teqp

Release 0.19.1

Ian Bell

Mar 15, 2024

CONTENTS:

CHAPTER

ONE

GETTING STARTED

1.1 Introduction

teqp (phonetically: tɛk pi) is a C++-based library with wrappers. It was written because implementing EOS (particularly working out the derivatives) is a painful, error-prone, and slow process. The advent of open-source automatic differentiation libraries makes the implementation of EOS as fast as hand-written derivatives, and much easier to implement without errors.

There is a paper about teqp: https://doi.org/10.1021/acs.iecr.2c00237

The documentation is based on the Python wrapper because it can be readily integrated with the documentation tools (sphinx in this case) and can be auto-generated at documentation build time.

1.2 Installation

1.2.1 Python

The library can be installed with:

```
pip install teqp
```

because the binary wheels for all major platforms are provided on pypi.

If you desire to build teqp yourself, it is recommended to pull from github and build a binary wheel, and then subsequently install that wheel:

```
git clone --recursive https://github.com/usnistgov/teqp
cd teqp
python setup.py bdist_wheel
pip install dist/*.whl # or replace with the appropriate binary wheel
```

1.2.2 C++

The build is cmake based. There are targets available for an interface library, etc. Please see `CMakeLists.txt`

CHAPTER

TWO

C++ INTERFACE

2.1 Introduction

The abstract base class defining the public C++ interface of teqp is documented in AbstractModel. This interface was developed because re-compilation of the core of `teqp` is VERY slow, due to the heavy use of templates, which makes the code very flexible, but difficult to work with when doing development. Especially users that would like to only *use* the library but not be forced to pay the price of recompilation benefit from this approach.

As a user, a new model instance (a std::unique_ptr<teqp::AbstactModel*>) can be created by passing properly formatted JSON data structure to the make_model() function.

2.2 Object Model

The object model in teqp is convoluted because of the requirements to have models that use templated types to allow the use of automatic differentiation types. Instances of classes with templated methods cannot be stored directly in generic STL containers like `std::vector` or `std::list` (though they can be stored in `std::tuple`, but `tuple` cannot be constructed at runtime because they have complete type knowledge and C++ is strongly typed). Thus, some sort of wrapping is required (in C++ the technical term is type erasure) to store objects of a homogenous interface in dynamic containers like `std::vector`.

A number of type-erasure classes are defined, especially the DerivativeAdapter class which does type erasure on a model that it holds. This DerivativeAdapter class has an interface that takes STL types (and Eigen arrays in some cases) as input arguments, and then calls lower-level methods that can operate with a range of different numerical types, and call the templated methods of a model.

As a developer/implementer of a thermodynamic model, the class implementing the thermodynamic model for a contribution to α must satisfy the following requirements:

- It must have a method called `alphar` that takes three arguments that are all generic types. The first argument is the temperature, the second argument is the molar density, and the third is the mole fractions. In the case of some equations of state for model potentials, the temperature and density are treated as being in reduced units. The function should be called `alphar` even for Helmholtz energy contributions that are for ideal gases. You can think of the `r` in `alphar` standing for `reduced` instead of `residual` if that helps.
- It must have a method called `R` that takes a single argument that is the mole fractions of the components. It then returns the molar gas constant of the mixture. For most models it suffices to return 8.31446261815324, which is the CODATA value of the molar gas constant , and is available in the teqp::constants namespace. The reason the `R` method must be implemented is the multiparameter models in which the molar gas constant of diffent components is slightly different based upon when the EOS was published. Also, some of the other models used different values of R (or Avogadro's constant) when being developed and if you want to get perfect reproducibility these details matter.

This model instance is then passed to one of two methods in the teqp::cppinterface::adapter namespace: teqp::cppinterface::adapter::make_owned() or teqp::cppinterface::adapter::make_cview(). As the name suggests, if you pass the class instance to the `make_owned` function, it takes ownership of the model and the argument passed to the function is invalidated. On the contrary, the `make_cview` method is just a "viewer" of the model without taking ownership, so you need to watch out that the lifetime of the model you pass to this function is longer than the time you are using the wrapper model.

For instance this minimal working model of the van der Waals EOS demonstrates some of the things to be aware of:

```
/// A (very) simple implementation of the van der Waals EOS
class myvdWEOS1 {
public:
    const double a, b;
    myvdWEOS1(double a, double b) : a(a), b(b) {};

    /// \brief Get the universal gas constant
    template<class VecType>
    auto R(const VecType& /*molefrac*/) const { return constants::R_CODATA2017; }

    /// The evaluation of \f$ \alpha^{\rm r}=a/(RT) \f$
    /// \param T The temperature
    /// \param rhotot The molar density
    /// \param molefrac The mole fractions of each component
    template<typename TType, typename RhoType, typename VecType>
    auto alphar(const TType &T, const RhoType& rhotot, const VecType &molefrac) const
↪{
        return teqp::forceeval(-log(1.0 - b * rhotot) - (a / (R(molefrac) * T)) *␣
↪rhotot);
    }
};
```

The name of the class is entirely arbitrary, you could call it just as well `GreatVdWModel` instead of `myvdWEOS1`.

A complete example could then read:

```
#include <catch2/catch_test_macros.hpp>

#include "teqp/cpp/teqpcpp.hpp"
#include "teqp/cpp/deriv_adapter.hpp"
#include "teqp/types.hpp"
#include "teqp/constants.hpp"

/// A (very) simple implementation of the van der Waals EOS
class myvdWEOS1 {
public:
    const double a, b;
    myvdWEOS1(double a, double b) : a(a), b(b) {};

    /// \brief Get the universal gas constant
    template<class VecType>
    auto R(const VecType& /*molefrac*/) const { return constants::R_CODATA2017; }

    /// The evaluation of \f$ \alpha^{\rm r}=a/(RT) \f$
    /// \param T The temperature
    /// \param rhotot The molar density
    /// \param molefrac The mole fractions of each component
    template<typename TType, typename RhoType, typename VecType>
    auto alphar(const TType &T, const RhoType& rhotot, const VecType &molefrac) const
```

(continues on next page)

(continued from previous page)

```
↪{
        return teqp::forceeval(-log(1.0 - b * rhotot) - (a / (R(molefrac) * T)) *␣
↪rhotot);
    }
};

TEST_CASE("Check adding a model at runtime"){
    using namespace teqp::cppinterface;
    using namespace teqp::cppinterface::adapter;

    auto j = R"(
    {"kind": "myvdW", "model": {"a": 1.2, "b": 3.4}}
    )"_json;

    ModelPointerFactoryFunction func = [](const nlohmann::json& j){ return make_
↪owned(myvdWEOS1(j.at("a"), j.at("b"))); };
    add_model_pointer_factory_function("myvdW", func);

    auto ptr = make_model(j);
}
```

In this runnable example (runnable once the include paths are correct and the code is linked against the `teqpcpp` C++ library), a new factory function is registered with the add_model_pointer_factory_function() function and then this function is used to generate a `std::unique_ptr<AbstractModel*>`. Once the model has been created, it is possible to cast it back to the original type, but you must know the type of the class that you are holding (at compile time). The teqp::cppinterface::adapter::get_model_cref() is a convenience function to do this casting.

2.3 C++ Details

2.3.1 Don’t return expressions

The most important thing to be sure of when developing models in teqp is that you do not return expressions from functions. For instance in the simple function:

```
template<typename T1, typename T2>
auto alphar(const T1 &v1, const T2& v2) {
    return v1 + v2;
}
```

if the types of `T1` and `T2` are both `autodiff::real` (the same problem occurs for other autodiff types), the value of `v1 + v2` is an expression type that is lazily evaluated, and the expression holds references to the actual values of the variables `v1` and `v2`. This lazy evaluation is how autodiff can be so fast. Once the expression is returned from this function, the variables that it was pointing to are no longer valid because they have fallen out of scope and you can silently be pointing to invalid memory locations.

In order to avoid this problem you can use the function `teqp::forceeval` to force the evaluation of the expression, copying all the variables into the expression, and removing the possibility of dangling references after the function returns.

One way to ensure that you are not running into this problem is to enable the Address Sanitizer option “Detect Use of stack after return” in XCode (its in the Diagnostic panel of the “Edit Scheme…” option). Other address sanitizer tools have similar functionality.

2.3.2 Generic return types

Taking the example shown above, in the function `alphar` all the arguments have templated type. Sometimes you will need to make use of one or more of the types in intermediate calculations within the function, and you might need to determine the type of an expression to for instance allocate a vector of this type. As an example, let's say that we are going to multiply three different variables together. In the `alphar` context, let's assume that `T` is of type double, `rhomolar` is of type `std::complex<double>` and `molefracs` is of type `Eigen::ArrayXcd`. In the case of the expression `T*rhomolar*molefracs[0]`, the result will be calculated based on the type promotion to a `std::complex<double>`, so the result type of this product is `std::complex<double>`. If you want to let the compiler determine this type for you, you can do:

```
using resulttype = std::common_type_t<double, std::complex<double>,␣
↪decltype(molefracs[0])>;
```

and if you need want to work with the types of the variables, usually because you need to cover all your bases for all the templat permutations, you can do instead

```
using resulttype = std::common_type_t<decltype(T), decltype(rhomolar),␣
↪decltype(molefracs[0])>;
std::vector<resulttype> buffer;
```

and if you need to remove the `const` of your variable types, you can do with `std::decay_t< >`.

CHAPTER

THREE

DERIVATIVES

3.1 Thermodynamic Derivatives

3.1.1 Helmholtz energy derivatives

Thermodynamic derivatives are at the very heart of teqp. All models are defined in the form $\alpha^r(T,\rho,z)$, where ρ is the molar density, and z are mole fractions. There are exceptions for models for which the independent variables are in simulation units (Lennard-Jones and its ilk).

Thereofore, to obtain the residual pressure, it is obtained as a derivative:

$$p^r = \rho RT \left(\rho \left(\frac{\partial \alpha^r}{\partial \rho} \right)_T \right)$$

and other residual thermodynamic properties are defined likewise.

We can define the concise derivative

$$\Lambda^r_{xy} = (1/T)^x (\rho)^y \left(\frac{\partial^{x+y} (\alpha^r)}{\partial (1/T)^x \partial \rho^y} \right)$$

so we can re-write the derivative above as

$$p^r = \rho RT \Lambda^r_{01}$$

Similar definitions apply for all the other thermodynamic properties, with the tot superscript indicating it is the sum of the residual and ideal-gas (not included in teqp) contributions:

$$\frac{p}{\rho RT} = 1 + \Lambda^r_{01}$$

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

$$\frac{u}{RT} = \Lambda^{\rm tot}_{10}$$

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

$$\frac{h}{RT} = 1 + \Lambda^r_{01} + \Lambda^{\rm tot}_{10}$$

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

$$\frac{s}{R} = \Lambda^{\rm tot}_{10} - \Lambda^{\rm tot}_{00}$$

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

$$\frac{g}{RT} = 1 + \Lambda^r_{01} + \Lambda^{\rm tot}_{00}$$

Derivatives of pressure:

$$\left(\frac{\partial p}{\partial \rho}\right)_T = RT\left(1 + 2\Lambda^r_{01} + \Lambda^r_{02}\right)$$

$$\left(\frac{\partial p}{\partial T}\right)_\rho = R\rho\left(1 + \Lambda^r_{01} - \Lambda^r_{11}\right)$$

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

$$\frac{c_v}{R} = -\Lambda^{\rm tot}_{20}$$

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

$$\frac{c_p}{R} = -\Lambda^{\rm tot}_{20} + \frac{(1 + \Lambda^r_{01} - \Lambda^r_{11})^2}{1 + 2\Lambda^r_{01} + \Lambda^r_{02}}$$

In teqp, these derivatives are obtained from methods like

- *`get_Arxy()`*
- *`get_Ar06n()`*

where the `A` in this context indicates the variable Λ above. This naming is perhaps not ideal, since A is sometimes the total Helmholtz energy, but it was a close visual mnemonic to the character Λ.

```
[1]: import teqp
     teqp.__version__
```

```
[1]: '0.19.1'
```

```
[2]: import numpy as np
```

```
[3]: Tc_K = [300]
     pc_Pa = [4e6]
     acentric = [0.01]
     model = teqp.canonical_PR(Tc_K, pc_Pa, acentric)
```

```
[4]: z = np.array([1.0])
     model.get_Ar01(300,300,z)
```

```
[4]: -0.06836660379313926
```

And there are additional methods to obtain all the derivatives up to a given order:

```
[5]: model.get_Ar06n(300,300,z) # derivatives 00, 01, 02, ... 06
```

```
[5]: array([-6.96613834e-02, -6.83666038e-02,  2.53578225e-03, -1.57011622e-04,
             1.68186288e-05, -2.23059409e-06,  3.82592585e-07])
```

But more derivatives are slower than fewer:

```
[6]: %timeit model.get_Ar01(300,300,z)
     %timeit model.get_Ar04n(300,300,z)
```

```
601 ns ± 2.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

```
1.11 µs ± 2.43 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Note: calling overhead is usually on the order of 1 microsecond

3.1.2 Virial coefficients

Virial coefficients represent the thermodynamics of the interaction of two-, three-, … bodies interacting with each other. They can be obtained rigorously if the potential energy surface of interaction is fully known. In general, such a surface can only be constructed for small rigid molecules. Many simple thermodynamic models do a poor job of predicting the thermodynamics captured by the virial coefficients.

The i-th virial coefficient is defined by

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

with the concise derivative term

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

teqp supports the virial coefficient directly, there is the `get_B2vir` method for the second virial coefficient:

```
[7]: model.get_B2vir(300, z)
```

```
[7]: -0.00023661263734465424
```

And the `get_Bnvir` method that allows for the calculation of higher virial coefficients:

```
[8]: model.get_Bnvir(7, 300, z)
```

```
[8]: {2: -0.0002366126373446542,
      3: 3.001768410777936e-08,
      4: -3.2409760373816364e-12,
      5: 3.961781646633723e-16,
      6: -4.5529239838367004e-20,
      7: 5.375927851118494e-24}
```

The `get_Bnvir` method was implemented because when doing automatic differentiation, all the intermediate derivatives are also retained.

There is also a method to calculate temperature derivatives of a given virial coefficient

```
[9]: model.get_dmBnvirdTm(2, 3, 300, z) # third temperature derivative of the second␣
     ↪virial coefficient
```

```
[9]: 1.0095625628421257e-10
```

3.1.3 Isochoric Thermodynamics Derivatives

In the isochoric thermodynamics formalism, the EOS is expressed in the Helmholtz energy density Ψ as a function of temperature and molar densities $\vec{\rho}$. This formalism is handy because it allows for a concise mathematical structure, well suited to implementation in teqp. For instance the pressure is obtained from (see https://doi.org/10.1002/aic.16074):

$$p = -\Psi + \sum_{i=1}^{N} \rho_i \mu_i$$

with the chemical potential μ_i obtained from

$$\mu_i = \left(\frac{\partial \Psi}{\partial \rho_i}\right)_{T,\rho_{j\neq i}}$$

The molar densities ρ_i are related to the total density and the mole fractions:

$$\rho_i = x_i \rho$$

In teqp, the isochoric derivative functions like `get_fugacity_coefficients`, `get_partial_molar_volumes` take as arguments the temperature `T` and the vector of molar concentrations `rhovec`$=\vec{\rho}$, which are obtained by multiplying the mole fractions by the total density.

Example:

[10]:
```
model = teqp.build_multifluid_model(["CO2","Argon"], teqp.get_datapath())
T, rhovec = 300, np.array([0.3,0.4])*300 # K, mol/m^3
display(model.get_fugacity_coefficients(T, rhovec))
display(model.get_partial_molar_volumes(T, rhovec))
```

```
array([0.97884567, 0.99866748])
```

```
array([0.00470644, 0.00480351])
```

3.2 Composition derivatives

For other mixture calculations composition derivatives of the form

$$\Lambda^r_{xyz_i} = (1/T)^x(\rho)^y\left(\frac{\partial^{x+y+z_i}(\alpha^r)}{\partial(1/T)^x\partial\rho^y\partial\mathbf{Z}_i^{z_i}}\right)$$

are needed. This function is exposed in teqp (as of version 0.19) as the function `get_ATrhoXi`. In order to limit the binary size and compilation time, x has a max of 2 and y does as well. `z_i` can be up to 3, and must be at least 1, otherwise you can use the other derivative functions that do not require any composition derivatives.

The mixed composition derivative of the form

$$\Lambda^r_{xyz_iz_j} = (1/T)^x(\rho)^y\left(\frac{\partial^{x+y+z_i+z_j}(\alpha^r)}{\partial(1/T)^x\partial\rho^y\partial\mathbf{Z}_i^{z_i}\partial\mathbf{Z}_j^{z_j}}\right)$$

supports x and y either 0 or 1, with at most two composition derivatives. The triple composition derivative of the form

$$\Lambda^r_{xyz_iz_jz_k} = (1/T)^x(\rho)^y\left(\frac{\partial^{x+y+z_i+z_j+z_k}(\alpha^r)}{\partial(1/T)^x\partial\rho^y\partial\mathbf{Z}_i^{z_i}\partial\mathbf{Z}_j^{z_j}\partial\mathbf{Z}_k^{z_k}}\right)$$

supports x and y either 0 or 1, with up to first derivatives in each composition variable. If this is not enough derivatives, open a feature request here : https://github.com/usnistgov/teqp/issues

3.2.1 τ and δ derivatives

In the multi-fluid modeling approach used in NIST REFPROP and the GERG-2004 GERG-2008 models, the derivatives are in the form

$$\Lambda^r_{xyz_i} = \tau^x\delta^y\left(\frac{\partial^{x+y+z_i}(\alpha^r)}{\partial\tau^x\partial\delta^y\partial\mathbf{Z}_i^{z_i}}\right)$$

with $\tau = T_{\rm red}(\mathbf{Z})/T$ and $\delta = \rho/\rho_{\rm red}(\mathbf{Z})$. The higher derivatives are similarly equal to

$$\Lambda^r_{xyz_iz_j} = \tau^x\delta^y\left(\frac{\partial^{x+y+z_i+z_j}(\alpha^r)}{\partial\tau^x\partial\delta^y\partial\mathbf{Z}_i^{z_i}\partial\mathbf{Z}_j^{z_j}}\right)$$

$$\Lambda^r_{xyz_iz_jz_k} = \tau^x\delta^y\left(\frac{\partial^{x+y+z_i+z_j+z_k}(\alpha^r)}{\partial\tau^x\partial\delta^y\partial\mathbf{Z}_i^{z_i}\partial\mathbf{Z}_j^{z_j}\partial\mathbf{Z}_k^{z_k}}\right)$$

The same limitations on the numbers of derivatives are used for the derivatives with $(1/T)$ and ρ as independent variables.

The Python methods are documented here:

- `get_ATrhoXi()`
- `get_ATrhoXiXj()`
- `get_ATrhoXiXjXk()`
- `get_AtaudeltaXi()`
- `get_AtaudeltaXiXj()`
- `get_AtaudeltaXiXjXk()`

3.2.2 xN (in)dependent

Let’s suppose that some quantity Υ depends on mole fractions. If all the mole fractions are considered to be independent, the total differential is obtained from

$$\mathrm{d}\Upsilon = \sum_j \left(\frac{\partial\Upsilon}{\partial x_j}\right)_{x_{i\neq j}}\mathrm{d}x_j$$

If instead the last mole fraction is defined to be dependent on the others via

$$x_N = 1 - \sum_{i=1}^{N-1} x_i$$

then the total differential is obtained from

$$\mathrm{d}\Upsilon = \sum_{j=1}^{N-1} \left(\frac{\partial\Upsilon}{\partial x_j}\right)_{x_{i\neq j}}\mathrm{d}x_j + \left(\frac{\partial\Upsilon}{\partial x_N}\right)_{x_{i\neq j}}\mathrm{d}x_N$$

where x_N is considered to be an independent variable in the derivative $\left(\frac{\partial\Upsilon}{\partial x_N}\right)_{x_{i\neq j}}$. Thus derivatives with respect to one of the dependent mole fractions (x_k with $k < N$) would be equal to

$$\left(\frac{\partial\Upsilon}{\partial x_k}\right)_{x_{j\neq k}} = \left(\frac{\partial\Upsilon}{\partial x_k}\right)_{x_{i\neq k}} - \left(\frac{\partial\Upsilon}{\partial x_N}\right)_{x_{i\neq N}}$$

because

$$\left(\frac{\partial x_N}{\partial x_i}\right) = -1$$

So if the library (e.g., CoolProp and TREND) allows for the fractions to be dependent (either option is allowed in CoolProp, TREND uses N-1 independent mole fractions), you can use the molar composition derivatives with the mole fractions treated as being independent to obtain derivatives with one of the mole fractions dependent on the other $N-1$ fractions.

```
[1]: import teqp, numpy as np
     teqp.__version__
```

```
[1]: '0.19.1'
```

```
[2]: j = {
         'components': ["Methane", "Nitrogen", "Oxygen"],
         'root': teqp.get_datapath(),
         'BIP': '',
         'departure': ''
     }
     model = teqp.make_model({'kind':'multifluid', 'model': j})


     T = 300 # K
     rhomolar = 3000 # mol/m^3
     z = np.array([0.3, 0.5, 0.2]) # mole fractions

     Tr = model.get_Tr(z)
     rhor = model.get_rhor(z)
     tau = Tr/T
     delta = rhomolar/rhor

     Ntau = 0
     Ndelta = 0
     Nxi = 1
     print(model.get_AtaudeltaXi(tau, Ntau, delta, Ndelta, z, 0, Nxi))

     Ntau = 1
     Ndelta = 0
     Nxi = 1
     print(model.get_AtaudeltaXi(tau, Ntau, delta, Ndelta, z, 0, Nxi))

     Ntau = 0
     Ndelta = 1
     Nxi = 1
     print(model.get_AtaudeltaXi(tau, Ntau, delta, Ndelta, z, 0, Nxi))

     Ntau = 2
     Ndelta = 0
     Nxi = 1
     print(model.get_AtaudeltaXi(tau, Ntau, delta, Ndelta, z, 0, Nxi))

     Ntau = 1
     Ndelta = 1
     Nxi = 1
     print(model.get_AtaudeltaXi(tau, Ntau, delta, Ndelta, z, 0, Nxi))
```

(continues on next page)

(continued from previous page)

```
Ntau = 0
Ndelta = 2
Nxi = 1
print(model.get_AtaudeltaXi(tau, Ntau, delta, Ndelta, z, 0, Nxi))

Ntau = 1
Ndelta = 0
Nxi = 1
Nxj = 1
print(model.get_AtaudeltaXiXj(tau, Ntau, delta, Ndelta, z, 0, Nxi, 1, Nxj))

Ntau = 0
Ndelta = 1
Nxi = 1
Nxj = 1
print(model.get_AtaudeltaXiXj(tau, Ntau, delta, Ndelta, z, 0, Nxi, 1, Nxj))

Ntau = 0
Ndelta = 0
Nxi = 1
Nxj = 1
Nxk = 1
print(model.get_AtaudeltaXiXjXk(tau, Ntau, delta, Ndelta, z, 0, Nxi, 1, Nxj, 2, Nxk))
```

```
-0.043587384253511226
-0.2118857998812584
-0.03650566667904927
-0.07488856488580686
-0.2069389009652925
0.014468933385218782
-0.005978809921279949
-0.00279185550001082
0.0
```

With CoolProp, version 6.6.0, the following script in C++:

```
#include "AbstractState.h"
#include "Backends/Helmholtz/MixtureDerivatives.h"

int main(){
    std::shared_ptr<CoolProp::AbstractState> AS(
        CoolProp::AbstractState::factory("HEOS","Methane&Nitrogen&Oxygen")
    );
    AS->set_mole_fractions({0.3, 0.5, 0.2});
    AS->specify_phase(CoolProp::iphase_gas);
    AS->update(CoolProp::DmolarT_INPUTS, 3000, 300);
    auto& HEOS = *dynamic_cast<CoolProp::HelmholtzEOSMixtureBackend*>(AS.get());
    auto xN  = CoolProp::x_N_dependency_flag::XN_INDEPENDENT;
    using md = CoolProp::MixtureDerivatives;
    std::cout << md::dalphar_dxi(HEOS, 0, xN) << std::endl;

    std::cout << md::d2alphar_dxi_dTau(HEOS, 0, xN)*AS->tau() << std::endl;
    std::cout << md::d2alphar_dxi_dDelta(HEOS, 0, xN)*AS->delta() << std::endl;
    std::cout << md::d3alphar_dxi_dTau2(HEOS, 0, xN)*pow(AS->tau(), 2) << std::endl;
    std::cout << md::d3alphar_dxi_dDelta_dTau(HEOS, 0, xN)*AS->tau()*AS->delta() <<␣
↪std::endl;
    std::cout << md::d3alphar_dxi_dDelta2(HEOS, 0, xN)*pow(AS->delta(), 2) << std::
```

(continues on next page)

(continued from previous page)

```
↪endl;

    std::cout << md::d3alphar_dxi_dxj_dTau(HEOS, 0, 1, xN)*AS->tau() << std::endl;
    std::cout << md::d3alphar_dxi_dxj_dDelta(HEOS, 0, 1, xN)*AS->delta() << std::endl;
    std::cout << md::d3alphardxidxjdxk(HEOS, 0, 1, 2, xN) << std::endl;
}
```

yields the output:

```
-0.0435874
-0.211886
-0.0365057
-0.0748886
-0.206939
0.0144689
-0.00597881
-0.00279186
0
```

which is the same as the above

CHAPTER
FOUR

MODELS

4.1 Constructing Models

With a few exceptions, most models are constructed by describing the model in JSON format, and passing the JSON-formatted information to the `make_model` function. There are some convenience functions exposed for backwards compatibility, but as of version 0.14.0, all model construction should go via this route.

