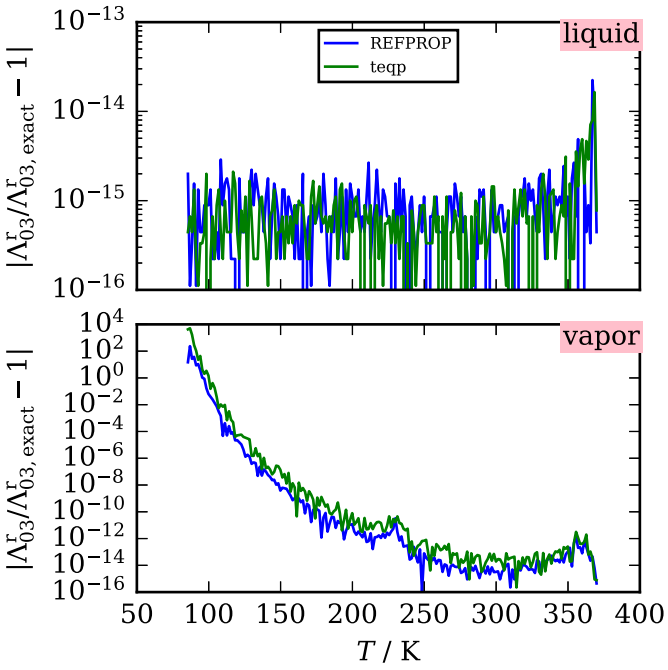


Figure 9: Relative deviations between values of $\Lambda^{\mathrm{r}}_{03}$ calculated by `teqp` and REFPROP versus those calculated by an extended precision calculation in `teqp` for propane.[51]

Although these results might seem to call into question the approach proposed in this work for metrology applications and standards work, it

is not obvious that the distinctions in error between `teqp` and REFPROP are to be expected. Further forensic study of the precision lost by intermediate rounding in `teqp` would be worthwhile. The key point is that for reasonable state points, down to perhaps a density of 1000 times lower than that at the critical point, the loss in precision is mostly within a part in 10^{10} for derivatives up to $\Lambda^{\mathrm{r}}_{03}$, even for the vapor phase.

6.4 Testing, Verification, and Documentation

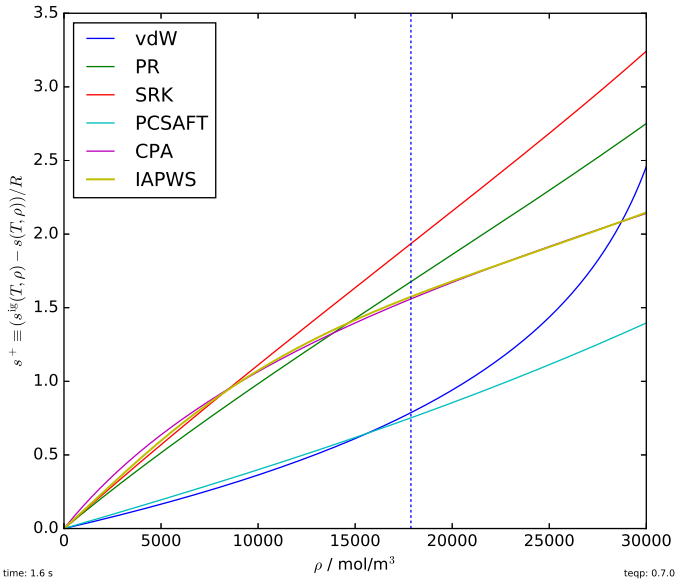
As `teqp` is a blank-sheet development effort, software development best practices were adopted from the outset. The code is under source control, the code is heavily tested, docker containers are used to assist with reproducibility of results, and continuous integration builds are launched at every code commit. A comprehensive suite of tests have been written with the `Catch2` testing library, and where feasible, a test-driven development style has been adopted; tests have been written in concert with new features. In some cases, multiple derivative methods are compared against each other (multicomplex, complex step, autodiff). Code coverage metrics from the tests are good, with tests covering more than 85% of the lines of code, and more than 94% of the core (a docker container for testing and code coverage is in `dev/docker/gcov`). Aside from the dependencies of the library (e.g., boost, Eigen, and the C++ standard library), the core of `teqp` is concisely written in less than 700 lines of header-only code (the model implementations are approximately another 1000 lines of code). All tests pass on the Microsoft and clang++ C++ compilers. Furthermore, to ensure no inappropriate memory access occurs, the test suite has been run through the valgrind memory checking tool (a docker configuration is in `dev/docker/valgrind`), and no errors were found. Documentation of the C++ code was generated with the doxygen library.

6.5 Gallery

In this section we provide some demonstrations of the capabilities of this library in a literate programming style; the graphical result is presented along with the Python code used to generate it. Inspiration has been taken from the gallery of the `matplotlib` plotting library. Comments have been removed from the source code for concision.