At the C++ level, the returned value from the `make_model` function is a `shared_ptr` that wraps a pointer to an `AbstractModel` class. The `AbstractModel` class is an abstract class which defines the public C++ interface.

In Python, construction is in two parts. First, the model is constructed, which only includes the common methods. Then, the model-specific attributes and methods are attached with the `attach_model_specific_methods` method.

The JSON structure is of two parts, the `kind` field is a case-sensitive string defining which model kind is being constructed, and the `model` field contains all the information needed to build the model. In the case of hard-coded models, nothing is provided in the `model` field, but it must still be provided.

Also, the argument to `make_model` must be valid JSON. So if you are working with numpy array datatypes, make sure to convert them to a list (which is convertible to JSON). Example below.

[1]:
```python
import teqp, numpy as np
teqp.__version__
```

[1]:
```
'0.19.1'
```

[2]:
```python
teqp.make_model({'kind': 'vdW1', 'model': {'a': 1, 'b': 2}})
```

[2]:
```
<teqp.teqp.AbstractModel at 0x7d6db379c050>
```

[3]:
```python
# Fields are case-sensitive
teqp.make_model({'kind': 'vdW1', 'model': {'a': 1, 'B': 2}})
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[3], line 2
      1 # Fields are case-sensitive
----> 2 teqp.make_model({'kind': 'vdW1', 'model': {'a': 1, 'B': 2}})

File /opt/conda/lib/python3.11/site-packages/teqp/__init__.py:47, in make_model(*args,
↪ **kwargs)
     42 def make_model(*args, **kwargs):
     43     """
     44     This function is in two parts; first the make_model function (renamed to _
↪make_model in the Python interface)
     45     is used to make the model and then the model-specific methods are␣
```

(continues on next page)

(continued from previous page)

```
↪attached to the instance
     46     """
---> 47     AS = _make_model(*args, **kwargs)
     48     attach_model_specific_methods(AS)
     49     return AS

RuntimeError: :{"B":2,"a":1}': required property 'b' not found in object
|/|\|:{"B":2,"a":1}': validation failed for additional property 'B': instance invalid␣
↪as per false-schema
```

```
[4]: # A hard-coded model
teqp.make_model({
    'kind': 'AmmoniaWaterTillnerRoth',
    'model': {}
})
```

```
[4]: <teqp.teqp.AbstractModel at 0x7d6d9cda79b0>
```

```
[5]: # Show what to do with numpy array
Tc_K = np.array([100,200])
pc_Pa = np.array([3e6, 4e6])
teqp.make_model({
    "kind": "vdW",
    "model": {
        "Tcrit / K": Tc_K.tolist(),
        "pcrit / Pa": pc_Pa.tolist()
    }
})
```

```
[5]: <teqp.teqp.AbstractModel at 0x7d6d9cda7b90>
```

```
[6]: # methane with conventional PC-SAFT
j = {
    'kind': 'PCSAFT',
    'model': {
        'coeffs': [{
            'name': 'methane',
             'BibTeXKey': 'Gross-IECR-2001',
             'm': 1.00,
             'sigma_Angstrom': 3.7039,
             'epsilon_over_k': 150.03,
        }]
    }
}
model = teqp.make_model(j)
```

4.2 General cubics

The reduced residual Helmholtz energy for the main cubic EOS (van der Waals, Peng-Robinson, and Soave-Redlich-Kwong) can be written in a common form (see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7365965/)

$$\alpha^r = \psi^{(-)} - \frac{\tau a_m}{RT_r}\psi^{(+)}$$

$$\psi^{(-)} = -\ln(1 - b_m\rho)$$

$$\psi^{(+)} = \frac{\ln\left(\frac{\Delta_1 b_m\rho + 1}{\Delta_2 b_m\rho + 1}\right)}{b_m(\Delta_1 - \Delta_2)}$$

with the constants given by:

- vdW: $\Delta_1 = 0$, $\Delta_2 = 0$
- SRK: $\Delta_1 = 1$, $\Delta_2 = 0$
- PR: $\Delta_1 = 1 + \sqrt{2}$, $\Delta_2 = 1 - \sqrt{2}$

The quantities a_m and b_m are described (with exact solutions for the numerical coefficients) for each of these EOS in https://pubs.acs.org/doi/abs/10.1021/acs.iecr.1c00847.

The models in teqp are instantiated based on knowledge of the critical temperature, pressure, and acentric factor. Thereafter all quantities are obtained from derivatives of α^r.

The Python class is here: `GeneralizedCubic`

```
[1]: import teqp
     teqp.__version__
```

```
[1]: '0.19.1'
```

```
[2]: import json
     import CoolProp.CoolProp as CP

     # Values taken from http://dx.doi.org/10.6028/jres.121.011
     Tc_K = [ 190.564, 154.581, 150.687 ]
     pc_Pa = [ 4599200, 5042800, 4863000 ]
     acentric = [ 0.011, 0.022, -0.002 ]

     # Instantiate Peng-Robinson model
     modelPR = teqp.canonical_PR(Tc_K, pc_Pa, acentric)

     # Instantiate Soave-Redlich-Kwong model
     modelSRK = teqp.canonical_SRK(Tc_K, pc_Pa, acentric)
```

```
[3]: # And you can get information about the model in JSON format
     # from the get_meta function
     modelPR.get_meta()
```

```
[3]: {'Delta1': 2.414213562373095,
      'Delta2': -0.41421356237309515,
      'OmegaA': 0.4572355289213822,
      'OmegaB': 0.07779607390388846,
      'kind': 'Peng-Robinson'}
```

4.2.1 Adjusting k_ij

Fine-tuned values of k_{ij} can be provided when instantiating the model, for Peng-Robinson and SRK. A complete matrix of all the k_{ij} values must be provided. This allows for asymmetric mixing models in which $k_{ij} \neq k_{ji}$.

```
[4]: k_12 = 0.01
     kmat = [[0,k_12,0],[k_12,0,0],[0,0,0]]
     teqp.canonical_PR(Tc_K, pc_Pa, acentric, kmat)
     teqp.canonical_SRK(Tc_K, pc_Pa, acentric, kmat)
```

```
[4]: <teqp.teqp.AbstractModel at 0x7894f86dd850>
```

4.2.2 Superancillary

The superancillary equation gives the co-existing liquid and vapor (orthobaric) densities as a function of temperature. The set of Chebyshev expansions was developed in https://pubs.acs.org/doi/abs/10.1021/acs.iecr.1c00847 . These superancillary equations are more accurate than iterative calculations in double precision arithmetic and also at least 10 times faster to calculate, and cannot fail in iterative routines, even extremely close to the critical point.

The superancillary equation is only exposed for pure fluids to remove ambiguity when considering mixtures. The returned tuple is the liquid and vapor densities

```
[5]: teqp.canonical_PR([Tc_K[0]], [pc_Pa[0]], [acentric[0]]).superanc_rhoLV(100)
```

```
[5]: (30846.392909514052, 42.480231719002326)
```

4.2.3 a and b

For the cubic EOS, it can be useful to obtain the a and b parameters directly. The b parameter is particularly useful because 1/b is the maximum allowed density in the EOS

```
[6]: import numpy as np
     z = np.array([0.3, 0.4, 0.3])
     modelPR.get_a(140, z), modelPR.get_b(140, z)
```

```
[6]: (0.1874177858906821, 2.1984349667726406e-05)
```

4.2.4 alpha functions

It can be advantageous to modify the alpha function to allow for more accurate handling of the attractive interactions. Coefficients are tabulated for many species in https://pubs.acs.org/doi/10.1021/acs.jced.7b00967 for the Peng-Robinson EOS with Twu alpha function and the values from the SI of that paper are in the csv file next to this file.

```
[7]: import pandas

     dfTwu = pandas.read_csv('fitted_Twu_coeffs.csv')
     def get_model(INCHIKey):
         row = dfTwu.loc[dfTwu['inchikey']==INCHIKey]
         if len(row) == 1:
             row = row.iloc[0]
         Tc_K = row['Tc_K']
         pc_MPa = row['pc_MPa']
         c = [row['c0'], row['c1'], row['c2']]
```

(continues on next page)

(continued from previous page)

```
    # The JSON definition of the EOS,
    # here a generic cubic EOS to allow for
    # specification of the alpha function(s)
    j = {
        'kind': 'cubic',
        'model': {
            'type': 'PR',
            'Tcrit / K': [Tc_K],
            'pcrit / Pa': [pc_MPa*1e6],
            'acentric': [0.1],
            'alpha': [{'type': 'Twu', 'c': c}]
        }
    }
    model = teqp.make_model(j)
    return model, j

# Hexane
model, j = get_model(INCHIKey='VLKZOEOYAKHREP-UHFFFAOYSA-N')
```

[8]:

```
# And how about we calculate the pressure and s^+ = -sr/R at NBP of water
model, j = get_model(INCHIKey='XLYOFNOQVPJJNP-UHFFFAOYSA-N')  # WATER

T = 373.1242958476844 # K, NBP of water
rhoL, rhoV = model.superanc_rhoLV(T)
z = np.array([1.0])
pL = rhoL*model.get_R(z)*T*(1.0 + model.get_Ar01(T, rhoL, z))
splusL = model.get_splus(T, rhoL*z)
print(pL, splusL)
```

```
102739.27983424198 6.03697343297877
```

Also implemented in version 0.17 are the alpha functions of Mathias-Copeman.

$$\alpha = (1 + c_0x + c_1x^2 + c_2x^3)^2$$

with

$$x = 1 + \sqrt{\frac{T}{T_ci}}$$

Parameters are tabulated for many fluids in the paper of Horstmann (https://doi.org/10.1016/j.fluid.2004.11.002) for the SRK EOS (only)

[9]:

```
# Here is an example from Horstmann
j = {
    "kind": "cubic",
    "model": {
        "type": "SRK",
        "Tcrit / K": [647.30],
        "pcrit / Pa": [22.048321e6],
        "acentric": [0.3440],
        "alpha": [
            {"type": "Mathias-Copeman", "c": [1.07830, -0.58321, 0.54619]}
        ]
    }
}
```

(continues on next page)

(continued from previous page)

```
model = teqp.make_model(j)
T = 373.1242958476844 # K
rhoL, rhoV = model.superanc_rhoLV(T)
z = np.array([1.0])
pL = rhoL*model.get_R(z)*T*(1.0 + model.get_Ar01(T, rhoL, z))
print('And with SRK and Mathias-Copeman parameters:', pL, 'Pa')
```

```
And with SRK and Mathias-Copeman parameters: 101639.22259842217 Pa
```

4.3 Quantum PR

The quantum-corrected Peng-Robinson model of Aasen *et al.* (https://doi.org/10.1063/1.5111364) can be used to account for quantum effects by empirical fits to the Feynman-Hibbs corrections.

The conventional Peng-Robinson approach is used, with an adjusted covolume b_i given by

$$b_i = b_{i,PR}\beta_i(T)$$

with

$$\beta_i(T) = \left(\frac{1 + A_i/(T+B_i)}{1 + A_i/(T_{ci}+B_i)}\right)^3$$

and Twu alpha functions are used to correct the attractive part.

[1]:
```
import numpy as np, matplotlib.pyplot as plt, pandas
import CoolProp.CoolProp as CP

import teqp
teqp.__version__
```

[1]:
```
'0.19.1'
```

[2]:
```
kij_library = {
    ('H2','Ne'): 0.18,
    ('He','H2'): 0.17
}
lij_library = {
    ('H2','Ne'): 0.0,
    ('He','H2'): -0.16
}

def get_model(names, c_factor=0):
    param_library = {
        'H2': {
            "Ls": [156.21],
            "Ms": [-0.0062072],
            "Ns": [5.047],
            "As": [3.0696],
            "Bs": [12.682],
            "cs / m^3/mol": [c_factor*-3.8139e-6],
            "Tcrit / K": [33.19],
            "pcrit / Pa": [12.964e5]
        },
```

(continues on next page)

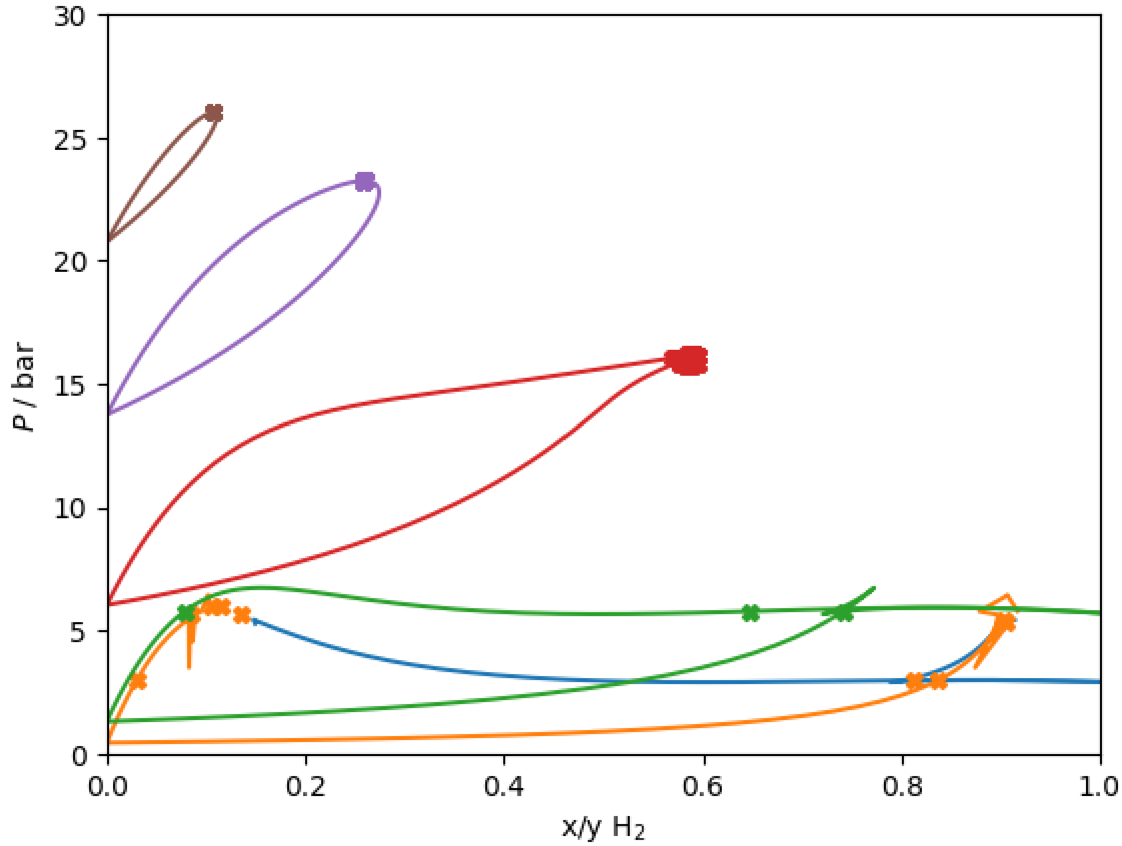
(continued from previous page)

```
        'Ne': {
            "Ls": [0.40453],
            "Ms": [0.95861],
            "Ns": [0.8396],
            "As": [0.4673],
            "Bs": [2.4634],
            "cs / m^3/mol": [c_factor*-2.4665e-6],
            "Tcrit / K": [44.492],
            "pcrit / Pa": [26.79e5]
        },
        'He': {
            "Ls": [0.48558],
            "Ms": [1.7173],
            "Ns": [0.30271],
            "As": [1.4912],
            "Bs": [3.2634],
            "cs / m^3/mol": [c_factor*-3.1791e-6],
            "Tcrit / K": [5.1953],
            "pcrit / Pa": [2.276e5]
        }
    }
    params = [param_library[name] for name in names]
    model = {k: [param[k][0] for param in params] for k in ['Ls','Ms','Ns','As','Bs',
↪'cs / m^3/mol','Tcrit / K','pcrit / Pa']}

    if len(names) == 1:
        model['kmat'] = [[0]]
        model['lmat'] = [[0]]
    else:
        kij = kij_library[names]
        model['kmat'] = [[0,kij],[kij,0]]
        lij = lij_library[names]
        model['lmat'] = [[0,lij],[lij,0]]

    j = {
        "kind": "QCPRAasen",
        "model": model
    }
    return teqp.make_model(j), j

model = get_model(('H2','Ne'))[0]
modelH2 = get_model(('H2',))[0]
modelNe = get_model(('Ne',))[0]

def get_traces(T, ipures):
    traces = []
    for ipure in ipures:
        rhovecL0 = np.array([0.0, 0.0])
        rhovecV0 = np.array([0.0, 0.0])
        if ipure == 1:
            rhoL, rhoV = modelNe.superanc_rhoLV(T)
        else:
            rhoL, rhoV = modelH2.superanc_rhoLV(T)
        rhovecL0[ipure] = rhoL
        rhovecV0[ipure] = rhoV

        opt = teqp.TVLEOptions();
```

(continues on next page)

(continued from previous page)

```
#         opt.polish=True;
#         opt.integration_order=5; opt.rel_err=1e-10;
#         opt.calc_criticality = True;
        opt.crit_termination=1e-10
        trace = model.trace_VLE_isotherm_binary(T, rhovecL0, rhovecV0, opt)
        traces.append(trace)
    return traces

for T in [24.59, 28.0, 34.66, 39.57, 42.50]:
    if T < 26.0:
        traces = get_traces(T, [0, 1])
    else:
        traces = get_traces(T, [1])

    for trace in traces:
        df = pandas.DataFrame(trace)

        # Plot the VLE solution
        line, = plt.plot(df['xL_0 / mole frac.'], df['pL / Pa']/1e5)
        plt.plot(df['xV_0 / mole frac.'], df['pL / Pa']/1e5, color=line.get_color())

    # Plot the VLLE solution if found
    for soln in model.find_VLLE_T_binary(traces):
        for rhovec in soln['polished']:
            rhovec = np.array(rhovec)
            rhotot = sum(rhovec)
            x = rhovec/rhotot
            p = rhotot*model.get_R(x)*T*(1+model.get_Ar01(T, rhotot, x))
            plt.plot(x[0], p/1e5, 'X', color=line.get_color())
            # print(T, rhovec, x[0], p/1e5, 'bar')

plt.gca().set(xlabel='x/y H$_2$', ylabel='$P$ / bar', xlim=(0,1), ylim=(0,30));
```



4.4 Advanced cubic mixing rules

In the advanced cubic mixing rules, the conventional cubic EOS is taken as the basis for the method (usually Peng-Robinson), but different rules are used for the attractive term a_m. The formulation reads:

$$\frac{a_m}{b_m} = \sum_i z_i \frac{a_i}{b_i} + \frac{a^{E,\gamma}_{\rm res}}{CEoS}$$

where $CEoS$ is a scaling parameter that is in principle linked with the EOS coefficients, but can also be allowed to be an adjustable parameter. The a_i and b_i are the pure fluid values of component i. The z_i are mole fractions. The mixture covolume is given by

$$b_m = \sum_i \sum_j z_i z_j b_{ij}$$

with

$$b_{ij} = \left(\frac{b_i^{1/s} + b_j^{1/s}}{2}\right)^s$$

The heart of the method is the definition of $a^{E,\gamma}_{\rm res}$, the residual contribution (not in the conventional thermodynamic sense) to the excess Helmholtz energy. There are many possible models here, but one that seems to work well is that of Wilson, for which the expression reads:

$$\frac{a^{E,\gamma}_{\rm res}}{RT} = -\sum_i z_i \ln\left(\sum_j z_j\Omega_{ji}(T)\right) - \sum_i z_i \ln\left(\frac{\phi_i}{z_i}\right)$$

with

$$\Omega_{ji} = \frac{v_j}{v_i}\exp(-A_{ij}/T)$$

and

$$\frac{\phi_i}{z_i} = \frac{v_i}{\sum_k z_k v_k}$$

with the $v_i = b_i$. The parameter $A_{ij} \neq A_{ji}$ in general, and is also given temperature dependence, which is also not supposed to be present according to the derivation. Thus, the models for A_{ij} read something like this here:

$$A_{ij} = m_{ij}T + n_{ij}$$

so m is non-dimensional and n has units of temperature.