The recent discussions of entropy scaling applied to transport properties of associating and polar fluids[54,55] have raised questions about how well our common EOS are able to reproduce the residual entropy. As an exploration of this question and a demonstration of the models available in `teqp`, Fig. 10 shows calculations of the residual entropy. The goal of this first example is primarily to demonstrate how to instantiate the models. Under the hood, s^+ is calculated from evaluation in the form $s^+ = \Lambda^{\mathrm{r}}_{00} - \Lambda^{\mathrm{r}}_{01}$. The PC-SAFT implementation in `teqp` does not include the association contribution (because the association contribution invokes iterative calculations for the association fractions), so the PC-SAFT results should be considered as only a demonstration of how to use PC-SAFT in `teqp`. If we consider the IAPWS model for water (the scientific formulation from Wagner and Pruß[56]) to be a reliable baseline for comparison of residual entropy values, the cubic plus association (CPA) is shown to yield the best predictions. The canonical cubic EOS (vdW, PR, SRK) yield qualitatively incorrect predictions. The key point to highlight in this example is that no derivatives are explicitly implemented; all are handled by numerical differentiation.

As a tracing example we trace vapor-liquid-equilibrium (VLE) isotherms of a binary mixture with the isochoric thermodynamics described above in Fig. 11. The EOS is Peng-Robinson with $k_{ij} = 0$ and parametric tracing[57] is used. The pure-fluid saturation densities are obtained from the superancillary curves[30] available in `teqp`. Tracings initiated at each of the two pure fluids are included, and they overlay almost perfectly, even without polishing the phase equilibrium solution. A key point

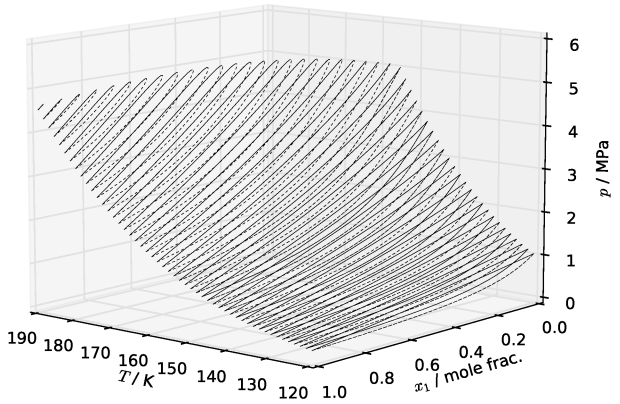



```
import timeit, numpy as np
import matplotlib.pyplot as plt
plt.style.use('classic')
import teqp

def build_models():
    Tc_K, pc_Pa, acentric = 647.096, 22064000.0, 0.3442920843

    water = {
        "a0i / Pa m^6/mol^2": 0.12277 , "bi / m^3/mol": 0.000014515, "c1": 0.67359,
        "Tc / K": 647.096, "epsABi / J/mol": 16655.0, "betaABi": 0.0692, "class": "4C"
    }
    j = {"cubic": "SRK", "pures": [water], "R_gas / J/mol/K": 8.3144598}

    datapath = teqp.get_datapath()
    def get_PCSAFT():
        c = teqp.SAFTCoeffs()
        # Values from https://doi.org/10.1016/j.fluid.2017.11.015,
        # but association contribution is ignored
        c.name = 'Water'
        c.m = 2.5472
        c.sigma_Angstrom = 2.1054
        c.epsilon_over_k = 138.63
        return teqp.PCSAFTEOS(coeffs=[c])

    return [
        ('vdW', teqp.vdWEOS([Tc_K], [pc_Pa])),
        ('PR', teqp.canonical_PR([Tc_K], [pc_Pa], [acentric])),
        ('SRK', teqp.canonical_SRK([Tc_K], [pc_Pa], [acentric])),
        ('PCSAFT', get_PCSAFT()),
        ('CPA', teqp.CPAfactory(j)),
        ('IAPWS', teqp.build_multifluid_model(["Water"], datapath))
    ]

fig, ax = plt.subplots(1,1,figsize=(7,6))
T = 700 # K
rhovec = np.geomspace(0.1, 30e3, 10000) # mol/m^3; critical density is 17873.8... mol/m^3
tic = timeit.default_timer()
for abbrv, model in build_models():
    splus = np.array([teqp.get_splus(model, T, [rho]) for rho in rhovec])
    plt.plot(rhovec, splus, label=abbrv, lw = 1.5 if abbrv=='IAPWS' else 1)
elap = timeit.default_timer()-tic
plt.axvline(17873.8, dashes=[2,2])
plt.legend(loc='best')
plt.gca().set(xlabel=r'$\rho$ / mol/m$^3$', ylabel=r'$s^+\equiv (s^{\rm ig}(T,\rho)-s(T,\rho)
plt.tight_layout(pad=0.2)
plt.gcf().text(0,0,f'time: {elap:0.1f} s', ha='left', va='bottom', fontsize=7)
plt.gcf().text(1,0,f'teqp: {teqp.__version__}', ha='right', va='bottom', fontsize=7)
plt.savefig('splus_water_700K.pdf')
plt.close()
```