[1]:

```python
import numpy, matplotlib.pyplot as plt, numpy as np, pandas
import teqp
teqp.__version__
```

[1]:

```
'0.19.1'
```

[2]:

```python
# Four isotherms of experimental data from doi: 10.1016/j.fluid.2016.05.015
import io, pandas
dat = pandas.read_csv(io.StringIO("""PointID y1 uy1 x1 ux1 p/bar up T/K
1 0.0274 0.0007 0.0068 0.0002 59.830 0.053 293.10
2 0.0664 0.0014 0.0183 0.0004 64.864 0.080 293.10
3 0.0978 0.0020 0.0298 0.0007 69.772 0.080 293.10
4 0.1199 0.0024 0.0424 0.0009 74.737 0.080 293.10
5 0.1219 0.0028 0.1132 0.0023 89.869 0.080 293.10
6 0.1339 0.0024 0.0995 0.0022 89.198 0.080 293.10
7 0.1399 0.0026 0.0943 0.0020 88.853 0.080 293.10
8 0.1461 0.0027 0.0823 0.0019 86.962 0.080 293.10
9 0.1466 0.0028 0.0778 0.0017 85.942 0.080 293.10
10 0.1466 0.0028 0.0772 0.0016 85.868 0.080 293.10
1 0.1378 0.0027 0.0159 0.0004 42.667 0.051 273.08
2 0.2143 0.0038 0.0297 0.0007 49.547 0.051 273.08
3 0.2612 0.0043 0.0411 0.0009 55.238 0.051 273.08
4 0.3209 0.0049 0.0609 0.0013 65.069 0.088 273.08
5 0.3554 0.0051 0.0786 0.0016 73.395 0.088 273.08
6 0.3758 0.0052 0.0978 0.0019 81.061 0.088 273.08
7 0.3903 0.0053 0.1190 0.0023 90.706 0.088 273.08
8 0.3914 0.0053 0.1477 0.0028 100.966 0.088 273.08
9 0.3879 0.0053 0.1614 0.0030 104.806 0.088 273.08
10 0.3724 0.0052 0.1875 0.0033 110.846 0.088 273.08
11 0.3550 0.0051 0.2068 0.0036 114.105 0.088 273.08
12 0.2727 0.0044 0.2531 0.0041 118.020 0.088 273.08
13 0.3343 0.0049 0.2268 0.0038 116.295 0.088 273.08
1 0.2048 0.0038 0.0106 0.0003 25.754 0.050 253.05
2 0.3019 0.0049 0.0217 0.0005 30.479 0.050 253.05
3 0.4638 0.0056 0.0436 0.0010 45.352 0.050 253.05
4 0.5319 0.0056 0.0647 0.0014 58.188 0.050 253.05
```

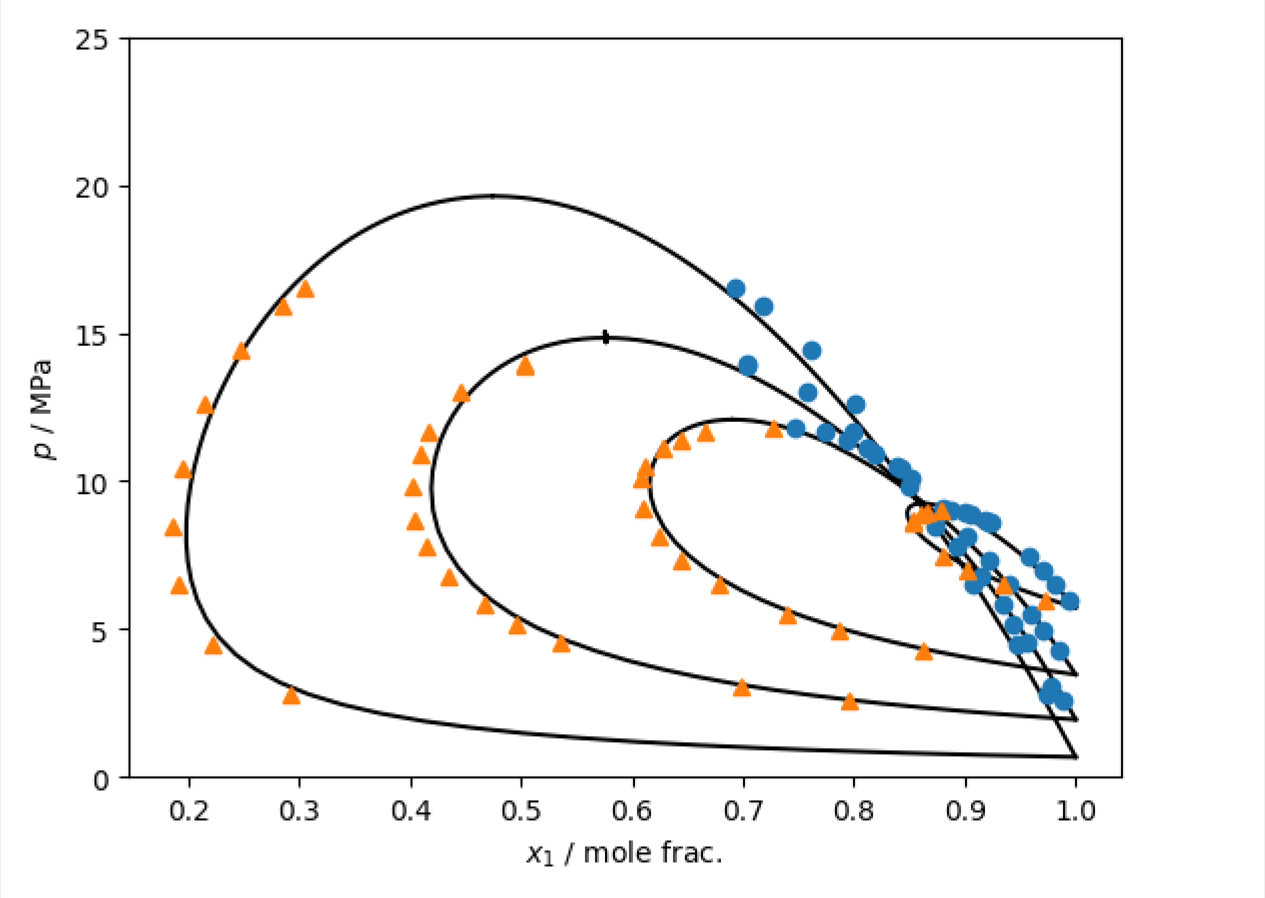
(continues on next page)

(continued from previous page)

```
5 0.5854 0.0054 0.1077 0.0021 78.315 0.084 253.05
6 0.5979 0.0054 0.1497 0.0028 98.276 0.084 253.05
7 0.5898 0.0054 0.1801 0.0032 109.241 0.084 253.05
8 0.5042 0.0057 0.0570 0.0012 51.343 0.084 253.05
9 0.5644 0.0055 0.0861 0.0017 67.594 0.084 253.05
10 0.5949 0.0054 0.1267 0.0024 86.883 0.084 253.05
11 0.5826 0.0054 0.2015 0.0035 116.614 0.084 253.05
12 0.5537 0.0055 0.2431 0.0040 129.873 0.084 253.05
13 0.4973 0.0055 0.2971 0.0046 139.161 0.084 253.05
14 0.4971 0.0055 0.2972 0.0046 139.261 0.084 253.05
1 0.7076 0.0050 0.0257 0.0006 27.983 0.056 223.10
2 0.7774 0.0041 0.0522 0.0011 44.918 0.056 223.10
3 0.8077 0.0036 0.0930 0.0019 64.906 0.081 223.10
4 0.8131 0.0035 0.1261 0.0024 84.799 0.081 223.10
5 0.8057 0.0035 0.1584 0.0029 104.410 0.081 223.10
6 0.7843 0.0038 0.1982 0.0035 125.782 0.081 223.10
7 0.7533 0.0041 0.2380 0.0040 144.287 0.081 223.10
8 0.7150 0.0045 0.2813 0.0044 159.015 0.081 223.10
9 0.6942 0.0047 0.3064 0.0047 165.347 0.081 223.10
"""), sep='\s+', engine='python')
```

[3]:

```
# Model from Lasala, FPE, 2016: https://doi.org/10.1016/j.fluid.2016.05.015
j = {
    "kind": "advancedPRaEres",
    "model": {
        "Tcrit / K": [304.21, 126.19],
        "pcrit / Pa": [7.383e6, 3395800.0],
        "alphas": [{"type": "PR78", "acentric": 0.22394}, {"type": "PR78", "acentric":␣
↪0.0372}],
        "aresmodel": {"type": "Wilson", "m": [[0.0, -3.4768], [3.5332, 0.0]], "n": [[0.
↪0, 825], [-585, 0.0]]},
        "options": {"s": 2.0, "brule": "Quadratic", "CEoS": -0.52398}
    }
}

model = teqp.make_model(j)
for T in [223.15, 253.05, 273.08, 293.1]:
    ipure = 0

    [rhoL0, rhoV0] = model.superanc_rhoLV(T, ipure)

    rhovecL0 = np.array([0.0, 0.0]); rhovecL0[ipure] = rhoL0
    rhovecV0 = np.array([0.0, 0.0]); rhovecV0[ipure] = rhoV0

    J = model.trace_VLE_isotherm_binary(T, rhovecL0, rhovecV0)
    df = pandas.DataFrame(J)
    plt.plot(df['xL_0 / mole frac.'], df['pL / Pa']/1e6,'k')
    plt.plot(df['xV_0 / mole frac.'], df['pV / Pa']/1e6,'k')

plt.plot(1-dat['x1'], dat['p/bar']/10, 'o')
plt.plot(1-dat['y1'], dat['p/bar']/10, '^')

plt.gca().set(xlabel='$x_1$ / mole frac.', ylabel='$p$ / MPa', ylim=(0, 25))
plt.show()
```



4.5 RK-PR

The EOS can be given as

$$\alpha^{\rm r} = \psi^{(-)} - \frac{a_m}{RT}\psi^{(+)}$$

$$\psi^{(-)} = -\ln(1-b_m\rho)$$

$$\psi^{(+)} = \frac{\ln\left(\frac{\Delta_1 b_m\rho+1}{\Delta_2 b_m\rho+1}\right)}{b_m(\Delta_1-\Delta_2)}$$

with the EOS fixed constants of

$$\Delta_1 = \sum_i x_i\delta_{1,i}$$

$$\Delta_2 = \frac{1-\Delta_1}{1+\Delta_1}$$

The attractive term goes like

$$a_i = a_{c,i}\left(\frac{2}{3+T/T_{c,i}}\right)^{k_i}$$

with quadratic mixing rules

$$a_m = \sum_i \sum_j x_i x_j (1-k_{ij})\sqrt{a_i(T)a_j(T)}$$

And the covolume also gets quadratic mixing rules

$$b_m = \sum_i \sum_j x_i x_j (1-l_{ij})(b_i+b_j)/2$$

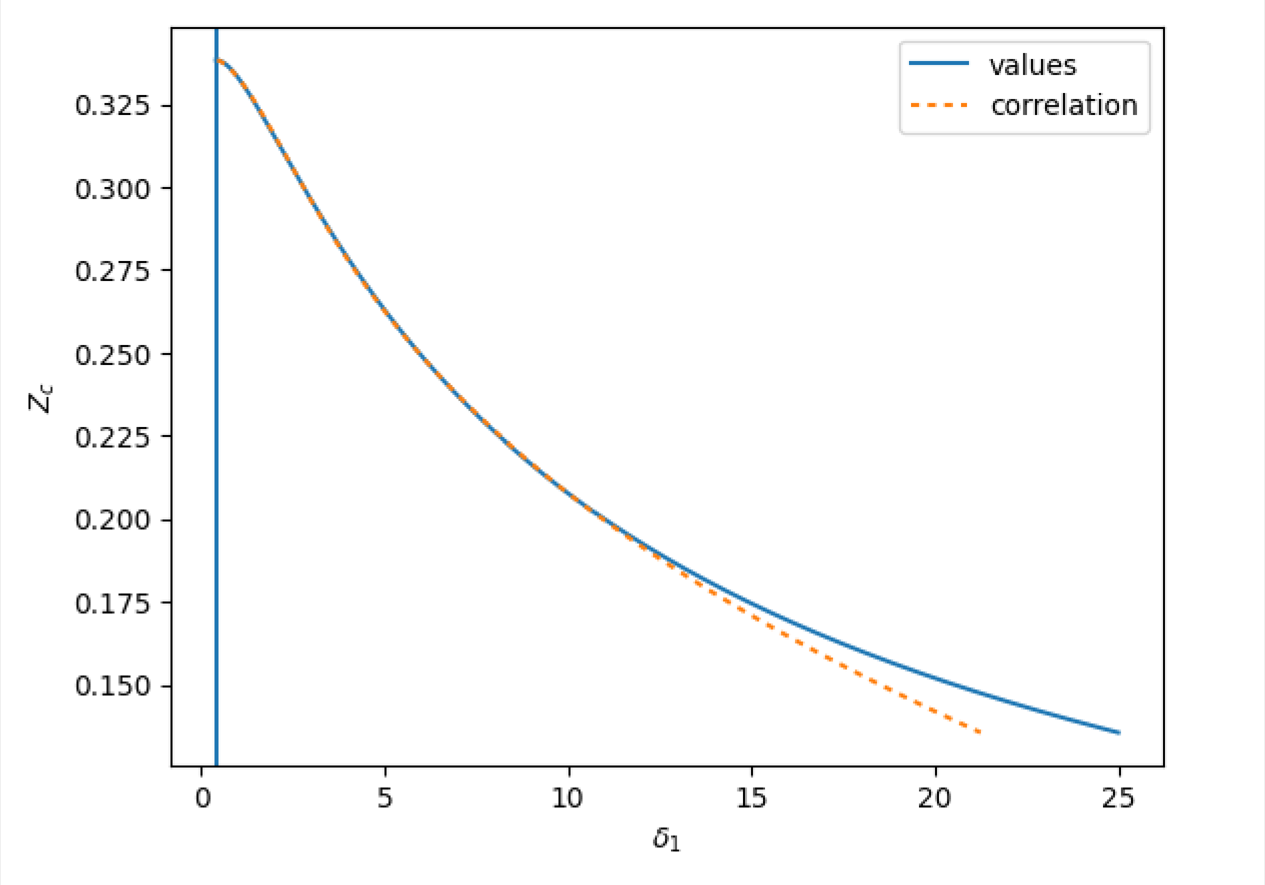
Thus, to implement the RK-PR model in predictive mode, the following steps are required:

1. Obtain the critical parameters Tc, pc
2. Solve for delta_1 from the experimental critical compressibility factor, begin with the values from the correlation
3. Solve for k by fixing the pressure at the T=0.7Tc. *In the case (e.g, CO:math:`_2`) that Tt < 0.7*Tc, use instead Tr=Tt/Tc

It may be necessary to adjust the values of $\delta_{1,i}$ and k_i for an individual component to better match the behavior of more polar components.

[1]:

```
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt
import teqp, numpy as np
import CoolProp.CoolProp as CP
import pandas

def delta1_correlation(Zc):
    # Eq. B.4 of Cismondi FPE 2005
    d1 = 0.428363
    d2 = 18.496215
    d3 = 0.338426
    d4 = 0.660000
    d5 = 789.723105
    d6 = 2.512392
    return d1 + d2*(d3-Zc)**d4 + d5*(d3-Zc)**d6

def Zc_delta1(delta1):
    # Eqs. B.1 to B.3 of Cismondi FPE 2005
    d1 = (1+delta1**2)/(1+delta1)
    y = 1 + (2*(1+delta1))**(1/3) + (4/(1+delta1))**(1/3)
    return y/(3*y + d1 - 1)

DELTA1 = np.linspace(np.sqrt(2)-1, 25, 1000)
ZZ = Zc_delta1(DELTA1)
plt.plot(DELTA1, ZZ, label='values')
DELTA1back = delta1_correlation(ZZ)
plt.axvline(np.sqrt(2)-1)
plt.plot(DELTA1back, ZZ, dashes=[2,2], label='correlation')
plt.gca().set(ylabel='$Z_c$', xlabel='$\delta_1$')
plt.legend(loc='best')
plt.show()

# for Zc in np.linspace(0.2, 0.3383, 1000):
#     resid = lambda x: Zc_delta1(x)-Zc
#     # print(resid(delta1_correlation(Zc)))
#     print(Zc, scipy.optimize.newton(resid, delta1_correlation(Zc)), delta1_
↪correlation(Zc))
```

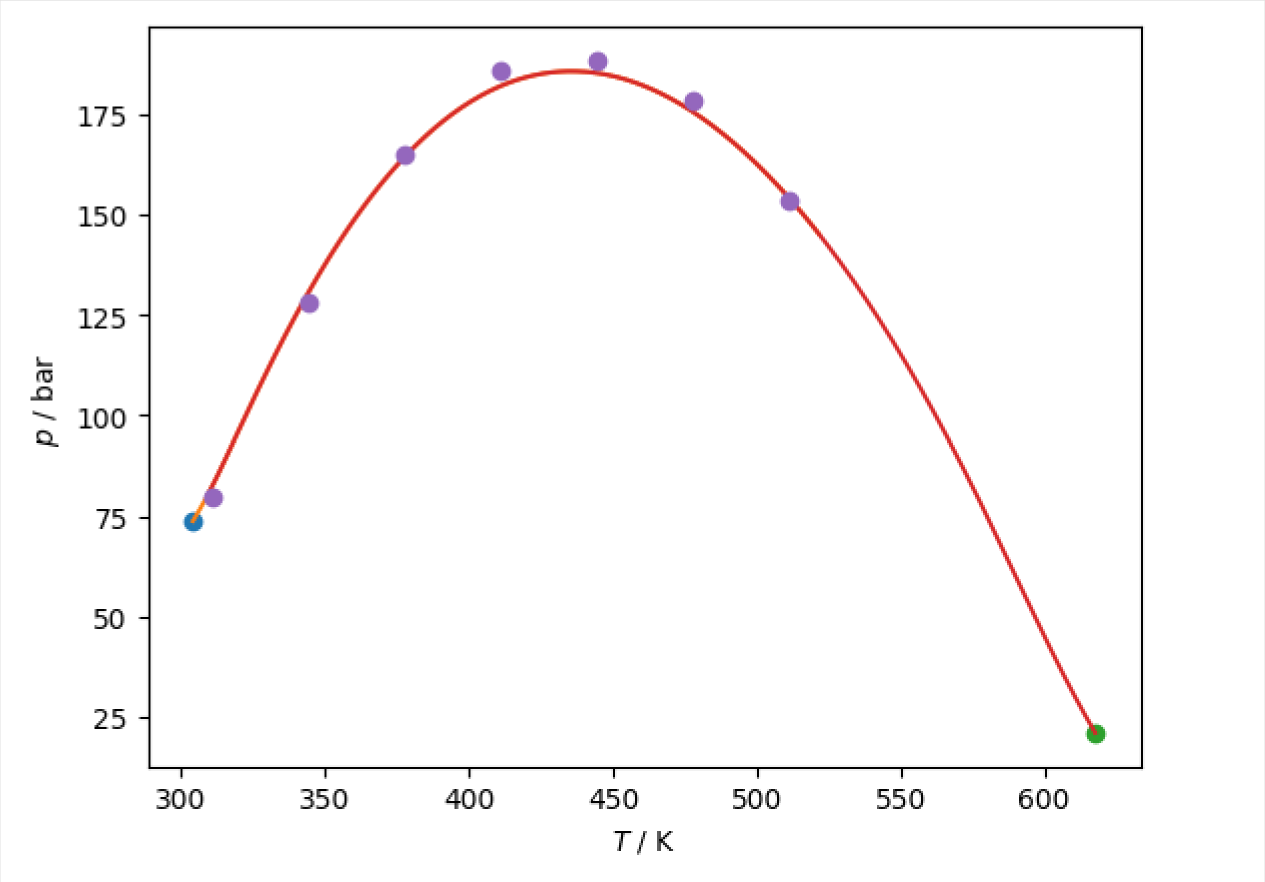


[2]:
```
names = ['CO2', 'n-Decane']

R = 8.31446261815324
Tc   = np.array([CP.PropsSI(k,"Tcrit") for k in names])
pc   = np.array([CP.PropsSI(k,"pcrit") for k in names])
rhoc = np.array([CP.PropsSI(k,"rhomolar_critical") for k in names])
Zcexp = pc/(rhoc*R*Tc)

# Use a rescaled Zc to obtain delta_1
Zc = 1.168*Zcexp

delta_1 = [scipy.optimize.newton(lambda x: Zc_delta1(x)-Zc_, delta1_correlation(Zc_))␣
↪for Zc_ in Zc]

def solve_for_k(i, p_target, Tr):
    """
    The value of k for the i-th component is based on getting
    the right vapor pressure, so a rootfinding routing is
    used to obtain these values
    """
    def objective(k):
        j = {
            "kind": "RKPRCismondi2005",
            "model": {
                "delta_1": [delta_1[i]],
                "Tcrit / K": [Tc[i]],
```

(continues on next page)

(continued from previous page)

```
                "pcrit / Pa": [pc[i]],
                "k": [k],
                "kmat": [[0.0]],
                "lmat": [[0.0]],
            }
        }
        model = teqp.make_model(j)
        T = Tr*Tc[i]
        z = np.array([1.0])
        a, b = model.get_ab(T, z)

        anc = teqp.build_ancillaries(model, Tc[i], rhoc[i], 150)
        rhoL, rhoV = model.pure_VLE_T(T, anc.rhoL(T), anc.rhoV(T), 10)
        p = T*R*rhoL*(1+model.get_Ar01(T, rhoL, z))

        return p-p_target
    return scipy.optimize.newton(objective, 2.1)

Tr = 0.7
i = 1
k_C10 = solve_for_k(i, CP.PropsSI('P','T',Tr*Tc[i],'Q',0,names[i]), Tr)

model = teqp.make_model({
    "kind": "RKPRCismondi2005",
    "model": {
        "delta_1": delta_1,
        "Tcrit / K": Tc.tolist(),
        "pcrit / Pa": pc.tolist(),
        "k": [2.23854, k_C10],
        "kmat": [[0,0],[0,0]],
        "lmat": [[0,0],[0,0]],
    }
})

# Start at both pures
for ipure in [0, 1]:
    Tc, rhoc = model.solve_pure_critical(300, 5000, {"alternative_pure_index":ipure,
↪"alternative_length": 2})
    z = np.array([0.0, 0.0]); z[ipure] = 1.0
    pc = Tc*R*rhoc*(1+model.get_Ar01(Tc, rhoc, z))
    plt.plot(Tc, pc/1e5, 'o')

    opt = teqp.TCABOptions(); opt.polish=True; opt.verbosity=100; opt.integration_
↪order=5; opt.rel_err=1e-10; opt.abs_err=1e-10
    trace = model.trace_critical_arclength_binary(Tc, z*rhoc, options=opt)
    df = pandas.DataFrame(trace)
    plt.plot(df['T / K'], df['p / Pa']/1e5)

# Overlay the data from Reamer and Sage, Cismondi additional data points not present␣
↪in Reamer and Sage
Tc_K = [310.928, 344.261, 377.594, 410.928, 444.261, 477.594, 510.928]
pc_kPa = np.array([7997.92, 12824.25, 16492.26, 18560.69, 18836.48, 17836.74, 15333.
↪94])
plt.plot(Tc_K, pc_kPa/1e2, 'o')

plt.gca().set(xlabel='$T$ / K', ylabel='$p$ / bar');
```



4.6 Cubic Plus Association (CPA)

The combination of a cubic EOS with association with the association term. The sum of the terms goes like:

$$\alpha^{\rm r} = \alpha^{\rm r}_{\rm cub} + \alpha^{\rm r}_{\rm assoc}$$

4.6.1 Cubic part

The residual contribution to α is expressed as the sum :

$$\alpha^{\rm r}_{\rm cub,rep} + \alpha^{\rm r}_{\rm cub,att}$$

where the cubic parts come from

The repulsive part of the cubic EOS contribution:

$$\alpha^{\rm r}_{\rm cub,rep} = -\ln(1 - b_{\rm mix}\rho)$$

The attractive part of the cubic EOS contribution:

$$\alpha^{\rm r}_{\rm cub,att} = -\frac{a_{\rm mix}}{RT}\frac{\ln\left(\frac{\Delta_1 b_{\rm mix}\rho+1}{\Delta_2 b_{\rm mix}\rho+1}\right)}{b_{\rm mix}\cdot(\Delta_1-\Delta_2)}$$

with the coefficients depending on the cubic type:

SRK: $\Delta_1 = 1$, $\Delta_2 = 0$

PR: $\Delta_1 = 1+\sqrt{2}$, $\Delta_2 = 1-\sqrt{2}$

The mixture models used for the $a_{\rm mix}$ and $b_{\rm mix}$ are the classical ones:

$$a_{\rm mix} = \sum_i \sum_j x_i x_j (1-k_{ij}) a_{ij}(T)$$

with x the mole fraction, k_{ij} a weighting parameter

$$a_{ij}(T) = \sqrt{a_i a_j}$$

and

$$a_i(T) = a_{0i}\left[1 + c_{1i}(1-\sqrt{T/T_{{\rm crit},i}})\right]^2$$

and for b:

$$b_{\rm mix} = \sum_i x_i b_i$$

so there are three cubic parameters per fluid that need to be obtained though fitting: b_i, a_{0i}, c_{1i}. The value of $a_{\rm ij}$ depends on temperature while $b_{\rm mix}$ does not.

4.6.2 Association part

For the association, one must have a solid understanding of the association approach that is being applied. To this end, a short discussion of the general approach is required.

[ ]:
```

```

4.7 LKP (Lee-Kesler-Plöcker)

The LKP model is a sort of hybrid between corresponding states and multiparameter EOS, simple EOS are developed for a reference fluid, and a simple fluid, and the acentric factor of the mixture is used to weight the two.

The reduced residual Helmholtz energy for the mixture is evaluated from

$$\alpha^{\rm r} = \left(1-\frac{\omega_{\rm mix}}{\omega_{\rm ref}}\right)\alpha^{\rm r}_{\rm simple} + \frac{\omega_{\rm mix}}{\omega_{\rm ref}}\alpha^{\rm r}_{\rm ref}$$

where the contributions are each of the form

$$\alpha^{\rm r}_X(\tau,\delta) = B\left(\frac{\delta}{Z_c}\right) + \frac{C}{2}\left(\frac{\delta}{Z_c}\right)^2 + \frac{D}{5}\left(\frac{\delta}{Z_c}\right)^5 - \frac{c_4\tau^3}{2\gamma}\left(\gamma\left(\frac{\delta}{Z_c}\right)^2 + \beta + 1\right)\exp\left(-\gamma\left(\frac{\delta}{Z_c}\right)^2\right) + \frac{c_4\tau^3}{2\gamma}(\beta+1)$$

where X is one of simple or reference (abbreviation: ref) with the matching sets of coefficients taken from this table:

var	simple	reference
b_1	0.1181193	0.2026579
b_2	0.265728	0.331511
b_3	0.154790	0.276550e-1
b_4	0.303230e-1	0.203488
c_1	0.236744e-1	0.313385e-1
c_2	0.186984e-1	0.503618e-1
c_3	0	0.169010e-1
c_4	0.427240e-1	0.41577e-1
d_1	0.155428e-4	0.487360e-4
d_2	0.623689e-4	0.740336e-5
β	0.653920	1.226
γ	0.601670e-1	0.03754
ω	0.0	0.3978

The terms in the contributions are given by:

$$B = b_1 - b_2\tau - b_3\tau^2 - b_4\tau^3$$

$$C = c_1 - c_2\tau + c_3\tau^3$$

$$D = d_1 + d_2\tau$$

For density, the reduced density δ is defined by

$$\delta = \frac{\rho}{\rho_{\rm red}} = v_{\rm c,mix}\rho$$

in which the reducing density is the reciprocal of the pseudo-critical volume obtained from

$$v_{\rm c,mix} = \sum_{i=1}^{N-1}\sum_{j=i+1}^{N} x_i x_j v_{ij}$$

$$v_{c,ij} = \frac{1}{8}(v_{c,i}^{1/3} + v_{c,j}^{1/3})^3$$

and the critical volumes are estimated from

$$v_{c,i} = (0.2905 - 0.085\omega_i)\frac{RT_{c,i}}{p_{c,i}}$$

For temperature, the reciprocal reduced density is defined by

$$\tau = \frac{T_{\rm c,mix}}{T}$$

with

$$T_{\rm c,mix} = \frac{1}{v_{c,mix}^{\eta}}\sum_{i=1}^{N-1}\sum_{j=i+1}^{N} x_i x_j v_{c,ij}^{\eta} T_{c,ij}$$

with $\eta = 0.25$ and

$$T_{c,ij} = k_{ij}\sqrt{T_{c,i}T_{c,j}}$$

Note: the default interaction parameter k_{ij} is therefore 1, rather than 0 in the case of SAFT and cubic models.

Finally the parameter Z_c is defined by

$$Z_c = 0.2905 - 0.085\omega_{\rm mix}$$

with the mixture acentric factor defined by

$$\omega_{\rm mix} = \sum_i x_i\omega_i$$

[1]:
```
import teqp, numpy as np
spec = {
    "Tcrit / K": [190.564, 126.192],
    "pcrit / Pa": [4.5992e6, 3.3958e6],
    "acentric": [0.011, 0.037],
    "R / J/mol/K": 8.3144598,
    "kmat": [[1.0, 0.977],[0.977, 1.0]]
}
model = teqp.make_model({'kind': 'LKP', 'model': spec}, validate=True)
```

[2]:
```
# A little sanity check, with the check value from TREND
expected = -0.18568096994998817
diff = abs(model.get_Ar00(300, 8000.1, np.array([0.8, 0.2])) - expected)
assert(diff < 1e-13)
```

4.8 Model Potentials

These EOS for model potentials are useful for understanding theory, and capture some (but perhaps not all) of the physics of “real” fluids.

[1]:
```
import teqp
teqp.__version__
```

[1]:
```
'0.19.1'
```

4.8.1 Square-well

The potential is defined by

$$V(r) = \begin{cases} \infty & r < \sigma \\ -\varepsilon & \sigma < r < \lambda\sigma \\ 0 & r > \lambda\sigma \end{cases}$$

from which an EOS can be developed by correlating results from molecular simulation. The EOS is from:

Rodolfo Espíndola-Heredia, Fernando del Río and Anatol Malijevsky Optimized equation of the state of the square-well fluid of variable range based on a fourth-order free-energy expansion J. Chem. Phys. 130, 024509 (2009); https://doi.org/10.1063/1.3054361

[2]:
```
model = teqp.make_model({
  "kind": "SW_EspindolaHeredia2009",
  "model": {
      "lambda": 1.3
  }
})
```

4.8.2 EXP-6

[3]:
```
model = teqp.make_model({
  "kind": "EXP6_Kataoka1992",
  "model": {
      "alpha": 12
  }
})
```

4.8.3 Lennard-Jones Fluid

The Lennard-Jones potential is given by

$$V(r) = 4\varepsilon\left((\sigma/r)^{12} - (\sigma/r)^{6}\right)$$

and EOS are available from many authors. teqp includes the EOS from Thol, Kolafa-Nezbeda, and Johnson.

[4]:
```
for kind, crit in [
    ["LJ126_TholJPCRD2016", (1.32, 0.31)], # Note the true critical point was not used
    ["LJ126_KolafaNezbeda1994", (1.3396, 0.3108)],
    ["LJ126_Johnson1993", (1.313, 0.310)]]:

    j = { "kind": kind, "model": {} }
    model = teqp.make_model(j)
    print(kind, model.solve_pure_critical(1.3, 0.3), crit)
```

```
LJ126_TholJPCRD2016 (1.3035125549100017, 0.3103860327864468) (1.32, 0.31)
LJ126_KolafaNezbeda1994 (1.3396478193468193, 0.3108038977722935) (1.3396, 0.3108)
LJ126_Johnson1993 (1.3130000571792173, 0.30999999768607838) (1.313, 0.31)
```

4.8.4 Two-Center Lennard-Jones Fluid

[5]:
```
model = teqp.make_model({
  'kind': '2CLJF-Dipole',
  'model': {
      "author": "2CLJF_Lisal",
      'L^*': 0.5,
      '(mu^*)^2': 0.1
  }
})
print(model.solve_pure_critical(1.3, 0.3))

model = teqp.make_model({
  'kind': '2CLJF-Quadrupole',
  'model': {
      "author": "2CLJF_Lisal",
      'L^*': 0.5,
      '(Q^*)^2': 0.1
  }
})
print(model.solve_pure_critical(1.3, 0.3))
```

```
(2.8282972062188056, 0.2005046666634018)
(2.832574303561834, 0.2003194655463274)
```

4.9 PC-SAFT

The PC-SAFT implementation in teqp is based on the implementation of Gross and Sadowski (https://doi.org/10.1021/ie0003887), with the typo from their paper fixed. It does NOT include the association contribution, only the dispersive contributions.

The model in teqp requires the user to specify the values of `sigma`, `epsilon/kB`, and `m` for each substance. A very few substances are hardcoded in teqp, for testing purposes.

The Python class is here: `PCSAFTEOS`

[1]:
```
import teqp
import numpy as np
teqp.__version__
```

[1]:
```
'0.19.1'
```

[2]:
```
TeXkey = 'Gross-IECR-2001'
ms = [1.0, 1.6069, 2.0020]
eoverk = [150.03, 191.42, 208.11]
sigmas = [3.7039, 3.5206, 3.6184]

coeffs = []
for i in range(len(ms)):
    c = teqp.SAFTCoeffs()
    c.m = ms[i]
    c.epsilon_over_k = eoverk[i]
    c.sigma_Angstrom = sigmas[i]
    coeffs.append(c)

model = teqp.PCSAFTEOS(coeffs)
```

[3]:
```
# Here are some rudimentary timing results
T = 300.0
rhovec = np.array([3.0, 4.0, 5.0])
rho = rhovec.sum()
x = rhovec/np.sum(rhovec)
%timeit model.get_fugacity_coefficients(T,rhovec)
%timeit (-1.0)*model.get_Ar20(T, rho, x)
%timeit model.get_partial_molar_volumes(T, rhovec)
```

```
4.27 µs ± 13.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
4.24 µs ± 2.63 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
19.1 µs ± 3.84 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

The model parameters can be queried:

[4]:
```
model.get_m(), model.get_epsilon_over_k_K(), model.get_sigma_Angstrom()
```

[4]:
```
(array([1.    , 1.6069, 2.002 ]),
 array([150.03, 191.42, 208.11]),
 array([3.7039, 3.5206, 3.6184]))
```

4.9.1 Adjusting k_ij

Fine-tuned values of k_{ij} can be provided when instantiating the model. A complete matrix of all the k_{ij} values must be provided. This allows for asymmetric mixing models in which $k_{ij} \neq k_{ji}$.

[5]:
```
k_01 = 0.01; k_10 = k_01
kmat = [[0,k_01,0],[k_10,0,0],[0,0,0]]
model = teqp.PCSAFTEOS(coeffs, kmat)
```

[6]:
```
# and the matrix of parameters can be printed back out
model.get_kmat()
```

[6]:
```
array([[0.  , 0.01, 0.  ],
       [0.01, 0.  , 0.  ],
       [0.  , 0.  , 0.  ]])
```

4.9.2 Superancillary

The superancillary equation for PC-SAFT has been developed, and is much more involved than that of the cubic EOS. As a consequence, the superancillary equation has been provided as a separate package rather than integrating it into to teqp to minimize the binary size of teqp. It can be installed from PYPI with: `pip install PCSAFTsuperanc`

The scaling in the superancillaries uses reduced variables:

$$\tilde{T} = T/(\epsilon/k_{\rm B})$$

$$\tilde{\rho} = \rho_{\rm N}\sigma^3$$

where $\rho_{\rm N}$ is the number density, and the other parameters are from the PC-SAFT model

[7]:
```
import PCSAFTsuperanc

sigma_m = 3e-10 # [meter]
e_over_k = 150.0 # [K]
m = 5

# The saturation temperature
T = 300

[Ttilde_crit, Ttilde_min] = PCSAFTsuperanc.get_Ttilde_crit_min(m=m)
print('Ttilde crit:', Ttilde_crit)

# Get the scaled densities for liquid and vapor phases
[tilderhoL, tilderhoV] = PCSAFTsuperanc.PCSAFTsuperanc_rhoLV(Ttilde=T/e_over_k, m=m)
# Convert back to molar densities
N_A = PCSAFTsuperanc.N_A # The value of Avogadro's constant used in superancillaries
rhoL, rhoV = [tilderho/(N_A*sigma_m**3) for tilderho in [tilderhoL, tilderhoV]]

# As a sanity check, confirm that we got the same pressure in both phases
c = teqp.SAFTCoeffs()
c.sigma_Angstrom = sigma_m*1e10
c.epsilon_over_k = e_over_k
c.m = m
model = teqp.PCSAFTEOS([c])
z = np.array([1.0])
pL = rhoL*model.get_R(z)*T*(1+model.get_Ar01(T, rhoL, z))
```

(continues on next page)

(continued from previous page)

```
pV = rhoV*model.get_R(z)*T*(1+model.get_Ar01(T, rhoV, z))
print('Pressures are:', pL, pV, 'Pa')
```

```
Ttilde crit: 2.648680568587752
Pressures are: 227809.12314460654 227809.12314409122 Pa
```

4.9.3 Maximum density

The maximum number density allowed by the EOS is defined based on the packing fraction. To get a molar density, divide by Avogadro's number. The function is conveniently exposed in Python:

[8]:

```
max_rhoN = teqp.PCSAFTEOS(coeffs).max_rhoN(130.0, np.array([0.3, 0.3, 0.4]))
display(max_rhoN)
max_rhoN/6.022e23 # the maximum molar density in mol/m^3
```

```
1.9139171771761775e+28
```

[8]:

```
31782.085306811314
```

4.9.4 Polar contributions

As of teqp version 0.15, quadrupolar and dipolar contributions have been added to the hard chain plus dispersion model which is referred to conventionally as PC-SAFT. The definitions of the reduced dipolar and quadrupolar parameters are not well documented, so they are given here. The work of Stoll, Vrabec, and Hasse (https://doi.org/10.1063/1.1623475) clearly describes the formulation of the star-scaling.

In SI units, the reduced squared dipole moment is defined by

$$(\mu^*)^2_{\rm conventional} = \frac{(\mu[Cm])^2}{4\pi\epsilon_0(\varepsilon[J])(\sigma[m])^3}$$

$$(Q^*)^2_{\rm conventional} = \frac{(\mu[Cm])^2}{4\pi\epsilon_0(\varepsilon[J])(\sigma[m])^5}$$

In the PC-SAFT formulation, the only difference is the addition of dividing the denominator by the number of segments m

$$(\mu^*)^2 = \frac{(\mu[Cm])^2}{4\pi\epsilon_0 m(\varepsilon/k_{\rm B}[K])k_B(\sigma[m])^3}$$

$$(Q^*)^2 = \frac{(Q[Cm^2])^2}{4\pi\epsilon_0 m(\varepsilon/k_B[K])k_B(\sigma[m])^5}$$

The unit conversions are obtained from

$$(\sigma[m]) = (10^{-10}m/A)(\sigma[A])$$

$$(\mu[Cm]) = (3.33564\times 10^{-30}Cm/D)(\mu[D])$$

and $\epsilon_0 = 8.85419e-12$ C^2 N^{-1} m^{-2} is the permittivity of vacuum.

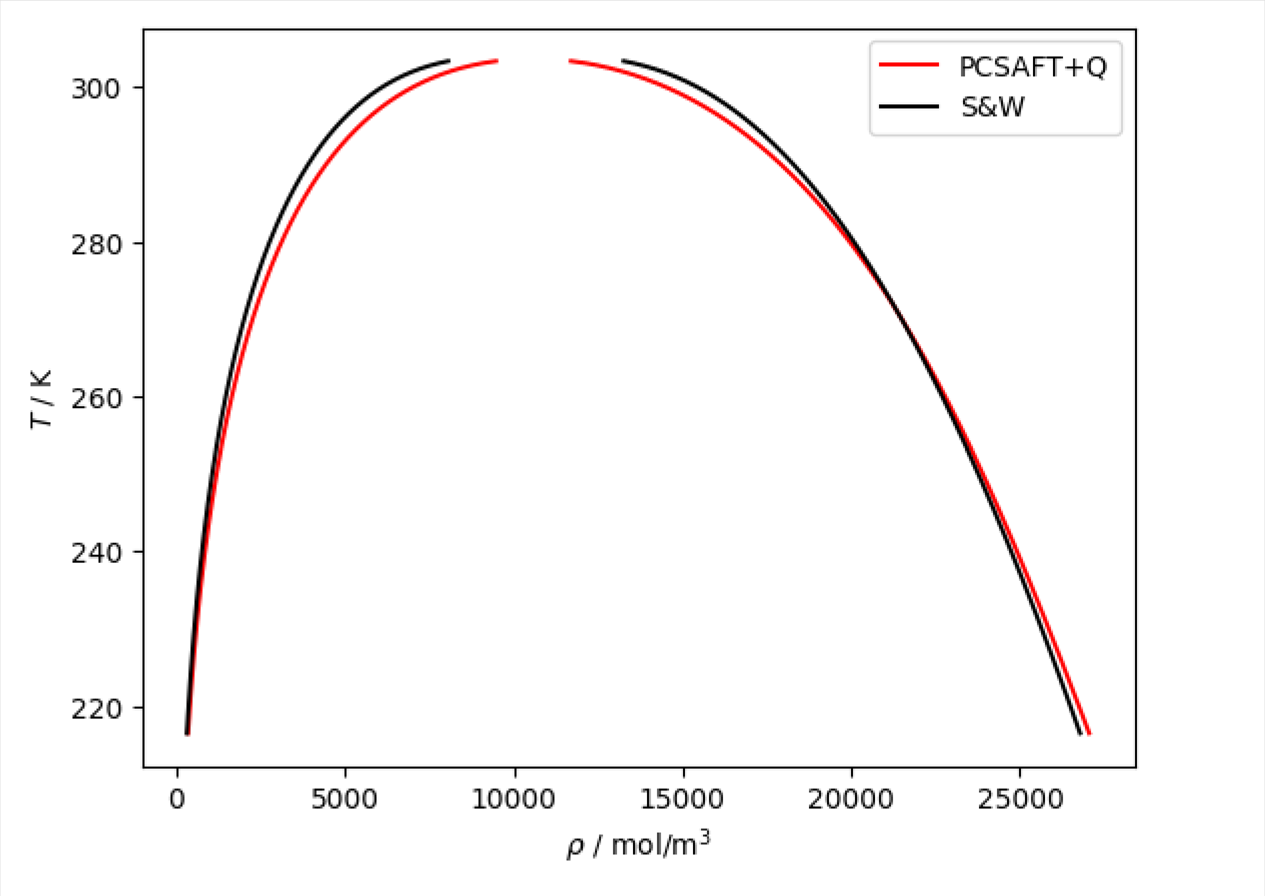
[9]:

```
# CO2 with quadrupolar contributions
j = {
    'kind': 'PCSAFT',
    'model': {
```

(continues on next page)

(continued from previous page)

```
        'coeffs': [{
            'name': 'CO2',
             'BibTeXKey': 'Gross-AICHEJ',
             'm': 1.5131,
             'sigma_Angstrom': 3.1869,
             'epsilon_over_k': 169.33,
             '(Q^*)^2': 1.26, # modified from the values in Gross and Vrabec since␣
↪the base model is different
             'nQ': 1
        }]
    }
}

model = teqp.make_model(j)
Tc, rhoc = model.solve_pure_critical(300, 11000)