Figure 10: Calculations of s^+ for water at a supercritical temperature of 700 K

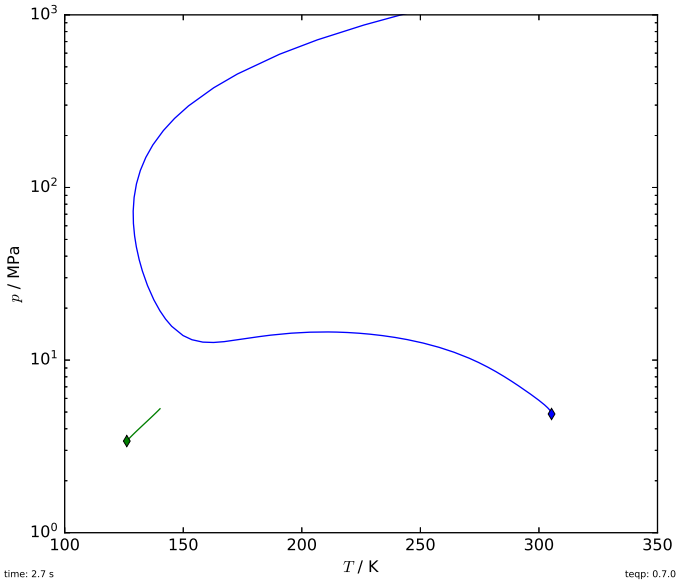
of this figure is to demonstrate how reliable the method is; it gracefully stops at the critical locus for supercritical isotherms and there are no failures of any tracings. Admittedly the Peng-Robinson EOS is not an extreme test because it is generally well behaved (in contrast to the critical region of some of the more accurate EOS[7]). But nevertheless, the tracing works well, and is computationally efficient for this example. The average time per trace is on the order of 40 milliseconds, including the plotting, JSON conversion, construction of pandas DataFrame, and so on. For subcritical isotherms, the tracing takes about 5 milliseconds.

Next, we trace the critical loci originating from each of the pure species for the binary mixture of nitrogen + ethane in Fig. 12. The algorithm is the parametric tracing one of Deiters and Bell,[6] with simple Euler integration. A trace, as described above, is initiated at each pure component indicated by a diamond marker. The tracer follows the critical locus until one of the termination conditions is met. The adaptive Runge-Kutta integrator is also available for the integration, but not needed (or recommended) in this case. The reader is invited to refer to Fig. 6 from Ref. 7 for a comparison with the tracing approach.




```
time: 3.2 s                                                                    teqp: 0.7.0
import json, timeit
import pandas, numpy as np, matplotlib.pyplot as plt
plt.style.use('classic')
import teqp

Tc_K = [190.564, 154.581]
pc_Pa = [4599200, 5042800]
acentric = [0.011, 0.022]
model = teqp.canonical_PR(Tc_K, pc_Pa, acentric)
fig, ax = plt.subplots(1,1,figsize=(7, 6), subplot_kw=dict(projection='3d'))
tic = timeit.default_timer()
for ifluid in [0,1]:
    model0 = teqp.canonical_PR([Tc_K[ifluid]], [pc_Pa[ifluid]], [acentric[ifluid]])
    for T in np.linspace(190, 120, 50):
        if T > Tc_K[ifluid]: continue
        [rhoL, rhoV] = model0.superanc_rhoLV(T)
        rhovecL = np.array([0.0, 0.0]); rhovecL[ifluid] = rhoL
        rhovecV = np.array([0.0, 0.0]); rhovecV[ifluid] = rhoV
        opt = teqp.TVLEOptions(); opt.calc_criticality = True
        df = pandas.DataFrame(teqp.trace_VLE_isotherm_binary(model, T, rhovecL, rhovecV, opt)
        df['too_critical'] = df.apply(
            lambda row: (abs(row['crit. conditions L'][0]) < 5e-8), axis=1)
        first_too_critical = np.argmax(df['too_critical'])
        df = df.iloc[0:(first_too_critical if first_too_critical else len(df))]
        line, = ax.plot(xs=df['T / K'], ys=df['xL_0 / mole frac.'], zs=df['pL / Pa']/1e6,
            lw=0.2, color='k')
        ax.plot(xs=df['T / K'], ys=df['xV_0 / mole frac.'], zs=df['pL / Pa']/1e6,
            dashes=[2,2], color=line.get_color(), lw=0.2)
elap = timeit.default_timer()-tic
ax.view_init(elev=10., azim=130)
ax.set(xlabel='$T$ / K', ylabel='$x_1$ / mole frac.', zlabel='$p$ / MPa')
fig.text(0,0,f'time: {elap:0.1f} s', ha='left', va='bottom', fontsize=7)
fig.text(1,0,f'teqp: {teqp.__version__}', ha='right', va='bottom', fontsize=7)
plt.tight_layout(pad=0.2)
plt.savefig('PR_VLE_trace.pdf')
plt.show()
```