T = Tc*0.999
rhoL_, rhoV_ = model.extrapolate_from_critical(Tc, rhoc, T)
rhoL, rhoV = model.pure_VLE_T(T, rhoL_, rhoV_, 10)

import CoolProp.CoolProp as CP
import matplotlib.pyplot as plt
import pandas
o = []
for T_ in np.linspace(T, 215, 1000):
    rhoL, rhoV = model.pure_VLE_T(T_, rhoL, rhoV, 10)
    try:
        o.append({
          'T': T_, 'rhoL': rhoL, 'rhoV': rhoV,
          'rhoLSW': CP.PropsSI('Dmolar','T',T_,'Q',0,'CO2'),
          'rhoVSW': CP.PropsSI('Dmolar','T',T_,'Q',1,'CO2')
        })
    except:
        pass
df = pandas.DataFrame(o)
plt.plot(df['rhoL'], df['T'], 'r', label='PCSAFT+Q')
plt.plot(df['rhoV'], df['T'], 'r')
plt.plot(df['rhoLSW'], df['T'], 'k', label='S&W')
plt.plot(df['rhoVSW'], df['T'], 'k')
plt.legend()
plt.gca().set(xlabel=r'$\rho$ / mol/m$^3$', ylabel='$T$ / K')
plt.show()
```

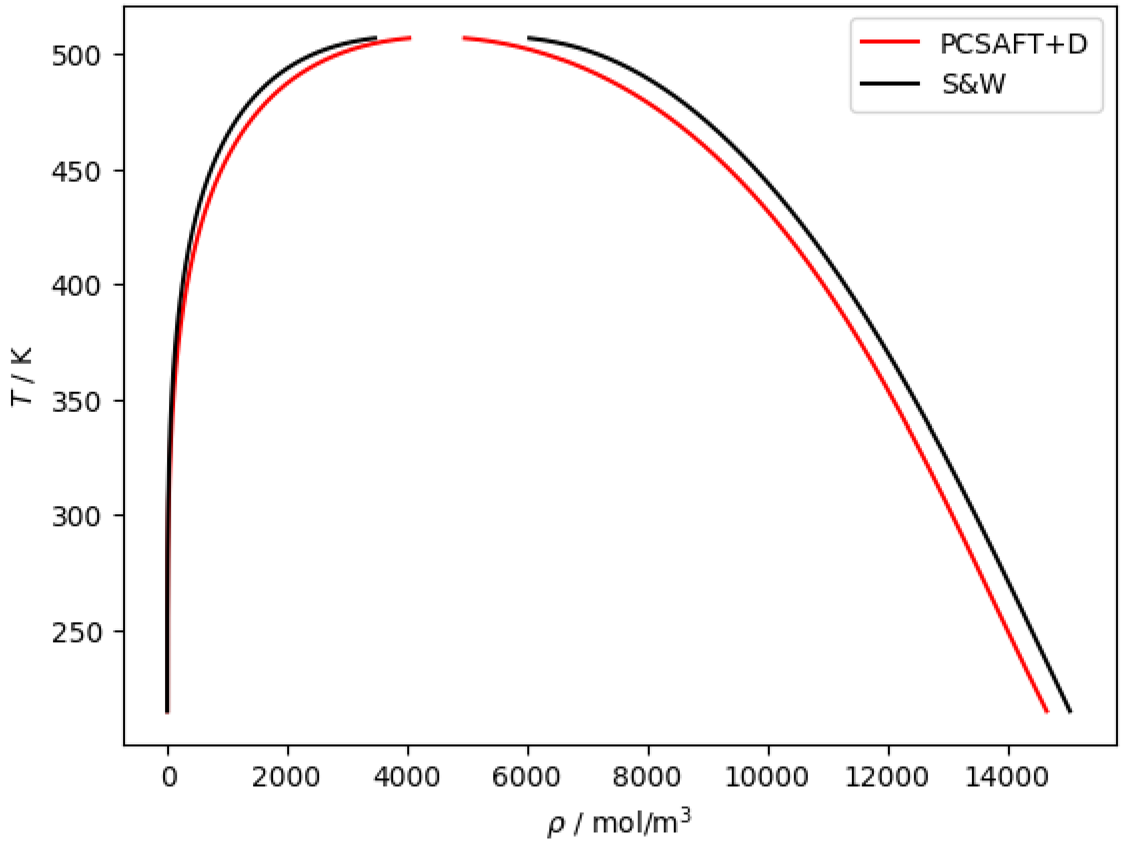


[10]:
```
# Acetone with dipolar contributions
j = {
    'kind': 'PCSAFT',
    'model': {
        'coeffs': [{
            'name': 'acetone',
             'BibTeXKey': 'Gross-IECR',
             'm': 2.7447,
             'sigma_Angstrom': 3.2742,
             'epsilon_over_k': 232.99,
             '(mu^*)^2': 1.9, # modified from the values in Gross and Vrabec since␣
↪the base model is different
             'nmu': 1
        }]
    }
}

model = teqp.make_model(j)
Tc, rhoc = model.solve_pure_critical(300, 11000)

T = Tc*0.999
rhoL_, rhoV_ = model.extrapolate_from_critical(Tc, rhoc, T)
rhoL, rhoV = model.pure_VLE_T(T, rhoL_, rhoV_, 10)

import CoolProp.CoolProp as CP
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import pandas
o = []
for T_ in np.linspace(T, 215, 1000):
    rhoL, rhoV = model.pure_VLE_T(T_, rhoL, rhoV, 10)
    try:
        o.append({
          'T': T_, 'rhoL': rhoL, 'rhoV': rhoV,
          'rhoLSW': CP.PropsSI('Dmolar','T',T_,'Q',0,'acetone'),
          'rhoVSW': CP.PropsSI('Dmolar','T',T_,'Q',1,'acetone')
        })
    except:
        pass
df = pandas.DataFrame(o)
plt.plot(df['rhoL'], df['T'], 'r', label='PCSAFT+D')
plt.plot(df['rhoV'], df['T'], 'r')
plt.plot(df['rhoLSW'], df['T'], 'k', label='S&W')
plt.plot(df['rhoVSW'], df['T'], 'k')
plt.legend()
plt.gca().set(xlabel=r'$\rho$ / mol/m$^3$', ylabel='$T$ / K')
plt.show()
```



4.10 SAFT-VR-Mie

The SAFT-VR-Mie EOS of Lafitte et al. (https://doi.org/10.1063/1.4819786) is based on the use of a Mie potential of the form

$$u(r) = C\epsilon\left((\sigma/r)^{\lambda_r} - (\sigma/r)^{\lambda_a}\right)$$

with

$$C = \frac{\lambda_r}{\lambda_r-\lambda_a}\left(\frac{\lambda_r}{\lambda_a}\right)^{\lambda_a/(\lambda_r-\lambda_a)}$$

which allows for a better representation of thermodynamic properties in general, but not always.

```
[1]: import teqp
     teqp.__version__
```

```
[1]: '0.19.1'
```

```
[2]: import numpy as np
     import pandas
     import matplotlib.pyplot as plt
     import CoolProp.CoolProp as CP
     import scipy.integrate
```

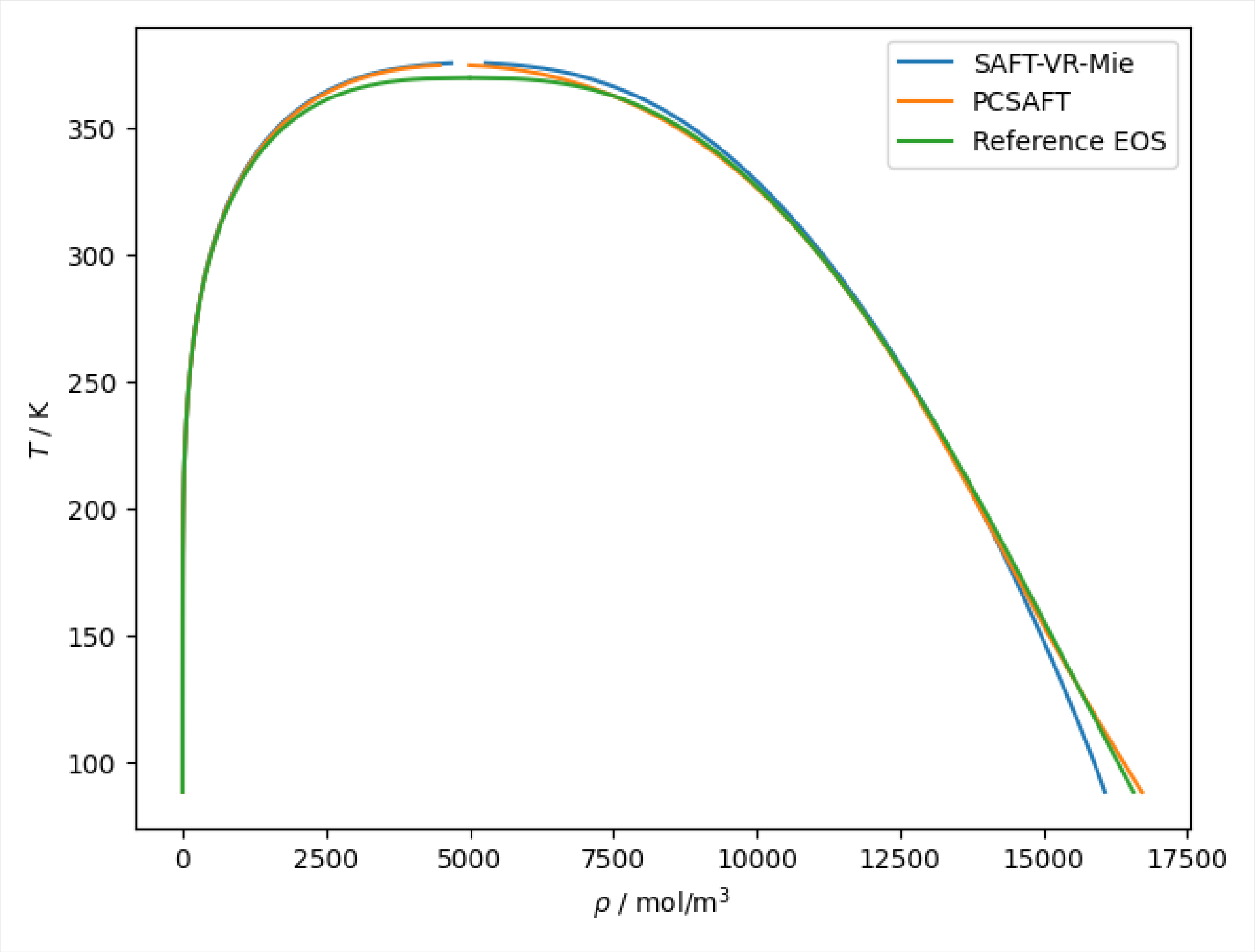
```
[3]: # Show two ways to instantiate a SAFT-VR-Mie model, the
     # first by providing the coefficients, and the second
     # by providing the name of the species. Only a very small
     # number of molecules are provided for testing, you should
     # plan on providing your own parameters.
     #
     # Show that both give the same result for the residual pressure

     z = np.array([1.0])
     model = teqp.make_model({
         "kind": 'SAFT-VR-Mie',
         "model": {
             "coeffs": [{
                 "name": "Ethane",
                 "BibTeXKey": "Lafitte",
                 "m": 1.4373,
                 "epsilon_over_k": 206.12, # [K]
                 "sigma_m": 3.7257e-10,
                 "lambda_r": 12.4,
                 "lambda_a": 6.0
             }]
         }
     })
     display(model.get_Ar01(300, 300, z))

     model = teqp.make_model({
         "kind": 'SAFT-VR-Mie',
         "model": {
             "names": ["Ethane"]
         }
     })
     display(model.get_Ar01(300, 300, z))
```

```
-0.04926724350863724
```

```
-0.04926724350863724
```

[4]:

```
# Here is an example of using teqp to trace VLE for propane
# with the default parameters of PC-SAFT and SAFT-VR-Mie
# models
for kind in ['SAFT-VR-Mie', 'PCSAFT']:
    j = {
        "kind": kind,
        "model": {
            "names": ["Propane"]
        }
    }
    model = teqp.make_model(j)

    z = np.array([1.0])
    Tc, rhoc = model.solve_pure_critical(300, 10000)

    # Extrapolate away from the critical point
    Ti = Tc*0.9997
    rhoL, rhoV = model.extrapolate_from_critical(Tc, rhoc, Ti)

    o = []
    T = Ti
    while T > 88:
        rhoL, rhoV = model.pure_VLE_T(T, rhoL, rhoV, 10)
        T -= 0.1
        o.append({'rhoL': rhoL, 'rhoV': rhoV, 'T': T})

    df = pandas.DataFrame(o)
    line, = plt.plot(df['rhoL'], df['T'], label=kind)
    plt.plot(df['rhoV'], df['T'], color=line.get_color())

# From the reference EOS of Lemmon et al. via CoolProp
name = 'Propane'
Tc = CP.PropsSI(name, 'Tcrit')
Ts = np.linspace(88, Tc, 1000)
rhoL = CP.PropsSI('Dmolar','T',Ts,'Q',0,name)
rhoV = CP.PropsSI('Dmolar','T',Ts,'Q',1,name)
line, = plt.plot(rhoL, Ts, label='Reference EOS')
plt.plot(rhoV, Ts, line.get_color())

plt.gca().set(xlabel=r'$\rho$ / mol/m$^3$', ylabel=r'$T$ / K')
plt.legend()
plt.tight_layout(pad=0.2)
plt.savefig('SAFTVRMIE_PCSAFT.pdf')
plt.show()
```



[5]:
```
# Time calculation of critical points
for kind in ['SAFT-VR-Mie', 'PCSAFT']:
    j = {
        "kind": kind,
        "model": {
            "names": ["Propane"]
        }
    }
    model = teqp.make_model(j)

    z = np.array([1.0])
    %timeit model.solve_pure_critical(300, 10000)
```

```
1.25 ms ± 6.94 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

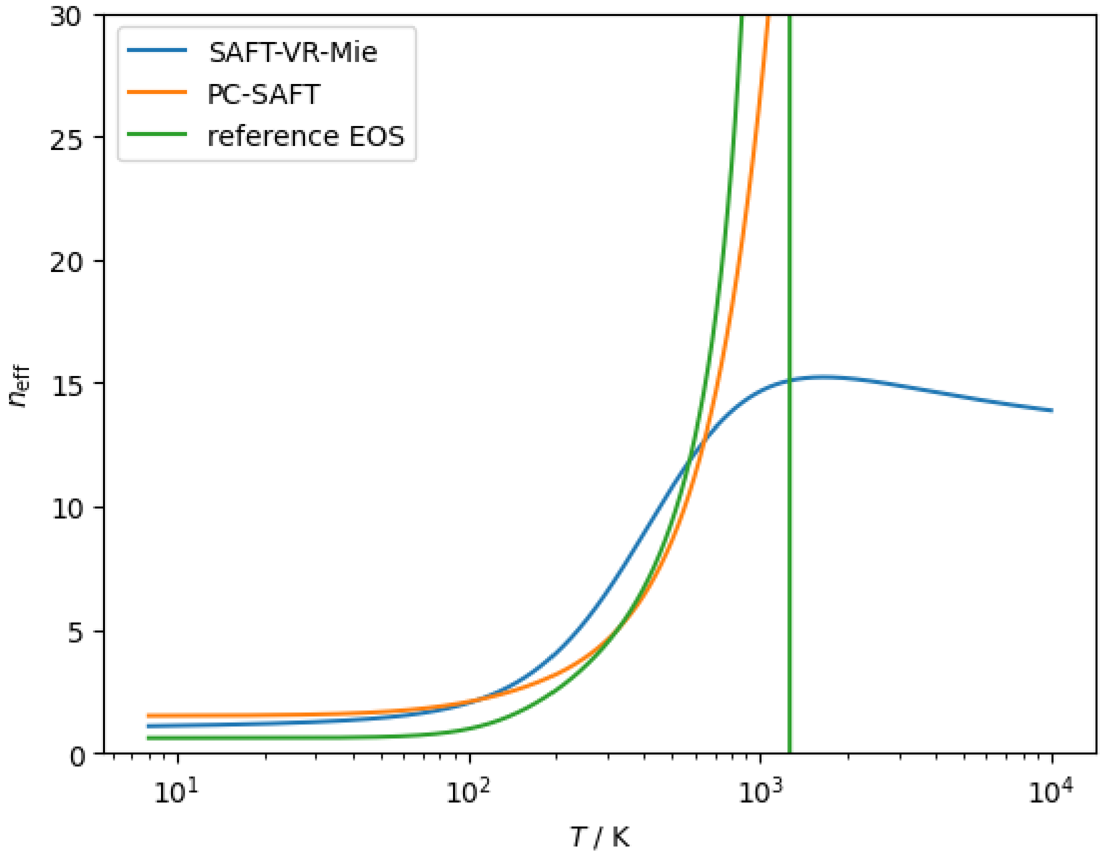
```
215 µs ± 188 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[6]:
```
# Checking the effective hardness of interaction,
# the neff parameter defined in https://doi.org/10.1063/5.0007583
# SAFT-VR-Mie comes closest to the right behavior
modelVR = teqp.make_model({
        "kind": 'SAFT-VR-Mie',
        "model": { "names": ["Methane"] }
})
```

(continues on next page)

(continued from previous page)

```
modelPCSAFT = teqp.make_model({
        "kind": 'PCSAFT',
        "model": { "names": ["Methane"] }
})
modelMF = teqp.build_multifluid_model(["Methane"], teqp.get_datapath())

for model, label in [(modelVR, 'SAFT-VR-Mie'),
                     (modelPCSAFT, 'PC-SAFT'),
                     (modelMF, 'reference EOS')]:
    z = np.array([1.0])
    rho = 1e-5
    T = np.geomspace(8, 10000, 10000)
    neff = []
    for T_ in T:
        neff.append(model.get_neff(T_, rho, z))
    plt.plot(T, neff, label=label)
plt.xscale('log')
plt.ylim(0, 30)
plt.gca().set(xlabel=r'$T$ / K', ylabel=r'$n_{\rm eff}$')
plt.legend()
plt.show()
```

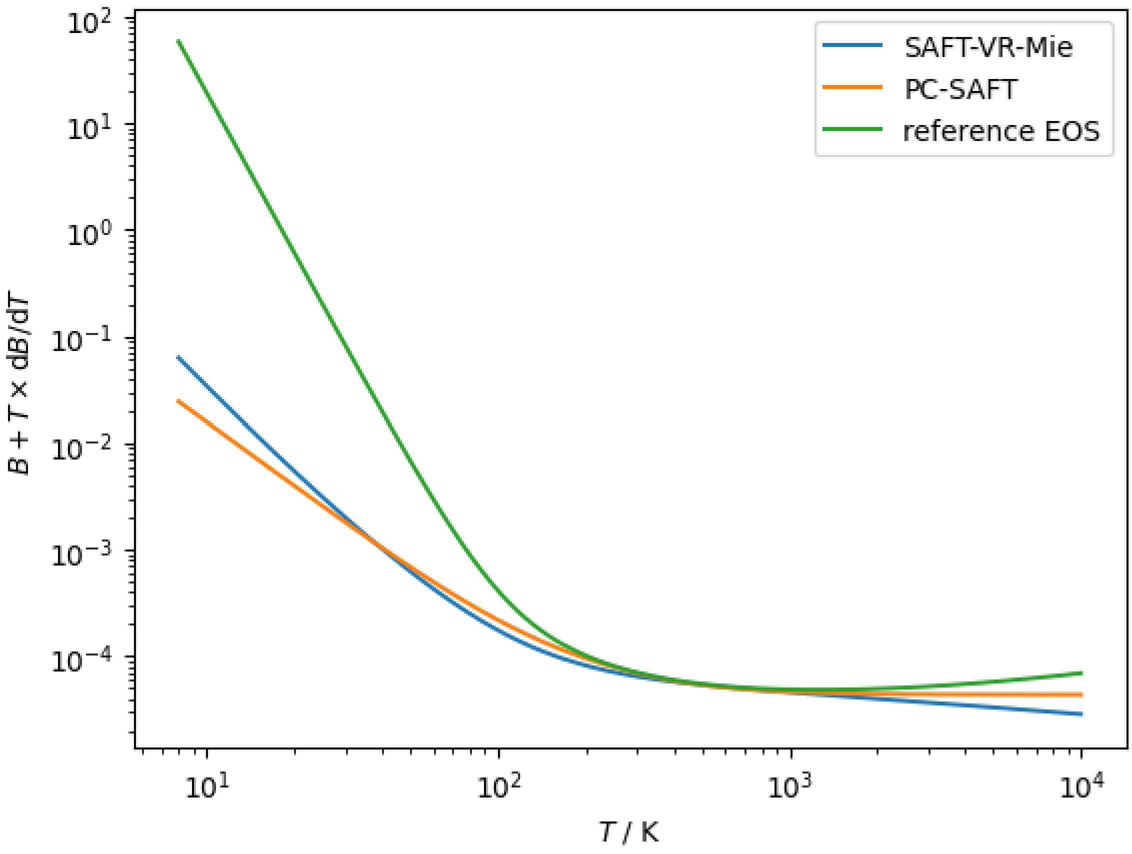


```
[7]: # Checking the temperature derivative of the virial coefficient
name = 'Methane'
modelVR = teqp.make_model({
```

(continues on next page)

(continued from previous page)

```
        "kind": 'SAFT-VR-Mie',
        "model": { "names": [name] }
})
modelPCSAFT = teqp.make_model({
        "kind": 'PCSAFT',
        "model": { "names": [name] }
})
modelMF = teqp.build_multifluid_model([name], teqp.get_datapath())

for model, label in [(modelVR, 'SAFT-VR-Mie'),
                     (modelPCSAFT, 'PC-SAFT'),
                     (modelMF, 'reference EOS')]:
    z = np.array([1.0])
    T = np.geomspace(8, 10000, 10000)
    n = 2
    B, TdBdT, thetan = [],[],[]
    for T_ in T:
        TdBdT.append(model.get_dmBnvirdTm(n, 1, T_, z)*T_)
        B.append(model.get_dmBnvirdTm(n, 0, T_, z))
        thetan.append(B[-1]+TdBdT[-1])
    plt.plot(T, thetan, label=label)
plt.xscale('log')
plt.yscale('log')
plt.gca().set(xlabel=r'$T$ / K', ylabel=r'$B+T\times$d$B$/d$T$')
plt.legend()
plt.show()
```



```
[8]: # Time model instantiation
for kind in ['SAFT-VR-Mie', 'PCSAFT']:
    j = {
        "kind": kind,
        "model": {
            "names": ["Propane"]
        }
    }
    %timeit teqp.make_model(j)
```

```
795 µs ± 2.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
331 µs ± 2.23 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

4.10.1 Calculation of diameter

The calculation of the diameter is based upon

$$d_{ii} = \int_0^{\sigma_{ii}} (1 - \exp(-\beta u_{ii}^{\rm Mie}(r)){\rm d}r$$

but the integrand is basically constant from 0 to some cutoff value of r, which we'll call $r_{\rm cut}$. So first we need to find the value of $r_{\rm cut}$ that makes the integrand take its constant value, which is explained well in the paper from Aasen (https://github.com/ClapeyronThermo/Clapeyron.jl/issues/152#issuecomment-1480324192). Finding the cutoff value is obtained when

$$\exp(-\beta u_{ii}^{\rm Mie}(r)) = EPS$$

where EPS is the numerical precision of the floating point type. Taking the logs of both sides,

$$-\beta u_{ii}^{\rm Mie} = \ln(EPS)$$

To get a starting value, it is first assumed that only the repulsive contribution contributes to the potential, yielding $u^{\rm rep} = C\epsilon(\sigma/r)^{\lambda_r}$ which yields

$$-\beta C\epsilon(\sigma/r)^{\lambda_r} = \ln(EPS)$$

and

$$(\sigma/r)_{\rm guess} = (-\ln(EPS)/(\beta C\epsilon))^{1/\lambda_r}$$

Then we solve for the residual $R(r) = 0$, where $R_0 = \exp(-u/T) - EPS$. Equivalently we can write the residual in logarithmic terms as $R = -u/T - \ln(EPS)$. This simplifies the rootfinding as you need R, R' and R'' to apply Halley's method, which are themselves quite straightforward to obtain because $R' = -u'/T$, $R'' = -u''/T$, where the primes are derivatives taken with respect to σ/r.

```
[9]: # Calculation of the residual function (needed for Halley's method)
import sympy as sy
kappa, j, lambda_r, lambda_a = sy.symbols('kappa, j, lambda_r, lambda_a')
u = kappa*(j**lambda_r - j**lambda_a)
display(sy.diff(u, j))
display(sy.simplify(sy.diff(u, j, 2)))
```

$$\kappa\left(-\frac{j^{\lambda_a}\lambda_a}{j} + \frac{j^{\lambda_r}\lambda_r}{j}\right)$$

$$\frac{\kappa\left(-j^{\lambda_a}\lambda_a^2 + j^{\lambda_a}\lambda_a + j^{\lambda_r}\lambda_r^2 - j^{\lambda_r}\lambda_r\right)}{j^2}$$

[10]:

```
# Here is a small example of using adaptive quadrature
# to obtain the quasi-exact value of d for ethane
# according to the pure-fluid parameters given in
# Lafitte et al.

epskB = 206.12 # [K]
sigma_m = 3.7257e-10 # [m]
lambda_r = 12.4
lambda_a = 6.0
C = lambda_r/(lambda_r-lambda_a)*(lambda_r/lambda_a)**(lambda_a/(lambda_r-lambda_a))
T = 300.0 # [K]

# The classical method based on adaptive quadrature
def integrand(r_m):
    u = C*epskB*((sigma_m/r_m)**(lambda_r) - (sigma_m/r_m)**(lambda_a))
    return 1.0 - np.exp(-u/T)

print('quasi-exact; (value, error estimate):')
exact, exact_error = scipy.integrate.quad(integrand, 0.0, sigma_m, epsrel=1e-16,␣
↪epsabs=1e-16)
print(exact*1e10, exact_error*1e10)

j = {"kind": 'SAFT-VR-Mie', "model": {"names": ["Ethane"]}}
model = teqp.make_model(j)
d = model.get_core_calcs(T, -1, z)["dmat"][0][0]
print('teqp; (value, error from quasi-exact in %)')
print(d, abs(d/(exact*1e10)-1)*100)
```

```
quasi-exact; (value, error estimate):
3.597838592720949 3.228005612223332e-12
teqp; (value, error from quasi-exact in %)
3.597838640613809 1.331156429529301e-06
```

4.11 SAFT-VR-Mie with polar contributions

[1]:

```
import teqp
teqp.__version__
```

[1]:

```
'0.19.1'
```

[2]:

```
import numpy as np
import matplotlib.pyplot as plt
import math
```

[3]:

```
ek = 100 # [K]
sigma_m = 3e-10

N_A = 6.022e23
fig, (ax1, ax2) = plt.subplots(2, 1)

# # From https://arxiv.org/pdf/mtrl-th/9501001.pdf which pulled from M. van Leeuwen␣
```

(continues on next page)

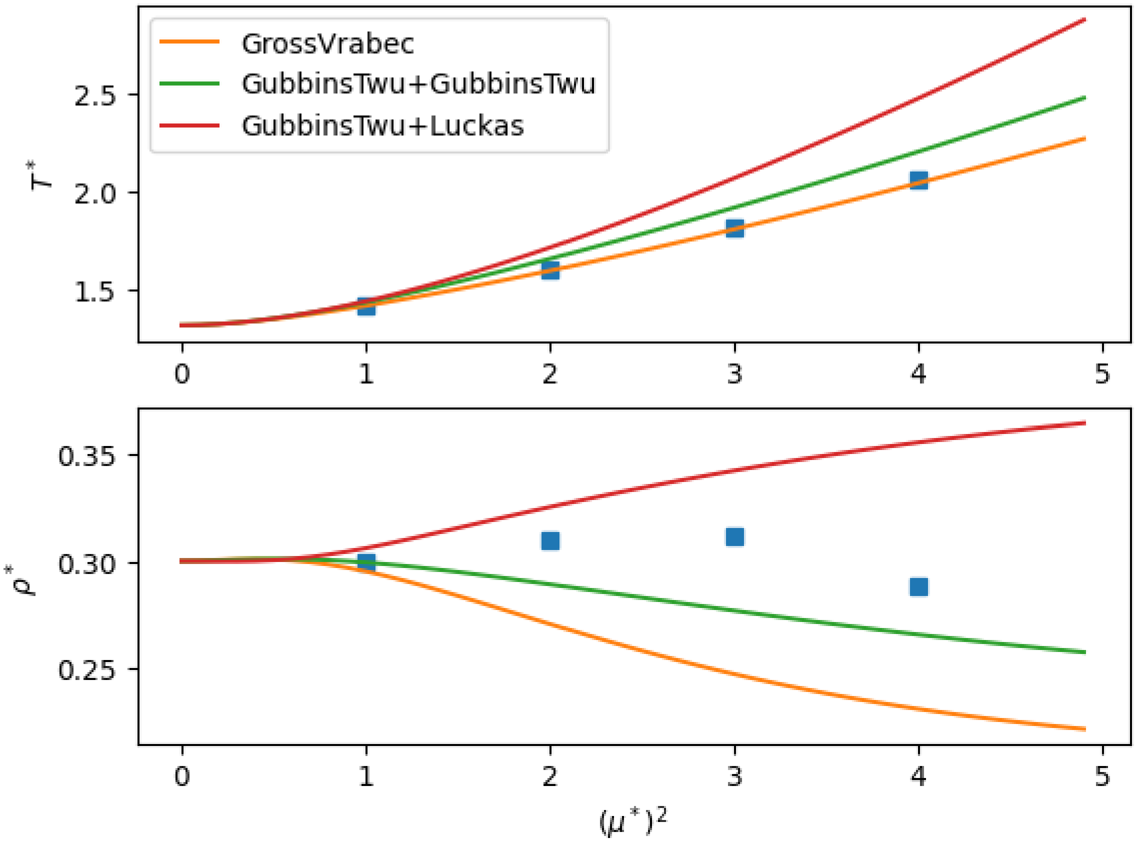
(continued from previous page)

```
↪and B. Smit, Phys. Rev. Lett. 71, 3991 (1993)
# These data need to be rescaled according to Hentschke et al. (DOI: https://doi.org/
↪10.1103/physreve.75.011506)
# mustar2 = [2.5, 3.0, 3.5, 4.0]
# T = [2.63, 3.35, 4.20, 5.07]
# rho = [0.29, 0.25, 0.24, 0.24]
# ax1.plot(mustar2, T, 'd')
# ax2.plot(mustar2, rho, 'd')

# Comparing with Hentschke, DOI: https://doi.org/10.1103/physreve.75.011506
mustar2 = [1, 2, 3, 4]
T = [1.41, 1.60, 1.82, 2.06]
rho = [0.30, 0.31, 0.312, 0.289]
ax1.plot(mustar2, T, 's')
ax2.plot(mustar2, rho, 's')

kB = 1.380649e-23 # Boltzmann's constant, J/K
epsilon_0 = 8.8541878128e-12 # Vacuum permittivity

for polar_model in ['GrossVrabec','GubbinsTwu+GubbinsTwu','GubbinsTwu+Luckas']:

    x = []; y = []; TT = []; DD = []
    rhostar_guess = 0.27
    Tstar_guess = 1.5
    for mustar2 in np.arange(0.001, 5, 0.1):
        z = np.array([1.0])
        mu2_C2m2 = 4.0*np.pi*epsilon_0*sigma_m**3*ek*kB*mustar2
        mu_Cm = mu2_C2m2**0.5
        model = teqp.make_model({
            "kind": 'SAFT-VR-Mie',
            "model": {
                "polar_model": polar_model,
                "coeffs": [{
                    "name": "Stockmayer",
                    "BibTeXKey": "me",
                    "m": 1.0,
                    "epsilon_over_k": ek, # [K]
                    "sigma_m": sigma_m,
                    "lambda_r": 12.0,
                    "lambda_a": 6.0,
                    "mu_Cm": mu_Cm,
                    "nmu": 1.0
                }]
            }
        })

        T, rho = model.solve_pure_critical(Tstar_guess*ek, rhostar_guess/(N_A*sigma_
↪m**3))
        # Store the values
        x.append(mustar2)
        TT.append(T/ek)
        DD.append(rho*N_A*sigma_m**3)
        # Update the guess for the next calculation
        Tstar_guess = TT[-1]
        rhostar_guess = DD[-1]

    ax1.plot(x, TT, label=polar_model)
```

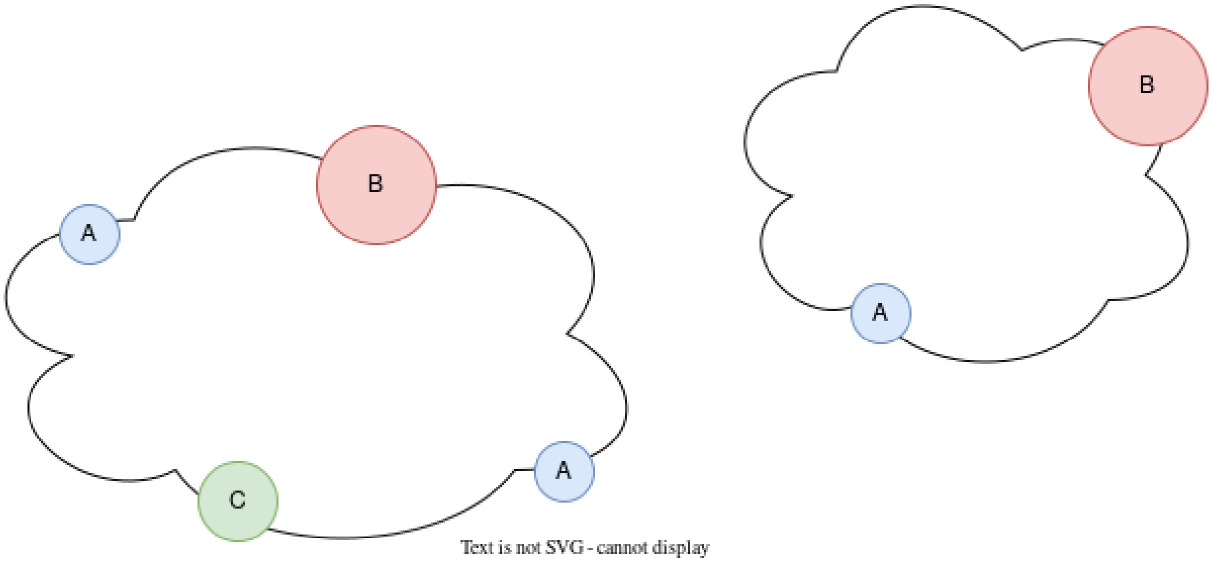
(continues on next page)

(continued from previous page)

```
    ax2.plot(x, DD)

ax1.legend(loc='best')
ax1.set(ylabel=r'$T^*$')
ax2.set(ylabel=r'$\rho^*$', xlabel=r'$(\mu^*)^2$')
plt.show()
```



4.12 Association



4.12.1 Site Interactions

Each unique site type per unique molecule is characterized by a numerical index `siteid`, which (for consistency with C++) starts with `0`. In the example above, the indices would go like:

`0`: site type A on left molecule (multiplicity of 2)

`1`: site type B on left molecule (multiplicity of 1)

`2`: site type C on left molecule (multiplicity of 1)

`3`: site type A on right molecule (multiplicity of 1)

`4`: site type B on right molecule (multiplicity of 1)

Within a molecule, the numbering of sites is arbitrary, but the mapping cannot be changed once it is defined.

Association can occur when a site can “dock” with another kind of site. In the most common kind of association used to model hydrogen bonding, there are two classes of sites, positive or negative (`e` and `H` in Clapeyron.jl). teqp allows for great flexibility in defining the site types and how they are permitted to interact with each other.

The work of Langenbach and Enders (2012) shows how to construct a counting matrix to make the successive substitution faster because not all sites are included in the summation, rather the sites within a molecule are clustered into groups, since all sites of a similar type will have the same association fractions. Thus a counting matrix **D** can be defined, with entries D_{IJ} for the pair of siteid `I` and `J` with the pseudocode

```
def get_DIJ(I, J):
    """ Return the value of an entry in the D_{IJ} matrix

    For a given unique site, look at all other sites on all other molecules
    """
    _, typei = inv_mapping[I]
    _, typej = inv_mapping[J]
    if typej in interaction_partners[typei]:
```

(continues on next page)

(continued from previous page)

```
        return counts[J]
    return 0
```

in which the dictionary `interaction_parameters` defines which sites are allowed to interact with each other. The typical alcohol+water family would be modeled with:

```
interaction_parameters = {'e': ['H'], 'H': ['e']}
```

and to follow the system considered above, we would have:

```
inv_mapping = {
    0: (0, 'A'),
    1: (0, 'B'),
    2: (0, 'C'),
    3: (1, 'A'),
    4: (1, 'B')
}
counts = [2, 1, 1, 1, 1] # multiplicities for each siteid
```

The definition of the dictionary `interaction_parameters` would depend on how you want to allow the sites to associate. Sites that are not permitted to interact with each other are removed from the D matrix (are set to zero).

The successive substitution step gives the estimated values with

$$X_{\rm step} = \frac{1}{1+\rho_N \sum_J x_J X_J D_{IJ}\Delta_{IJ}}$$

in which ρ_N is the number density (molecules per volume) of the entire mixture, Δ_{IJ} is the interaction strength (volume per site) between site with `siteid` of `I` and that with `siteid` of `J` and x_J is the mole fraction of the molecule that site `J` is found in.

Acceleration can be achieved by taking only a partial step of successive substitution, weighted by α:

$$X_{\rm new} = \alpha X_{\rm old} + (1-\alpha)X_{\rm step}$$

This is the method utilized in Langenbach and Enders.

4.12.2 Interaction strength

The interaction site strength is a matrix with side length of the number of `siteid`. It is a block matrix because practically speaking the interaction sites are still about molecule-molecule interactions

$$\Delta_{IJ} = g b_{IJ}\beta_{IJ}\left(\exp\left(\frac{\epsilon_{IJ}}{RT}\right)-1\right)/N_A$$

Reminder: b, β, and ϵ values are associated with the *molecule*, not the site.

CR1 combining rule

In the CR1 combining rule:

$$b_{IJ} = b_{ij} = \frac{b_i+b_j}{2}$$

$$\beta_{IJ} = \beta_{ij} = \sqrt{\beta_i\beta_j}$$

$$\epsilon_{IJ} = \epsilon_{ij} = \frac{\epsilon_i+\epsilon_j}{2}$$

in which i is the molecule index associated with `siteid` I and the same for j and J

4.12.3 Radial distribution function

CS: $g = \frac{2-\eta}{2(1-\eta)^2}$

KG: $g = \frac{1}{1-1.9\eta}$

where $\eta = b_{\rm mix}\rho/4$ in which ρ is density with units to match the reciprocal of $b_{\rm mix}$ (so if $b_{\rm mix}$ is mean covolume per atom, then ρ is the number density $\rho_{\rm N}$)

References:

K. Langenbach & S. Enders (2012): Cross-association of multi-component systems, Molecular Physics, 110:11-12, 1249-1260; https://dx.doi.org/10.1080/00268976.2012.668963

[1]:

```
import teqp, numpy as np
```

[2]:

```
ethanol = {
    "a0i / Pa m^6/mol^2": 0.85164,
    "bi / m^3/mol": 0.0491e-3,
    "c1": 0.7502,
    "Tc / K": 513.92,
    "epsABi / J/mol": 21500.0,
    "betaABi": 0.008,
    "sites": ["e","H"]
}
water = {
    "a0i / Pa m^6/mol^2": 0.12277,
    "bi / m^3/mol": 0.0000145,
    "c1": 0.6736,
    "Tc / K": 647.13,
    "epsABi / J/mol": 16655.0,
    "betaABi": 0.0692,
    "sites": ["e","e","H","H"]
}

jCPA = {
    "cubic": "SRK",
    "radial_dist": "CS",
#     "combining": "CR1", # No other option is implemented yet
    "pures": [ethanol, water],
    "R_gas / J/mol/K": 8.31446261815324
}

model = teqp.make_model({"kind": "CPA", "model": jCPA, "validate": False}, False)
T = 303.15 # K
rhomolar = 1/3.0680691201961814e-5 # mol/m^3
molefracs = np.array([0.3, 0.7])

# Note: passing data back and forth in JSON format is done for convenience and␣
↪flexibility, not speed
res = model.get_assoc_calcs(T, rhomolar, molefracs)

print('D:', np.array(res['D']))
print('Δ:', np.array(res['Delta']))
print('X_A:', np.array(res['X_A']))
print('siteid->(component, name):', res['to_CompSite'])
print('(component, name)->siteid:', res['to_siteid'])
print('multiplicities:', np.array(res['counts']))
```

```
D: [[0 1 0 2]
 [1 0 2 0]
 [0 1 0 2]
 [1 0 2 0]]
Δ: [[5.85623687e-27 5.85623687e-27 4.26510827e-27 4.26510827e-27]
 [5.85623687e-27 5.85623687e-27 4.26510827e-27 4.26510827e-27]
 [4.26510827e-27 4.26510827e-27 2.18581242e-27 2.18581242e-27]
 [4.26510827e-27 4.26510827e-27 2.18581242e-27 2.18581242e-27]]
X_A: [0.062584   0.062584   0.10938445 0.10938445]
siteid->(component, name): [[0, [0, 'H']], [1, [0, 'e']], [2, [1, 'H']], [3, [1, 'e
↪']]]
(component, name)->siteid: [[[0, 'H'], 0], [[0, 'e'], 1], [[1, 'H'], 2], [[1, 'e'],␣
↪3]]
multiplicities: [1 1 2 2]
```

[3]:

```
# For completeness, here is the worked Python example that was used to develop the␣
↪association implementation in teqp:

import collections
import numpy as np

class AssocClass:
    def __init__(self, molecules):

        # Get all the kinds of sites present
        mapping = {}
        counts = {}

        def sort_sites(sites):
            counts = collections.Counter(sites)
            out = []
            for k in ['B', 'P', 'N']:
                if k in counts:
                    out += [k]*counts[k]
            return out

        uid = 0
        for i, molecule in enumerate(molecules):
            for site in sort_sites(set(molecule)):
                mapping[(i, site)] = uid
                counts[uid] = molecule.count(site)
                uid += 1
        inv_mapping = {v:k for k,v in mapping.items()} # from superindex to (molecule,
↪ site pair)

        interaction_partners = {
                'B': ('N', 'P', 'B'),
                'N': ('P', 'B'),
                'P': ('N', 'B')
        }

        def get_DIJ(I, J):
            """ Return the value of an entry in the D_{IJ} matrix

            For a given unique site, look at all other sites on all other molecules
            """
            _, typei = inv_mapping[I]
```

(continues on next page)

(continued from previous page)

```
            _, typej = inv_mapping[J]
            if typej in interaction_partners[typei]:
                return counts[J]
            return 0

        Ngroups = len(mapping)
        D = np.zeros((Ngroups, Ngroups), dtype=int)
        for I in range(Ngroups):
            for J in range(Ngroups):
                D[I, J] = get_DIJ(I, J)

        # Store variables needed for later use
        self.D = D
        self.counts = counts
        self.inv_mapping = inv_mapping
        self.Ngroups = Ngroups

        # ethanol, water
        self.b_Lmol = np.array([0.0491, 0.0145])
        self.epsilon_barLmol = np.array([215.00, 166.55])
        self.beta = [8e-3, 69.2e-3]

        self.b_m3mol = self.b_Lmol/1e3
        R = 8.31446261815324 # J/(mol*K)
        self.epsilon_K = self.epsilon_barLmol*100/R # K, from (bar*L)/mol * (1e5 Pa/
↪bar) * (Pa / 1000 L), Pa*m^3 = J, then we divide by R to do [J/mol]/[J/mol/K] -> K

    def get_xJ(self, moleculemolefracs):
        """
        Return the fractions of sites within the mixture, not to be confused
        with the mole fractions of molecules within the mixture
        """
        counter = 0
        xJ = np.zeros((self.D.shape[0],))
        for J in range(self.D.shape[0]):
            j, sitej = self.inv_mapping[J] # molecule index and site name
            xJ[J] = self.counts[J]*moleculemolefracs[j]
            counter += xJ[J]
        return xJ/counter

    def get_bmix(self, molefracs):
        return (self.b_m3mol*molefracs).sum()

    def get_bij(self, i, j):
        """ CR1 """
        return (self.b_m3mol[i] + self.b_m3mol[j])/2

    def get_epsilon_k_IJ_CR1(self, *, i, j):
        """ CR1 """
        return (self.epsilon_K[i] + self.epsilon_K[j])/2

    def get_beta_IJ_CR1(self, *, i, j):
        """ CR1 """
        return (self.beta[i]*self.beta[j])**0.5

    def get_DeltaIJ(self, T, rhomolar, molefracs, *, i, j):
        b_ij = self.get_bij(i, j)
```

(continues on next page)

(continued from previous page)

```
        bmix = self.get_bmix(molefracs)
        eta = bmix*rhomolar/4 # packing fraction
        g_ij = (2-eta)/(2*(1-eta)**3)
        beta = self.get_beta_IJ_CR1(i=i,j=j) # dimensionless
        eRT = self.get_epsilon_k_IJ_CR1(i=i,j=j)/T # epsilon/(R*T), dimensionless
        return g_ij*b_ij*beta*(np.exp(eRT)-1.0) # epsilon_k_IJ is in K, beta_IJ is␣
↪dimensionless

    def get_Delta(self, T, rhomolar, *, molefracs, Ngroups):
        Delta = np.zeros((Ngroups, Ngroups))
        for I in range(Ngroups):
            i, _ = self.inv_mapping[I]
            for J in range(Ngroups):
                j, _ = self.inv_mapping[J]
                Delta[I, J] = self.get_DeltaIJ(T, rhomolar, i=i, j=j,␣
↪molefracs=molefracs)
        return Delta

    def X_iter_Langenbach(self, T:float, rhomolar:float, molefracs, init):
        """Iterate with successive substitution to obtain the non-bonded fraction of␣
↪each site

        Args:
            T (float): Temperature, K
            rhomolar (float): Molar density, mol/m^3
            molefracs (array): Mole fractions of the components
            init (array): Starting values for X_A

        Returns:
            array: non-bonded fractions for each site as one big array, indexed by␣
↪site family

        TODO: why do we need mole fractions here and site fractions elsewhere?
        """
        # xJ = np.array(self.get_xJ(moleculemolefracs=molefracs), ndmin=2) # row␣
↪vector

        XXJ = np.array([ molefracs[self.inv_mapping[J][0]] for J in range(self.
↪Ngroups)])
        N_A = 6.02214076e23 # [1/mol]
        Delta = self.get_Delta(T, rhomolar, Ngroups=self.Ngroups,␣
↪molefracs=molefracs)/N_A
        rhoN = rhomolar*N_A # number density, in 1/m^3
        Y = np.array(init[:], ndmin=2) # copy, row vector

        DD = self.D*Delta # coefficient-wise product
        DDX = XXJ*DD # coefficient-wise product

        for _ in range(30):
            # The naive treatment
            summer = 0.0
            for J in range(self.Ngroups):
                summer += Y[0,J]*XXJ[J]*self.D[:,J]*Delta[:,J]
            # Optimized treatment
            summer2 = (DDX@Y.T).squeeze()
            # print(summer, summer2)
            term = rhoN*summer2
```