Figure 11: Isothermal vapor-liquid equilibria curves traced from the pure fluids for methane + ethane




```
import timeit
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('classic')
import pandas
import teqp

def get_critical_curve(ipure):
    """ Return curve as pandas DataFrame """
    names = ['Nitrogen', 'Ethane']
    model = teqp.build_multifluid_model(names, teqp.get_datapath())
    T0 = model.get_Tcvec()[ipure]
    rho0 = np.array([1.0/model.get_vcvec()[ipure]]*2)
    rho0[1-ipure] = 0
    o = teqp.TCABOptions()
    o.init_dt = 1.0 # step in the parameter
    o.rel_err = 1e-8
    o.abs_err = 1e-5
    o.integration_order = 5
    o.calc_stability = True
    o.polish = True
    curveJSON = teqp.trace_critical_arclength_binary(model, T0, rho0, '', o)
    df = pandas.DataFrame(curveJSON)
    rhotot = df['rho0 / mol/m^3']+df['rho1 / mol/m^3']
    df['z0 / mole frac.'] = df['rho0 / mol/m^3']/rhotot
    return df

if __name__ == '__main__':
    fig, ax = plt.subplots(1,1,figsize=(7, 6))
    tic = timeit.default_timer()
    for ipure in [1,0]:
        df = get_critical_curve(ipure)
        first_unstable = np.argmax(~df['locally stable'])
        df = df.iloc[0:(first_unstable if first_unstable else len(df))]
        line, = plt.plot(df['T / K'], df['p / Pa']/1e6, '-')
        plt.plot(df['T / K'].iloc[0], df['p / Pa'].iloc[0]/1e6, 'd',
            color=line.get_color())

    elap = timeit.default_timer()-tic
    plt.gca().set(xlabel='$T$ / K', ylabel='$p$ / MPa',
        xlim=(100, 350), ylim=(1, 1e3))
    plt.yscale('log')
    plt.tight_layout(pad=0.2)
    plt.gcf().text(0,0,f'time: {elap:0.1f} s', ha='left', va='bottom', fontsize=7)
    plt.gcf().text(1,0,f'teqp: {teqp.__version__}', ha='right', va='bottom', fontsize=7)
    plt.savefig('N2_ethane_critical.pdf')
    plt.close()
```

Figure 12: Critical curves traced from the pure fluids for nitrogen + ethane

7 Caveats and Limitations

While this library is already in production use,[8] two important limitations should be highlighted:

- The ideal-gas part of the Helmholtz energy is not included in `teqp`. This is because there is historically a diverse set of terms that have been used, and including all of them is difficult to do in a generic manner. $\alpha^{(\mathrm{ig})}$ (and its derivatives) may be defined in the same manner as α^{r} and can be included in the `teqp` architecture without much additional complication.
- Phase equilibria calculations and iterative calculations are essential but are not included. An example of an iterative calculation is to calculate density given the temperature, pressure, and composition. Iterative and phase equilibrium calculations are error-prone (especially for mixtures), and require excellent starting values to stand a chance of converging to the correct solution. Methods are available[58] to make some parts of that calculation extremely reliable, but the generic density rootfinding problem is non-trivial. Phase equilibria for mixtures are even more challenging, especially so for mixtures that are non-Type I according to the classification of van Konynenburg and Scott.[59]

Conclusion

The results in this work demonstrate that the use of numerical differentiation can allow for the development of very flexible thermodynamic property libraries quite closely reproducing the reference implementation in REFPROP. The computational speed penalty is modest at worst, and faster than the reference implementation in some cases. A few lines of code are all that is required to implement a new thermodynamic model. The derivative algorithms are agnostic as to what model is being used, which will allow for a new generation of very modular thermodynamic calculation libraries. The selection of implemented algorithms give a flavor of what is possible in further development efforts.

The code of `teqp` is included in the NIST data repository `https://doi.org/10.18434/mds2-2483` and the working repository of the code is at `https://github.com/usnistgov/teqp`. The release used for this paper corresponds to the version indicated in the figures in the gallery.

The vision is that the scope of `teqp` will include as many equations of state as the community would like to implement. It is planned to develop (but probably not within `teqp`) new algorithms and phase equilibrium routines taking advantage of innovations in superancillary equations[30,31] and one-dimensional rootfinding.[58] This combination (along with the isochoric tracing approaches described above), should make for very reliable thermodynamic calculations that are also computationally efficient.

Acknowledgments

The authors thank: Tobias Loew for laying the groundwork for the object model used in the C++ implementation, Pierre Walker and Jeffrey Young for detailed reviews and discussion, Andreas Jaeger and David Zhu for help with implementing CPA.

8 Appendix

8.1 Conventional derivatives

Mirroring Table 3.9 from Span,[17] we explicitly use molar density ρ, and R is the molar universal gas constant.[60] The nomenclature uses the derivative term defined in Eq. (2)
Pressure ($p \equiv -(\partial a/\partial v)_T$):

$$\frac{p}{\rho RT} = 1 + \Lambda^{\mathrm{r}}_{01} \tag{36}$$

Internal energy ($u = a + Ts$):

$$\frac{u}{RT} = \Lambda^{\mathrm{tot}}_{10} \tag{37}$$

Enthalpy ($h = u + p/\rho$):

$$\frac{h}{RT} = 1 + \Lambda_{01}^{\mathrm{r}} + \Lambda_{10}^{\mathrm{tot}} \tag{38}$$

Entropy ($s \equiv -(\partial a/\partial T)_v$):

$$\frac{s}{R} = \Lambda_{10}^{\mathrm{tot}} - \Lambda_{00}^{\mathrm{tot}} \tag{39}$$

Gibbs energy ($g = h - Ts$):

$$\frac{g}{RT} = 1 + \Lambda_{01}^{\mathrm{r}} + \Lambda_{00}^{\mathrm{tot}} \tag{40}$$

Derivatives of pressure:

$$\left(\frac{\partial p}{\partial \rho}\right)_T = RT\left(1 + 2\Lambda_{01}^{\mathrm{r}} + \Lambda_{02}^{\mathrm{r}}\right) \tag{41}$$

$$\left(\frac{\partial p}{\partial T}\right)_\rho = R\rho\left(1 + \Lambda_{01}^{\mathrm{r}} - \Lambda_{11}^{\mathrm{r}}\right) \tag{42}$$

Isochoric specific heat ($c_v \equiv (\partial u/\partial T)_v$):

$$\frac{c_v}{R} = -\Lambda_{20}^{\mathrm{tot}} \tag{43}$$

Isobaric specific heat ($c_p \equiv (\partial h/\partial T)_p$; see Eq. 3.56 from Span[17] for the derivation):

$$\frac{c_p}{R} = -\Lambda_{20}^{\mathrm{tot}} + \frac{\left(1 + \Lambda_{01}^{\mathrm{r}} - \Lambda_{11}^{\mathrm{r}}\right)^2}{1 + 2\Lambda_{01}^{\mathrm{r}} + \Lambda_{02}^{\mathrm{r}}} \tag{44}$$

Speed of Sound

$$w^2 \equiv -\frac{v^2}{M}\left(\frac{\partial p}{\partial v}\right)_s = \frac{1}{M}\left(\frac{\partial p}{\partial \rho}\right)_s = \frac{c_p}{M c_v}\left(\frac{\partial p}{\partial \rho}\right)_T \tag{45}$$

where M is the molar mass and ρ and v are specific.

8.2 Isochoric derivatives (Pure fluid)

Helmholtz energy:

$$a = \frac{\Psi}{\rho} \tag{46}$$

Pressure:

$$p = \left(\frac{\partial \Psi}{\partial \rho}\right)_T \rho - \Psi \tag{47}$$

Internal energy:

$$u = \frac{1}{\rho}\left(\Psi - T\left(\frac{\partial \Psi}{\partial T}\right)_\rho\right) = -\frac{T^2}{\rho}\left(\frac{\partial (\Psi/T)}{\partial T}\right)_\rho \tag{48}$$

Enthalpy:

$$h = \left(\frac{\partial \Psi}{\partial \rho}\right)_T - \frac{T}{\rho}\left(\frac{\partial \Psi}{\partial T}\right)_\rho \tag{49}$$

Entropy:

$$s = -\frac{1}{\rho}\left(\frac{\partial \Psi}{\partial T}\right)_\rho \tag{50}$$

Chemical potential (equal to Gibbs energy for pure substance):

$$\mu = \left(\frac{\partial \Psi}{\partial \rho}\right)_T \tag{51}$$

Isothermal rigidity:

$$\left(\frac{\partial p}{\partial \rho}\right)_T = \left(\frac{\partial^2 \Psi}{\partial \rho^2}\right)_T \rho \tag{52}$$

Isochoric tension:

$$\beta_V \equiv \left(\frac{\partial p}{\partial T}\right)_\rho = \left(\frac{\partial^2 \Psi}{\partial T \partial \rho}\right)\rho - \left(\frac{\partial \Psi}{\partial T}\right)_\rho \tag{53}$$

Isothermal compressibility:

$$\kappa_T \equiv -\frac{1}{V_m}\left(\frac{\partial V_m}{\partial p}\right)_T = \frac{1}{\rho}\left(\frac{\partial \rho}{\partial p}\right)_T = \frac{1}{\left(\frac{\partial^2 \Psi}{\partial \rho^2}\right)_T \rho^2} \tag{54}$$

Isobaric expansivity:

$$\alpha_p \equiv \frac{1}{V_m}\left(\frac{\partial V_m}{\partial T}\right)_p = \frac{\left(\frac{\partial \Psi}{\partial T}\right)_\rho - \left(\frac{\partial^2 \Psi}{\partial T \partial \rho}\right)\rho}{\left(\frac{\partial^2 \Psi}{\partial \rho^2}\right)_T} \tag{55}$$

Isochoric heat capacity:

$$c_v = -\frac{T}{\rho}\left(\frac{\partial^2 \Psi}{\partial T^2}\right)_\rho \tag{56}$$

Isobaric heat capacity:

$$c_p = c_v + \frac{T}{\rho}\kappa_T \beta_V^2 \tag{57}$$

8.3 Mixture derivatives

Residual part of chemical potential (Eq. 7.69 of GERG-2004[50])

$$\mu_i^{\mathrm{r}} = RT\left(\frac{\partial(n\alpha^{\mathrm{r}})}{\partial n_i}\right)_{T,V,n_j} = \left(\frac{\partial \Psi^{\mathrm{r}}}{\partial \rho_i}\right)_{T,\rho_j} \tag{58}$$