(continues on next page)

(continued from previous page)

```
            Y = 0.5*(Y+1/(1+term))

        return Y

    def X_A_pure_Langenbach(self, i:int, T:float, rhomolar:float):
        """Calculate the association fractions for a pure fluid
        based upon the method of Eq. 20, from
        Langenbach & Enders, Mol. Phys.
        URL: https://www.tandfonline.com/doi/abs/10.1080/00268976.2012.668963

        Args:
            int (int): Index of the pure fluid
            T (float): Temperature, K
            rhomolar (float): Molar density, mol/m^3
            molefracs (_type_): Molar fractions, array

        TODO: why do we need site fractions here and mole fractions elsewhere?
        """
        molefracs = [0]*len(self.b_m3mol)
        molefracs[i] = 1
        xJ = self.get_xJ(moleculemolefracs=molefracs)
        N_A = 6.02214076e23 # [1/mol]
        Delta = self.get_Delta(T, rhomolar, Ngroups=self.Ngroups,␣
↪molefracs=molefracs)/N_A
        common = np.array(2*rhomolar*N_A*(xJ@self.D@Delta), ndmin=2).sum(axis=0)
        return (np.sqrt(1+2*common)-1)/common

    def X_A_pure_HuangRadosz(self, *, i:int, T:float, rhomolar:float, klass:str):
        """Use the explicit solutions from Huang and Radosz to obtain the
        association fraction for a pure fluid

        Args:
            i (int): The fluid index for which the method is being applied
            T (float): Temperature, K
            rhomolar (float): Molar density, in mol/m^3
            klass (str): Association class, one in {'2B','3B','4C'}

        Returns:
            float: value of X_A
        """

        b_ij = b_cubic = self.get_bij(i=i,j=i)
        betaABi = self.get_beta_IJ_CR1(i=i,j=i)
        R = 8.31446261815324
        RT = R*T
        epsABi = self.get_epsilon_k_IJ_CR1(i=i,j=i)*R # To get J/mol

        eta = b_ij*rhomolar/4 # packing fraction
        g_vm_ref = (2-eta)/(2*(1-eta)**3)
        DeltaAiBj = g_vm_ref*(np.exp(epsABi/RT) - 1.0)*b_cubic* betaABi

        if klass == '2B':
            X_A = (-1.0 + (1.0 + 4.0 * rhomolar * DeltaAiBj)**0.5) / (2.0 * rhomolar␣
↪* DeltaAiBj)
        elif klass == '3B':
            X_A = ((-(1.0 - rhomolar * DeltaAiBj) + np.sqrt((1.0 + rhomolar *␣
↪DeltaAiBj)**2 + 4.0 * rhomolar * DeltaAiBj)) / (4.0 * rhomolar * DeltaAiBj))
```

(continues on next page)

(continued from previous page)

```
        elif klass == '4C':
            X_A = (-1.0 + np.sqrt(1.0 + 8.0 * rhomolar * DeltaAiBj)) / (4.0 *␣
↪rhomolar * DeltaAiBj)

        return X_A

if __name__ == '__main__':
    a = AssocClass([('B'), ('P','N','N'), ('P')])
    assert(a.D.tolist() == [[1, 1, 2, 1], [1, 0, 2, 0], [1, 1, 0, 1], [1, 0, 2, 0]])

    #### 4C water
    a = AssocClass([(), ('P','P','N','N')])
    T = 303.15
    rhomolar = 1/1.7915123921401366e-5
    X_A_Clapeyron = 0.07920738195861185 # version 0.5.9
    X_A_HR = a.X_A_pure_HuangRadosz(i=1, T=T, rhomolar=rhomolar, klass='4C')
    X_A_La = a.X_A_pure_Langenbach(i=1, T=T, rhomolar=rhomolar)[0]
    assert(abs(X_A_HR - X_A_Clapeyron) < 1e-10)
    assert(abs(X_A_La - X_A_Clapeyron) < 1e-10)
    a.X_iter_Langenbach(T=T, rhomolar=rhomolar, molefracs=[0,1], init=np.array([1.0,␣
↪1.0]))

    ### 2B ethanol
    a = AssocClass([('P','N'), ()])
    T = 303.15
    rhomolar = 1/1.7915123921401366e-5
    X_A_Clapeyron = 0.020464699705843845 # version 0.5.9
    X_A_HR = a.X_A_pure_HuangRadosz(i=0, T=T, rhomolar=rhomolar, klass='2B')
    X_A_La = a.X_A_pure_Langenbach(i=0, T=T, rhomolar=rhomolar)[0]
    assert(abs(X_A_HR - X_A_Clapeyron) < 1e-10)
    assert(abs(X_A_La - X_A_Clapeyron) < 1e-10)
    a.X_iter_Langenbach(T=T, rhomolar=rhomolar, molefracs=[1,0], init=np.array([1.0,␣
↪1.0]))

    a = AssocClass([('P','N'), ('P','P','N','N')])
    T = 303.15
    print(a.D)
    rhomolar = 1/3.0680691201961814e-5
    print(a.get_Delta(T, rhomolar, molefracs=[0.3, 0.7], Ngroups=4)/6.02214076e23)
    print(a.X_iter_Langenbach(T=T, rhomolar=rhomolar, molefracs=[0.3,0.7], init=np.
↪array([1.0, 1.0, 1, 1])))
```

```
[[0 1 0 2]
 [1 0 2 0]
 [0 1 0 2]
 [1 0 2 0]]
[[5.85623687e-27 5.85623687e-27 4.26510827e-27 4.26510827e-27]
 [5.85623687e-27 5.85623687e-27 4.26510827e-27 4.26510827e-27]
 [4.26510827e-27 4.26510827e-27 2.18581242e-27 2.18581242e-27]
 [4.26510827e-27 4.26510827e-27 2.18581242e-27 2.18581242e-27]]
[[0.062584   0.062584   0.10938445 0.10938445]]
```

4.13 Multi-fluid EOS

Peering into the innards of teqp

[1]:
```
import timeit, json
import pandas
import numpy as np
import teqp
teqp.__version__
```

[1]:
```
'0.19.1'
```

4.13.1 Ancillary Equations

Ancillary equations are provided along with multiparameter equations of state. The give a good *approximation* to the phase equilibrium densities. There are routines in teqp to use the ancillary equations provided with the EOS. First a class containing the ancillary equations is obtained, then methods on that class are called

[2]:
```
model = teqp.build_multifluid_model(["Methane"], teqp.get_datapath())
anc = model.build_ancillaries()
T = 100.0 # [K]
rhoL, rhoV = anc.rhoL(T), anc.rhoV(T)
print('Densities are:', rhoL, rhoV, 'mol/m^3')
```

```
Densities are: 27357.335621492966 42.04100696197727 mol/m^3
```

But those densities do not correspond to the *true* phase equilibrium solution, so we need to polish the solution:

[3]:
```
Niter = 10
rhoLtrue, rhoVtrue = model.pure_VLE_T(T, rhoL, rhoV, Niter)
print('VLE densities are:', rhoLtrue, rhoVtrue, 'mol/m^3')
```

```
VLE densities are: 27357.147019094467 42.047982278351704 mol/m^3
```

And looking the densities, they are slightly different after the phase equilibrium calculation

4.13.2 Ammonia-Water

Tillner-Roth and Friend provided a hard-coded model that is in a form not compatible with the other multi-fluid models. It is available via the high-level factory function

[4]:
```
AW = teqp.AmmoniaWaterTillnerRoth()
AW.get_Ar01(300, 300, np.array([0.9, 0.0]))
```

[4]:
```
-0.09731055757504622
```

4.13.3 Pure fluid loading

```
[5]: # By default teqp looks for fluids relative to the set of fluids in ROOT/dev/fluids
     # The name (case-sensitive) should match the .json file, without the json extension.
     %timeit model = teqp.build_multifluid_model(["Methane"], teqp.get_datapath())
```

```
32.6 ms ± 111 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[6]: # And if you provide valid aliases, alias lookup will be used to resolve the name
     # But beware, this is rather a lot slower than the above because all fluid files need␣
     ↪to be read
     # in to build the alias map
     %timeit model = teqp.build_multifluid_model(["n-C1H4"], teqp.get_datapath())
```

```
32.6 ms ± 164 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

So, how to make it faster? Only do it once and cache

```
[7]: # Here is the set of possible aliases to absolute paths of files
     # Building this map takes a little while (somewhat faster in C++) due to all the file␣
     ↪reads
     # If you know your files will not change, good idea to build this alias map yourself.
     %timeit aliasmap = teqp.build_alias_map(teqp.get_datapath())
     aliasmap = teqp.build_alias_map(teqp.get_datapath())
     list(aliasmap.keys())[0:10] # the first 10 aliases in the dict
```

```
32.1 ms ± 332 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
[7]: ['1,2-DICHLOROETHANE',
      '1,2-dichloroethane',
      '1-BUTENE',
      '1-Butene',
      '100-41-4',
      '10024-97-2',
      '102687-65-0',
      '106-42-3',
      '106-97-8',
      '106-98-9']
```

```
[8]: # Then load the absolute paths from the alias map,
     # which will guarantee that you hit exactly what you were looking for,
     # resolving aliases as needed
     identifiers = [aliasmap[n] for n in ["n-C1H4"]]
     %timeit model = teqp.build_multifluid_model(identifiers, teqp.get_datapath())
```

```
542 µs ± 3.18 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

At some point soon teqp will support in-memory loading of JSON data for the pure components, without requiring reads from the operating system

```
[9]: # And you can also load the JSON that teqp is loading for the pure fluids
     pureJSON = teqp.collect_component_json(['Neon','Hydrogen'], teqp.get_datapath())
```

4.13.4 Mixture model loading

```
[10]: # Load the default JSON for the binary interaction parameters
      BIP = json.load(open(teqp.get_datapath()+'/dev/mixtures/mixture_binary_pairs.json'))
```

```
[11]: # You can obtain interaction parameters either by pairs of names, where name is the␣
      ↪name that teqp uses, the ["INFO"]["NAME"] field
      params, swap_needed = teqp.get_BIPdep(BIP, ['Methane','Ethane'])
      params
```

```
[11]: {'BibTeX': 'Kunz-JCED-2012',
       'CAS1': '74-82-8',
       'CAS2': '74-84-0',
       'F': 1.0,
       'Name1': 'Methane',
       'Name2': 'Ethane',
       'betaT': 0.996336508,
       'betaV': 0.997547866,
       'function': 'Methane-Ethane',
       'gammaT': 1.049707697,
       'gammaV': 1.006617867}
```

```
[12]: # Or also by CAS#
      params, swap_needed = teqp.get_BIPdep(BIP, ['74-82-8','74-84-0'])
      params
```

```
[12]: {'BibTeX': 'Kunz-JCED-2012',
       'CAS1': '74-82-8',
       'CAS2': '74-84-0',
       'F': 1.0,
       'Name1': 'Methane',
       'Name2': 'Ethane',
       'betaT': 0.996336508,
       'betaV': 0.997547866,
       'function': 'Methane-Ethane',
       'gammaT': 1.049707697,
       'gammaV': 1.006617867}
```

```
[13]: # But mixing is not allowed
      params, swap_needed = teqp.get_BIPdep(BIP, ['74-82-8','Ethane'])
      params
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[13], line 2
      1 # But mixing is not allowed
----> 2 params, swap_needed = teqp.get_BIPdep(BIP, ['74-82-8','Ethane'])
      3 params

ValueError: Can't match the binary pair for: 74-82-8/Ethane
```

4.13.5 Estimation of interaction parameters

Estimation of interaction parameters can be used when no mixture model is present. The `flags` keyword argument allows the user to control how estimation is applied. The `flags` keyword argument should be a dictionary, with keys of `"estimate"` to provide the desired estimation scheme as-needed. For now, the only allowed estimation scheme is `Lorentz-Berthelot`.

If it is desired to force the estimation, the `"force-estimate"` to force the use of the provided estimation scheme for all binaries, even when a proper mixture model is available. The value associated with `"force-estimate"` is ignored.

```
[14]: params, swap_needed = teqp.get_BIPdep(BIP, ['74-82-8','74-84-0'], flags={'force-
      ↪estimate':'yes', 'estimate': 'Lorentz-Berthelot'})
      params
```

```
[14]: {'F': 0.0, 'betaT': 1.0, 'betaV': 1.0, 'gammaT': 1.0, 'gammaV': 1.0}
```

```
[15]: # And without the force, the forcing is ignored
      params, swap_needed = teqp.get_BIPdep(BIP, ['74-82-8','74-84-0'], flags={'estimate':
      ↪'Lorentz-Berthelot'})
      params
```

```
[15]: {'BibTeX': 'Kunz-JCED-2012',
       'CAS1': '74-82-8',
       'CAS2': '74-84-0',
       'F': 1.0,
       'Name1': 'Methane',
       'Name2': 'Ethane',
       'betaT': 0.996336508,
       'betaV': 0.997547866,
       'function': 'Methane-Ethane',
       'gammaT': 1.049707697,
       'gammaV': 1.006617867}
```

```
[16]: # And the same flags can be passed to the multifluid model constructor
      model = teqp.build_multifluid_model(
          ['74-82-8','74-84-0'],
          teqp.get_datapath(),
          flags={'force-estimate':'yes', 'estimate': 'Lorentz-Berthelot'})
```

4.14 Multfluid mutant

These adapted multifluid models are used for fitting departure functions. The pure fluids remain fixed while you can adjust the mixture model, both the interaction parameters as well as the departure function terms

```
[1]: import teqp, numpy as np
     teqp.__version__
```

```
[1]: '0.19.1'
```

```
[2]: basemodel = teqp.build_multifluid_model(['Nitrogen','Ethane'], teqp.get_datapath())
     s = {
         "0": {
             "1": {
                 "BIP": {
```

(continues on next page)

(continued from previous page)

```
                "betaT": 1.1,
                "gammaT": 0.9,
                "betaV": 1.05,
                "gammaV": 1.3,
                "Fij": 1.0
            },
            "departure":{
                "type": "none"
            }
        }
    }
}
mutant = teqp.build_multifluid_mutant(basemodel, s)
```

[3]:
```
%timeit teqp.build_multifluid_mutant(basemodel, s)
```

```
25 µs ± 1.9 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[4]:
```
mutant.get_Ar01(300, 3.0, np.array([0.5, 0.5]))
```

[4]:
```
-0.00017517184039893556
```

4.15 REFPROP <10.0 conversion

As of `teqp` version 0.19.0, it is possible to read in the .FLD and HMX.BNC of NIST REFPROP 10.0 and load them into `teqp` multifluid models. There are two approaches; either you can pass paths to the files of interest, or you can load them into JSON once, and pass the converted JSON back into teqp's `make_model` function.

The conversion code is uses that of REFPROP-interop and the fluid file format of CoolProp is used.

The example is based on the interaction parameters provided in the supporting information of the paper Mixture Model for Refrigerant Pairs R-32/1234yf, R-32/1234ze(E), R-1234ze(E)/227ea, R-1234yf/152a, and R-125/1234yf by Ian Bell

[1]:
```
import json
import teqp
teqp.__version__
```

[1]:
```
'0.19.1'
```

[2]:
```
# The first approach, we just pass paths to the files, they live in the folder
# containing this notebook, and teqp does the conversion on the fly
jsimple = {
    'kind': 'multifluid',
    'model': {
        'HMX.BNC': 'HMX.BNC',
        'components': ['R152A.FLD', 'NEWR1234YF.FLD'],
    }
}
model = teqp.make_model(jsimple)
```

[3]:
```
%timeit teqp.make_model(jsimple)
```

```
45.8 ms ± 89.5 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[4]:
```
# Convert each of the FLD files to JSON
FLD0 = teqp.convert_FLD('R152A.FLD', name='R152A')
FLD1 = teqp.convert_FLD('NEWR1234YF.FLD', name='R1234YF')
BIP, DEP = teqp.convert_HMXBNC('HMX.BNC')
```

[5]:
```
jconverted = {
    "kind": "multifluid",
    "model": {
        "components": [FLD0, FLD1],
        "BIP": BIP,
        "departure": DEP
    }
}
model = teqp.make_model(jconverted)
```

[6]:
```
%timeit teqp.make_model(jconverted)
```

```
608 µs ± 2.55 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

From this example you can note that the first method is a lot slower because the FLD->JSON conversion needs to happen for each call, while in the second method it is much faster because only the JSON parsing needs to be done in `teqp`.

[7]:
```
# It is also possible to prefix the path to indicate that the
# indicated file (after the FLD::) should be converted from REFPROP format
jconverted = {
    "kind": "multifluid",
    "model": {
        "components": ["FLDPATH::R152A.FLD", 'FLDPATH::NEWR1234YF.FLD'],
        "BIP": BIP,
        "departure": DEP
    }
}
model = teqp.make_model(jconverted)
```

[8]:
```
%timeit teqp.make_model(jconverted)
```

```
41.4 ms ± 98.1 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

4.16 GERG

In the GERG-2004 and GERG-2008 models, the pure fluids are modeled with high-accuracy multiparameter EOS. The model is covered exhaustively in the GERG-2004 monograph: https://www.gerg.eu/wp-content/uploads/2019/10/TM15.pdf and in the GERG-2008 paper: https://doi.org/10.1021/je300655b

The following components are supported (case-sensitive) in GERG-2004:

- methane
- nitrogen
- carbondioxide
- ethane
- propane
- n-butane

- isobutane
- n-pentane
- isopentane
- n-hexane
- n-heptane
- n-octane
- hydrogen
- oxygen
- carbonmonoxide
- water
- helium
- argon

and GERG-2008 adds the components:

- hydrogensulfide
- n-nonane
- n-decane

(as well as modifying the pure component EOS for carbon monoxide and isopentane).

The interaction parameters and departure functions are not editable (by design) and the EOS parameters are hard-coded. No ancillary equations are available along with the GERG-2004 model, but you can use the on-the-fly ancillary generator of teqp.

The residual portions of these models were added in version 0.18.0, and it is planned to add the ideal-gas portions as well at a later date. The residual portion is enough for many applications like phase equilibria and critical locus tracing.

The kind is 'GERG2004resid' for the GERG-2004 residual model and 'GERG2008resid' for the GERG-2008 residual model

[1]:

```
import teqp
import numpy as np
import pandas
import matplotlib.pyplot as plt

teqp.__version__
```

[1]:

```
'0.19.1'
```

[2]:

```
model = teqp.make_model({'kind':"GERG2004resid", 'model':{"names": ['methane','ethane
↪']}})
```

[3]:

```
# Note that names are case-sensitive; this doesn't work
model = teqp.make_model({'kind':"GERG2004resid", 'model':{"names": ['MeThAnE','ethane
↪']}})
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[3], line 2
      1 # Note that names are case-sensitive; this doesn't work
----> 2 model =␣
```

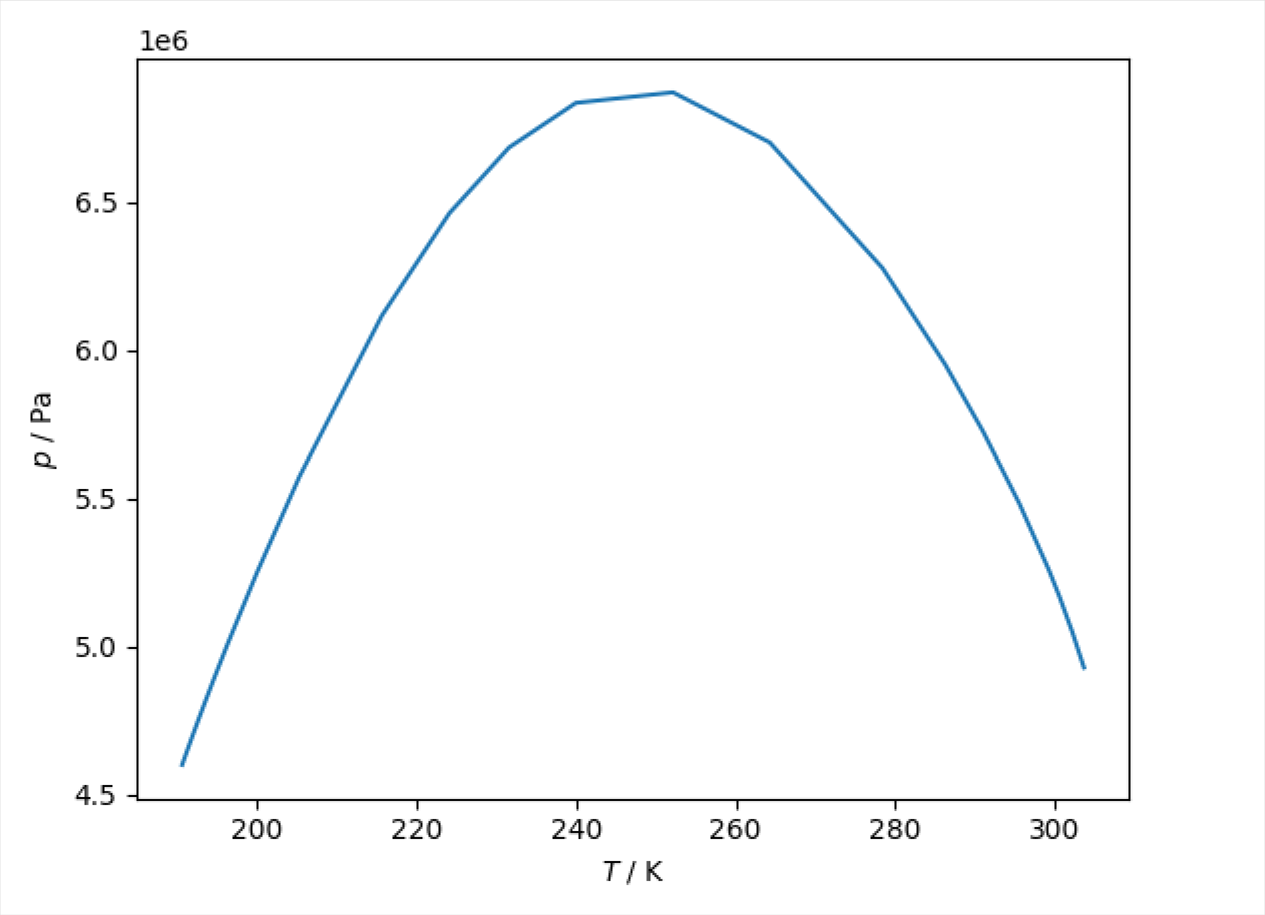
(continues on next page)

(continued from previous page)

```
↪teqp.make_model({'kind':"GERG2004resid", 'model':{"names": ['MeThAnE','ethane']}})

File /opt/conda/lib/python3.11/site-packages/teqp/__init__.py:47, in make_model(*args,
↪ **kwargs)
     42 def make_model(*args, **kwargs):
     43     """
     44     This function is in two parts; first the make_model function (renamed to _
↪make_model in the Python interface)
     45     is used to make the model and then the model-specific methods are␣
↪attached to the instance
     46     """
---> 47     AS = _make_model(*args, **kwargs)
     48     attach_model_specific_methods(AS)
     49     return AS

ValueError: Unable to load pure info for MeThAnE
```