Fugacity coefficient (Eq. 7.27 of GERG-2004[50])

$$\ln(\varphi_i) = \left(\frac{\partial(n\alpha^{\mathrm{r}})}{\partial n_i}\right)_{T,V,n_j} - \ln Z \tag{59}$$

$$= \frac{1}{RT}\left(\frac{\partial \Psi^{\mathrm{r}}}{\partial \rho_i}\right)_{T,\rho_j} - \ln\left(1 + \Lambda_{01}^{\mathrm{r}}\right) \tag{60}$$

Fugacity (Eq. 5.42 of GERG-2004[50])

$$f_i = \rho_i RT \exp\left(\left(\frac{\partial(n\alpha^{\mathrm{r}})}{\partial n_i}\right)_{T,V,n_j}\right) \tag{61}$$

Literature Cited

(1) Kunz, O.; Klimeck, R.; Wagner, W.; Jaeschke, M. *The GERG-2004 Wide-Range Equation of State for Natural Gases and Other Mixtures*; VDI Verlag GmbH, 2007.

(2) Bell, I. H.; Wronski, J.; Quoilin, S.; Lemort, V. Pure and Pseudo-pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp. *Ind. Eng. Chem. Res.* **2014**, *53*, 2498–2508, DOI: 10.1021/ie4033999.

(3) Bell, I. H.; Jäger, A. Helmholtz Energy Transformations of Common Cubic Equations of State for Use with Pure Fluids and Mixtures. *J. Res. Nat. Inst. Stand. Technol.* **2016**, *121*, 238, DOI: 10.6028/jres.121.011.

(4) Hammer, M.; Aasen, A.; Wilhelmsen, Ø. ThermoPack. online, 2022 (accessed March 23, 2022); https://github.com/SINTEF/thermopack.

(5) Wilhelmsen, Ø.; Aasen, A.; Skaugen, G.; Aursand, P.; Austegard, A.; Aursand, E.; Gjennestad, M. A.; Lund, H.; Linga, G.; Hammer, M. Thermodynamic Modeling with Equations of State: Present Challenges with Established Methods. *Ind. Eng. Chem. Res.* **2017**, *56*, 3503–3515, DOI: 10.1021/acs.iecr.7b00317.

(6) Deiters, U. K.; Bell, I. H. Calculation of Critical Curves of Fluid Mixtures through Solution of Differential Equations. *Ind. Eng. Chem. Res.* **2020**, *59*, 19062–19076, DOI: 10.1021/acs.iecr.0c03667.

(7) Bell, I. H.; Jäger, A. Calculation of critical points from Helmholtz-energy-explicit mixture models. *Fluid Phase Equilib.* **2017**, *433*, 159–173, DOI: 10.1016/j.fluid.2016.10.030.

(8) Bell, I. H.; Riccardi, D.; Bazyleva, A.; McLinden, M. O. Survey of Data and Models for Refrigerant Mixtures Containing Halogenated Olefins. *J. Chem. Eng. Data.* **2021**, *66*, 2335–2354, DOI: 10.1021/acs.jced.1c00192.

(9) Deiters, U. K. A Modular Program System for the Calculation of Thermodynamic Properties of Fluids. *Chem. Eng. Technol.* **2000**, *23*, 581–584, DOI: 10.1002/1521-4125(200007)23:7<581::aid-ceat581>3.0.co;2-p.

(10) Walker, P. J.; Yew, H.-W.; Riedemann, A. Clapeyron.jl: An extensible, open-source fluid-thermodynamics toolkit. *Ind Eng. Chem. Res.* **2022, submitted**,

(11) Rehner, P.; Bauer, G. Application of Generalized (Hyper-) Dual Numbers in Equation of State Modeling. *Frontiers in Chemical Engineering* **2021**, *3*, DOI: 10.3389/fceng.2021.758090.

(12) van der Waals, J. D. Over de Continuiteit van den Gas- en Vloeistoftoestand. Ph.D. thesis, University of Leiden, 1873.

(13) Outcalt, S. L.; McLinden, M. O. A Modified Benedict-Webb-Rubin Equation of State for the Thermodynamic Properties of R152a (1,1-difluoroethane). *J. Phys. Chem. Ref. Data* **1996**, *25*, 605–636, DOI: 10.1063/1.555979.

(14) IAPWS, Revised Release on the IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam, revision 7. 2012.

(15) Wagner, W.; Cooper, J. R.; Dittmann, A.; Kijima, J.; Kretzschmar, H.-J.; Kruse, A.; Mareš, R.; Oguchi, K.; Sato, H.; Stöcker, I.; Šifner, O.; Takaishi, Y.; Tanishita, I.; Trübenbach, J.; Willkommen, T. The IAPWS Industrial Formulation 1997 for the Thermodynamic Properties of Water and Steam. *J. Eng. Gas Turbines Power* **2000**, *122*, 150–184, DOI: 10.1115/1.483186.

(16) Dieterici, C. Ueber den kritischen Zustand. *Ann Phys Chem (Wiedemanns Ann)* **1899**, *69*, 685–705.

(17) Span, R. *Multiparameter Equations of State - An Accurate Source of Thermodynamic Property Data*; Springer-Verlag, Berlin, 2000.

(18) Lemmon, E. W.; Bell, I. H.; Huber, M. L.; McLinden, M. O. NIST Standard Reference Database 23: Reference Fluid Thermodynamic and Transport Properties-REFPROP, Version 10.0, National Institute of Standards and Technology. http://www.nist.gov/srd/nist23.cfm, 2018.

(19) Span, R.; Beckmüller, R.; Hielscher, S.; Jäger, A.; Mickoleit, E.; Neumann, T.; Pohl, S.; Semrau, B.; Thol, M. TREND. Thermodynamic Reference and Engineering Data 5.0. 2020.

(20) Lemmon, E. W.; Jacobsen, R. T.; Penoncello, S. G.; Friend, D. G. Thermodynamic Properties of Air and Mixtures of Nitrogen, Argon, and Oxygen from 60 to 2000 K at Pressures to 2000 MPa. *J. Phys. Chem. Ref. Data* **2000**, *29*, 331–385, DOI: 10.1063/1.1285884.

(21) Jäger, A.; Breitkopf, C.; Richter, M. The Representation of Cross Second Virial Coefficients by Multifluid Mixture Models and Other Equations of State. *Ind. Eng. Chem. Res.* **2021**, *60*, 9286–9295, DOI: 10.1021/acs.iecr.1c01186.

(22) Konowalow, D. Über die Dampfspannungen der Flüssigkeitsgemische. *Ann. Phys.* **1881**, *250*, 34–52; 219–226, Volume also cited as Wied. Ann. 14.

(23) Gibbs, J. W. In *The Scientific Papers of J. Willard Gibbs*; van Name, R. G., Bumstead, H. A., Eds.; Dover Publications, New York, 1961; Vol. 1: Thermodynamics; reprint of a book of 1906 containing articles originally published in 1876–1878.

(24) Deiters, U. K.; Kraska, T. *High-Pressure Fluid Phase Equilibria—Phenomenology and Computation*; Supercritical Fluid Science and Technology (E. Kiran, ed.); Elsevier, Amsterdam, 2012; Vol. 2.

(25) Deiters, U. K. Differential equations for the calculation of fluid phase equilibria. *Fluid Phase Equilib.* **2016**, *428*, 164–173, DOI: 10.1016/j.fluid.2016.04.014.

(26) Deiters, U. K. Differential equations for the calculation of isopleths of multicomponent fluid mixtures. *Fluid Phase Equilib.* **2017**, *447*, 72–83, DOI: 10.1016/j.fluid.2017.03.022.

(27) Bell, I. H.; Deiters, U. K. On the construction of binary mixture p-x and T-x diagrams from isochoric thermodynamics. *AIChE J.* **2018**, *64*, 2745–2757, DOI: 10.1002/aic.16074.

(28) Runge–Kutta–Fehlberg method (RK45). 2022 (accessed September 12, 2020); https://en.wikipedia.org/wiki/Runge\OT1\textendashKutta\OT1\textendashFehlberg_method.

(29) Cash, J. R.; Karp, A. H. A variable-order Runge–Kutta method for initial-value problems with rapidly varying right-hand sides. *ACM Trans. Math. Software* **1990**, *16*, 201–222, DOI: 10.1145/79505.79507.

(30) Bell, I. H.; Deiters, U. K. Superancillary Equations for Cubic Equations of State. *Ind. Eng. Chem. Res.* **2021**, *60*, 9983–9991, DOI: 10.1021/acs.iecr.1c00847.

(31) Bell, I. H.; Alpert, B. K. Efficient and Precise Representation of Pure Fluid Phase Equilibria with Chebyshev Expansions. *Int. J. Thermophys.* **2021**, *42*, 75, DOI: 10.1007/s10765-021-02824-x.

(32) Higham, N. Differentiation With(out) a Difference. 2018 (accessed June 1, 2018); https://sinews.siam.org/Details-Page/differentiation-without-a-difference.

(33) Deiters, U. K.; Bell, I. H. Precise numerical differentiation of thermodynamic functions with multicomplex variables. *J. Res. NIST* **2021**, *126*, 126033, DOI: https://doi.org/10.6028/jres.126.033.

(34) Fike, J.; Alonso, J. The Development of Hyper-Dual Numbers for Exact Second-Derivative Calculations. 49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition. 2011; DOI: 10.2514/6.2011-886.

(35) Fike, J. A. Multi-objective Optimization Using Hyper-dual Numbers. Ph.D. thesis, Stanford University, 2013.

(36) Rall, L. B. *Automatic differentiation: Techniques and applications (Lecture Notes in Computer Science)*, 1st ed.; Springer, Berlin, Germany, 1981; p 166.

(37) Güneş Baydin, A.; Pearlmutter, B. A.; Andreyevich Radul, A.; Mark Siskind, J. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.* **2018**, *18*, 1–43.

(38) Hascoet, L.; Pascual, V. The Tapenade automatic differentiation tool. *ACM Transactions on Mathematical Software* **2013**, *39*, 1–43, DOI: 10.1145/2450153.2450158.

(39) Carpenter, B.; Hoffman, M. D.; Brubaker, M.; Lee, D. D.; Li, P.; Betancourt, M. The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *CoRR* **2015**, *abs/1509.07164*, DOI: 10.48550/arXiv.1509.07164.

(40) Bell, B. CppAD: a package for C++ algorithmic differentiation. 2021 (accessed Oct 29, 2021); http://www.coin-or.org/CppAD.

(41) Andersson, J. A. E.; Gillis, J.; Horn, G.; Rawlings, J. B.; Diehl, M. CasADi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation* **2019**, *11*, 1–36, DOI: 10.1007/s12532-018-0139-4.

(42) Griewank, A.; Juedes, D.; Utke, J. Algorithm 755: ADOL-C. *ACM Transactions on Mathematical Software* **1996**, *22*, 131–167, DOI: 10.1145/229473.229474.

(43) Vassilev, V.; Vassilev, M.; Penev, A.; Moneta, L.; Ilieva, V. Clad — Automatic Differentiation Using Clang and LLVM. *Journal of Physics: Conference Series* **2015**, *608*, 012055, DOI: 10.1088/1742-6596/608/1/012055.

(44) Hogan, R. J. Fast Reverse-Mode Automatic Differentiation using Expression Templates in C++. *ACM Transactions on Mathematical Software* **2014**, *40*, 1–16, DOI: 10.1145/2560359.

(45) Leal, A. M. M. autodiff, a modern, fast and expressive C++ library for automatic differentiation. 2018 (accessed Oct 29, 2021); https://autodiff.github.io/.

(46) Leal, A. M. Reaktoro: A unified framework for modeling chemically reactive systems. 2015 (accessed Oct 29, 2021); www.reaktoro.org.

(47) Leal, A. M. M.; Kulik, D. A.; Smith, W. R.; Saar, M. O. An overview of computational methods for chemical equilibrium and kinetic calculations for geochemical and reactive transport modeling. *Pure and Applied Chemistry* **2017**, *89*, 597–643, DOI: 10.1515/pac-2016-1107.

(48) Jakob, W.; Rhinelander, J.; Moldovan, D. pybind11 – Seamless operability between C++11 and Python. 2016; https://github.com/pybind/pybind11.

(49) Gross, J.; Sadowski, G. Perturbed-Chain SAFT: An Equation of State Based on a Perturbation Theory for Chain Molecules. *Ind. Eng. Chem. Res.* **2001**, *40*, 1244–1260, DOI: 10.1021/ie0003887.

(50) Kunz, O.; Wagner, W. The GERG-2008 Wide-Range Equation of State for Natural Gases and Other Mixtures: An Expansion of GERG-2004. *J. Chem. Eng. Data* **2012**, *57*, 3032–3091, DOI: 10.1021/je300655b.

(51) Lemmon, E. W.; McLinden, M. O.; Wagner, W. Thermodynamic Properties of Propane. III. A Reference Equation of State for Temperatures from the Melting Line to 650 K and Pressures up to 1000 MPa. *J. Chem. Eng. Data* **2009**, *54*, 3141–3180, DOI: 10.1021/je900217v.

(52) Fornberg, B. Generation of finite difference formulas on arbitrarily spaced grids. *Math. Comp.* **1988**, *51*, 699–699, DOI: 10.1090/s0025-5718-1988-0935077-0.

(53) Householder, A. S. *The numerical treatment of a single nonlinear equation*; McGraw-Hill: London, 1970.

(54) Lötgering-Lin, O.; Fischer, M.; Hopp, M.; Gross, J. Pure Substance and Mixture Viscosities Based on Entropy Scaling and an Analytic Equation of State. *Ind. Eng. Chem. Res.* **2018**, *57*, 4095–4114, DOI: 10.1021/acs.iecr.7b04871.

(55) Yang, X.; Xiao, X.; May, E. F.; Bell, I. H. Entropy Scaling of Viscosity—III: Application to Refrigerants and Their Mixtures. *J. Chem. Eng. Data* **2021**, *66*, 1385–1398, DOI: 10.1021/acs.jced.0c01009.

(56) Wagner, W.; Pruß, A. The IAPWS Formulation 1995 for the Thermodynamic Properties of Ordinary Water Substance for General and Scientific Use. *J. Phys. Chem. Ref. Data* **2002**, *31*, 387–535, DOI: 10.1063/1.1461829.

(57) Deiters, U. K.; Bell, I. H. Calculation of phase envelopes of fluid mixtures through parametric marching. *AIChE J.* **2019**, *65*, e16730, DOI: 10.1002/aic.16730.

(58) Bell, I. H.; Alpert, B. K. Exceptionally reliable density-solving algorithms for multiparameter mixture models from Chebyshev expansion rootfinding. *Fluid Phase Equilib.* **2018**, *476B*, 89–102, DOI: 10.1016/j.fluid.2018.04.026.

(59) van Konynenburg, P. H.; Scott, R. L. Critical Lines and Phase Equilibria in Binary Van Der Waals Mixtures. *Philos. T. R. Soc. A* **1980**, *298*, 495–540, DOI: 10.1098/rsta.1980.0266.

(60) Mohr, P. J.; Newell, D. B.; Taylor, B. N.; Tiesinga, E. Data and analysis for the CODATA 2017 special fundamental constants adjustment. *Metrologia* **2018**, *55*, 125–146, DOI: 10.1088/1681-7575/aa99bc.

TOC Graphic