[4]:

```
# Here we trace the critical locus for methane+ethane
rhovec0 = np.array([0.0, 0.0])
ifluid = 0
T0 = model.get_Tcvec()[0]
rhovec0[ifluid] = 1/model.get_vcvec()[0]
trace = model.trace_critical_arclength_binary(T0=T0, rhovec0=rhovec0)
df = pandas.DataFrame(trace)
plt.plot(df['T / K'], df['p / Pa'])
plt.gca().set(xlabel='$T$ / K', ylabel='$p$ / Pa');
```



```
[5]: model = teqp.make_model({'kind':"GERG2004resid", 'model':{"names": ['methane']}})
```

```
[6]: # Build an on-the-fly ancillary equation
     # (not as accurate as the specialized ones, but works acceptably in many cases)
     anc = teqp.build_ancillaries(model, Tc=model.get_Tcvec()[0], rhoc = 1/model.get_
     ↪vcvec()[0], Tmin=60)

     # And then use the dynamic ancillary to calculate VLE at 100 K
     T = 100 # K
     rhoL, rhoV = model.pure_VLE_T(T, anc.rhoL(T), anc.rhoV(T), 10)
     rhoL, rhoV, 'mol/m^3 for liquid and vapor'
```

```
[6]: (27361.12577999801, 42.046298502526746, 'mol/m^3 for liquid and vapor')
```

4.17 Extended Corresponding States

This implements the method of Huber and Ely: https://doi.org/10.1016/0140-7007(94)90083-3

It does not include the undocumented temperature and density terms that are included in REFPROP

```
[1]: import teqp
     teqp.__version__
```

[1]:

```
'0.19.1'
```

[2]:

```
import numpy as np
import CoolProp.CoolProp as CP
```

[3]:

```
# These parameters are from Huber & Ely
j = {
    "kind": "multifluid-ECS-HuberEly1994",
    "model": {
      "reference_fluid": {
            "name": teqp.get_datapath() + "/dev/fluids/R134a.json",
            "acentric": 0.326680,
            "Z_crit": 4.056e6/(5030.8*8.314471*374.179),
            "T_crit / K": 374.179,
            "rhomolar_crit / mol/m^3": 5030.8
      },
      "fluid": {
            "name": "R143a",
            "f_T_coeffs": [ -0.22807e-1, -0.64746],
            "h_T_coeffs": [ 0.36563, -0.26004e-1],
            "acentric": 0.25540,
            "T_crit / K": 346.3,
            "rhomolar_crit / mol/m^3": (1/0.194*1000),
            "Z_crit": 3.76e6/(346.3*8.314471*(1/0.194*1000))
      }
    }
}

model = teqp.make_model(j)
z = np.array([1.0])
R = model.get_R(z)
T, rho = 400, 2600
p = rho*R*T*(1+model.get_Ar01(T, rho, z))
display('pressure from ECS:', p)

display('pressure from EOS:', CP.PropsSI('P','T',T,'Dmolar',rho,'R143a'))
```

```
'pressure from ECS:'
```

```
5556329.442047298
```

```
'pressure from EOS:'
```

```
5478978.746656995
```

4.18 Ideal-gas Models

The collection of ideal-gas contributions are described below. They are summed to yield the ideal-gas contribution from

$$\alpha^{\rm ig} = \sum_i x_i \left(\alpha^{\rm ig}_i(T,\rho) + \ln(x_i)\right)$$

Null mole fractions $x_i = 0$ do not contribute to the summation because

$$\lim_{x_i \to 0} x_i \ln(x_i) = 0$$

4.18.1 IdealHelmholtzConstant

JSON arguments: `"a"`

$$\alpha^{\rm ig} = a$$

4.18.2 IdealHelmholtzLogT

JSON arguments: `"a"`

$$\alpha^{\rm ig} = a\ln(T)$$

which should be compared with the original form in GERG (and REFPROP and CoolProp)

$$\alpha^{\rm ig} = a^*\ln(\tau)$$

with $\tau = T_r/T$.

4.18.3 IdealHelmholtzLead

JSON arguments: `"a_1"`, `"a_2"`

$$\alpha^{\rm ig} = \ln(\rho) + a_1 + a_2/T$$

which should be compared with the original form in GERG (and REFPROP and CoolProp)

$$\alpha^{\rm ig} = \ln(\delta) + a_1^* + a_2^*\tau$$

Note that a_1 contains an additive factor of $-\ln(\rho_r)$ and a_2 contains a multiplicative factor of T_r relative to the former because $\delta = \rho/\rho_r$ and $\tau = T_r/T$.

4.18.4 IdealHelmholtzPowerT

JSON arguments: `"n"`, `"t"`

$$\alpha^{\rm ig} = \sum_k n_k T^{t_k}$$

4.18.5 IdealHelmholtzPlanckEinstein

JSON arguments: `"n"`, `"theta"`

$$\alpha^{\rm ig} = \sum_k n_k \ln(1-\exp(-\theta_k/T))$$

4.18.6 IdealHelmholtzPlanckEinsteinGeneralized

JSON arguments: `"n"`, `"c"`, `"d"`, `"theta"`

$$\alpha^{\rm ig} = \sum_k n_k \ln(c_k + d_k \exp(\theta_k/T))$$

4.18.7 IdealHelmholtzGERG2004Cosh

JSON arguments: `"n"`, `"theta"`

$$\alpha^{\rm ig} = \sum_k n_k \ln(|\cosh(\theta_k/T)|)$$

See Table 7.6 in GERG-2004 monograph

4.18.8 IdealHelmholtzGERG2004Sinh

JSON arguments: `"n"`, `"theta"`

$$\alpha^{\rm ig} = \sum_k n_k \ln(|\sinh(\theta_k/T)|)$$

4.18.9 IdealHelmholtzCp0Constant

JSON arguments: `"c"`, `"T_0"`

$$\alpha^{\rm ig} = c\left(\frac{T-T_0}{T} - \ln\left(\frac{T}{T_0}\right)\right)$$

from a term that is like

$$\frac{c_{p0}}{R} = c$$

4.18.10 IdealHelmholtzCp0PowerT

JSON arguments: `"c"`, `"t"`, `"T_0"`

$$\alpha^{\rm ig} = c\left[T^t\left(\frac{1}{t+1} - \frac{1}{t}\right) - \frac{T_0^{t+1}}{T(t+1)} + \frac{T_0^t}{t}\right]$$

from a term that is like

$$\frac{c_{p0}}{R} = cT^t, t \neq 0$$

The C++ classes implementing these functions are at:

- IdealHelmholtzConstant
- IdealHelmholtzLogT
- IdealHelmholtzLead
- IdealHelmholtzPowerT

- IdealHelmholtzPlanckEinstein
- IdealHelmholtzPlanckEinsteinGeneralized
- IdealHelmholtzGERG2004Cosh
- IdealHelmholtzGERG2004Sinh
- IdealHelmholtzCp0Constant
- IdealHelmholtzCp0PowerT

Conversion

Conversion of terms from CoolProp format to teqp format is carried out in the function CoolProp2teqp_alphaig_term_reformatter().

For instance the leading term in CoolProp goes like:

$$\alpha = \ln(\delta) + a_1^* + a_2^*\tau$$

with the * indicating the CoolProp formulation. The term reads like

$$\alpha = \ln(\rho) + a_1 + a_2/T$$

in teqp. Refactoring the CoolProp term reads

$$\alpha = \ln(\rho) - \ln(\rho_r) + a_1^* + a_2^*\left(\frac{T_r}{T}\right)$$

so that $a_1 = a_1^* - \ln(\rho_r)$ and $a_2 = a_2^* T_r$

In some cases reconstitutions of terms are required, as the supported terms in the libraries are somewhat different. The term used in CoolProp to do the offsets to enthalpy and entropy is of the form

$$\alpha = a_1^* + a_2^*\tau = a_1^* + a_2^*\left(\frac{T_r}{T}\right)$$

so that term can be rewritten as an `IdealHelmholtzPowerT` with coefficients of a_1^* and $a_2^* T_r$ and exponents of 0 and -1.

Most of the remaining terms can be converted in a straightforward fashion, except for some of GERG formulations that are a bit trickier. Mostly, the only conversion required is to multiply or divide by reducing temperatures so that all arguments are in terms of temperature as independent variable.

The mathematics describing how to do the conversion from a term in c_p^0/R follows:

$$\alpha_0 = \frac{a_0}{RT} = -1 + \ln\frac{\rho T}{\rho_0 T_0} + \frac{h_0^0}{RT} - \frac{s_0^0}{R} + \frac{1}{RT}\int_{T_0}^{T} c_p^0(T)dT - \frac{1}{R}\int_{T_0}^{T}\frac{c_p^0(T)}{T}dT$$

$$\alpha_0 = \frac{a_0}{RT} = \ln(\rho) + \ln(T) - \ln(\rho_0 T_0) - 1 + \frac{h_0^0}{RT} - \frac{s_0^0}{R} + \frac{1}{RT}\int_{T_0}^{T} c_p^0(T)dT - \frac{1}{R}\int_{T_0}^{T}\frac{c_p^0(T)}{T}dT$$

You can set the values of h_0^0 and h_0^0 to any value, including zero. So if you are converting a term from c_p^0/R, then you could do

$$\alpha_0 = \frac{a_0}{RT} = \ln(\rho) + \ln(T) - \ln(\rho_0 T_0) - 1 + \frac{1}{RT}\int_{T_0}^{T} c_p^0(T)dT - \frac{1}{R}\int_{T_0}^{T}\frac{c_p^0(T)}{T}dT$$

[1]:

```
import teqp, os, numpy as np, json
display(teqp.__version__)
```

```
'0.19.1'
```

[2]:

```
path = teqp.get_datapath()+'/dev/fluids/n-Propane.json'
assert(os.path.exists(path))
jig = teqp.convert_CoolProp_idealgas(path, 0)
print('As in the fluid file (matches Lemmon JPCRD 2009 exactly)::::')
print(json.dumps(json.load(open(path))['EOS'][0]['alpha0'],indent=1))
print('\n\nAnd after conversion::::')
print(json.dumps(jig,indent=1))
```

```
As in the fluid file (matches Lemmon JPCRD 2009 exactly)::::
[
 {
  "a1": -4.970583,
  "a2": 4.29352,
  "type": "IdealGasHelmholtzLead"
 },
 {
  "a": 3,
  "type": "IdealGasHelmholtzLogTau"
 },
 {
  "n": [
   3.043,
   5.874,
   9.337,
   7.922
  ],
  "t": [
   1.062478,
   3.344237,
   5.363757,
   11.762957
  ],
  "type": "IdealGasHelmholtzPlanckEinstein"
 }
]


And after conversion::::
{
 "R": 8.314472,
 "terms": [
  {
   "R": 8.314472,
   "a_1": -13.487776191416238,
   "a_2": 1588.1301128,
   "type": "Lead"
  },
  {
   "R": 8.314472,
   "a": 17.739616992418114,
   "type": "Constant"
  },
  {
```

(continues on next page)

(continued from previous page)

```
   "R": 8.314472,
   "a": -3.0,
   "type": "LogT"
  },
  {
   "R": 8.314472,
   "n": [
    3.043,
    5.874,
    9.337,
    7.922
   ],
   "theta": [
    392.99998742,
    1236.99982393,
    1984.0000767299998,
    4351.00016473
   ],
   "type": "PlanckEinstein"
  }
 ]
}
```

Note that the two leading terms of Lemmon generates three terms in teqp because the forms of the terms are slightly different

[3]:

```
# As an worked example, the conversions can be carried out like so, with the values␣
↪from Lemmon given name of b instead of a
b_1 = -4.970583
b_2 = 4.29352
Tr = 369.89 # K
rhor = 5000 # mol/m^3
print('a_1:', b_1-np.log(rhor))
print('a_2:', b_2*Tr)
# The 3*ln(tau) term turns into 3*ln(Tr) - 3*ln(T)
print(np.log(Tr)*3)
# and the theta values are obtained
t = np.array([1.062478, 3.344237, 5.363757, 11.762957])
print((t*Tr).tolist())
```

```
a_1: -13.487776191416238
a_2: 1588.1301128
17.739616992418114
[392.99998742, 1236.99982393, 1984.0000767299998, 4351.00016473]
```

[4]:

```
aig = teqp.IdealHelmholtz([jig])
-aig.get_Ar20(300, 3, np.array([1.0]))
```

[4]:

```
7.863830967842212
```

CHAPTER FIVE

ALGORITHMS

5.1 Phase equilibria

Two basic approaches are implemented in teqp:

- Iterative calculations given guess values
- Tracing along iso-curves (constant temperature, etc.) powered by the isochoric thermodynamics formalism

```
[1]: import teqp
     import numpy as np
     import pandas
     import matplotlib.pyplot as plt
     teqp.__version__
```

```
[1]: '0.19.1'
```

5.1.1 Iterative Phase Equilibria

Pure fluid

For a pure fluid, phase equilibrium between two phases is defined by equating the pressures and Gibbs energies in the two phases. This represents a 2D non-linear rootfinding problem. Newton's method can be used for the rootfinding, and in teqp, automatic differentiation is used to obtain the necessary Jacobian matrix so the implementation is quite efficient.

The method requires guess values, which are the densities of the liquid and vapor densities. In some cases, ancillary or superancillary equations have been developed which provide curves of guess densities as a function of temperature.

For a pure fluid, you can use the `pure_VLE_T` method to carry out the iteration.

The Python method is here: `pure_VLE_T`

```
[2]: # Instantiate the model
     model = teqp.canonical_PR([300], [4e6], [0.1])

     T = 250 # [K], Temperature to be used

     # Here we use the superancillary to get guess values (actually these are more
     # accurate than the results we will obtain from iteration!)
     rhoL0, rhoV0 = model.superanc_rhoLV(T)
     display('guess:', [rhoL0, rhoV0])

     # Carry out the iteration, return the liquid and vapor densities
```

(continues on next page)

(continued from previous page)

```
# The guess values are perturbed to make sure the iteration is actually
# changing the values
model.pure_VLE_T(T, rhoL0*0.98, rhoV0*1.02, 10)
```

```
'guess:'
```

```
[12735.311173407898, 752.4082303122791]
```

```
[2]: array([12735.31117341,   752.40823031])
```

Binary Mixture

For a binary mixture, the approach is roughly similar to that of a pure fluid. The pressure is equated between phases, and the chemical potentials of each component in each phase are forced to be the same.

Again, the user is required to provide guess values, in this case molar concentrations in each phase, and a Newton method is implemented to solve for the phase equilibrium. The analytical Jacobian is obtained from automatic differentiation.

The `mix_VLE_Tx` function is the binary mixture analog to `pure_VLE_T` for pure fluids.

The Python method is here: `mix_VLE_Tx`

```
[3]: zA = np.array([0.01, 0.99])
model = teqp.canonical_PR([300,310], [4e6,4.5e6], [0.1, 0.2])
model1 = teqp.canonical_PR([300], [4e6], [0.1])
T = 273.0 # [K]
# start off at pure of the first component
rhoL0, rhoV0 = model1.superanc_rhoLV(T)

# then we shift to the given composition in the first phase
# to get guess values
rhovecA0 = rhoL0*zA
rhovecB0 = rhoV0*zA

# carry out the iteration
code, rhovecA, rhovecB = model.mix_VLE_Tx(T, rhovecA0, rhovecB0, zA,
     1e-10, 1e-10, 1e-10, 1e-10,  # stopping conditions
     10 # maximum number of iterations
    )
code, rhovecA, rhovecB
```

```
[3]: (<VLE_return_code.xtol_satisfied: 1>,
 array([  128.66049209, 12737.38871682]),
 array([  12.91868229, 1133.77242677]))
```

You can (and should) check the value of the return code to make sure the iteration succeeded. Do not rely on the numerical value of the enumerated return codes!

5.2 Tracing (isobars and isotherms)

When it comes to mixture thermodynamics, as soon as you add another component to a pure component to form a binary mixture, the complexity of the thermodynamics entirely changes. For that reason, mixture iterative calculations for mixtures are orders of magnitude more difficult to carry out. Asymmetric mixtures can do all sorts of interesting things that are entirely unlike those of pure fluids, and the algorithms are therefore much, much more complicated. Formulating phase equilibrium problems is not much more complicated than for pure fluids, but the most challenging aspect is to obtain good guess values from which to start an iterative routine, and the difficulty of this problem increases with the complexity of the mixture thermodynamics.

Ulrich Deiters and Ian Bell have developed a number of algorithms for tracing phase equilibrium solutions as the solution of ordinary differential equations rather than carrying out iterative routines for a given state point. The advantage of the tracing calculations is that they can often be initiated at a state point that is entirely known, for instance the pure fluid endpoint for a subcritical isotherm or isobar.

The Python method is here: `trace_VLE_isotherm_binary`

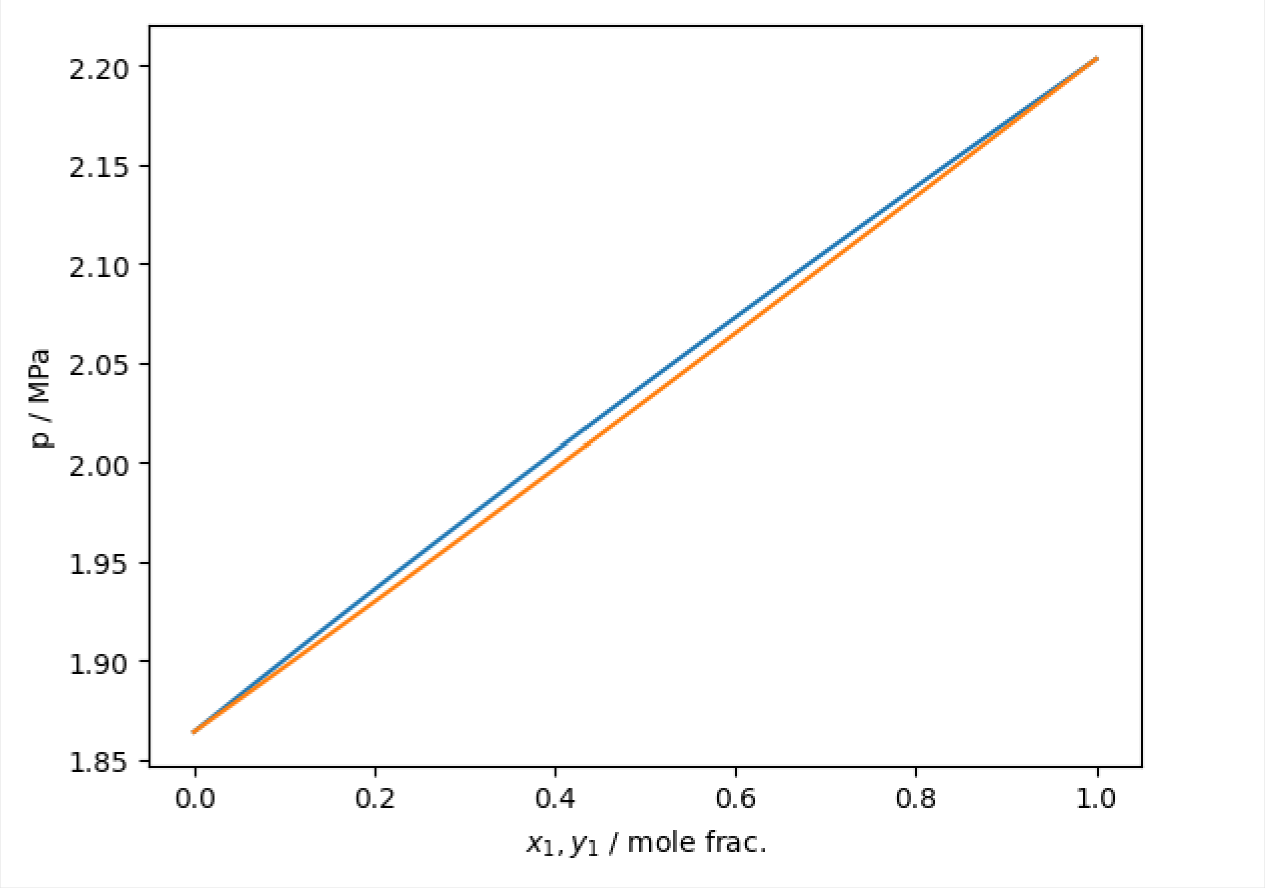
The C++ implementation returns a string in JSON format, which can be conveniently operated upon, for instance after converting the returned data structure to a `pandas.DataFrame`. A simple example of plotting a subcritical isotherm for a "boring" mixture is presented here:

[4]:
```
model = teqp.canonical_PR([300,310], [4e6,4.5e6], [0.1, 0.2])
model1 = teqp.canonical_PR([300], [4e6], [0.1])
T = 273.0 # [K]
rhoL0, rhoV0 = model1.superanc_rhoLV(T) # start off at pure of the first component
j = model.trace_VLE_isotherm_binary(T, np.array([rhoL0, 0]), np.array([rhoV0, 0]))
display(str(j)[0:100]+'...') # The first few bits of the data
df = pandas.DataFrame(j) # Now as a data frame
df.head(3)
```

```
"[{'T / K': 273.0, 'c': -1.0, 'drho/dt': [-0.618312383229212, 0.7690760182230469, -0.
↪1277526773161415…"
```

[4]:
```
   T / K    c                                        drho/dt        dt  \
0  273.0 -1.0  [-0.618312383229212, 0.7690760182230469, -0.12…  0.000010   
1  273.0 -1.0  [-0.6183123817120353, 0.7690760162922189, -0.1…  0.000045   
2  273.0 -1.0  [-0.6183123827116788, 0.7690760173388914, -0.1…  0.000203   

        pL / Pa       pV / Pa                               rhoL / mol/m^3  \
0  2.203397e+06  2.203397e+06                    [10697.985891540735, 0.0]   
1  2.203397e+06  2.203397e+06  [10697.985885357639, 7.690760309421386e-06]   
2  2.203397e+06  2.203397e+06   [10697.98585753358, 4.229918121248511e-05]   

                                rhoV / mol/m^3         t  xL_0 / mole frac.  \
0                    [1504.6120879290752, 0.0]  0.000000                1.0   
1  [1504.6120866515366, 9.945415375682985e-07]  0.000010                1.0   
2  [1504.6120809026731, 5.469978386095445e-06]  0.000055                1.0   

   xV_0 / mole frac.  
0                1.0  
1                1.0  
2                1.0  
```

[5]:
```
plt.plot(df['xL_0 / mole frac.'], df['pL / Pa']/1e6)
plt.plot(df['xV_0 / mole frac.'], df['pL / Pa']/1e6)
plt.gca().set(xlabel='$x_1,y_1$ / mole frac.', ylabel='p / MPa')
plt.show()
```



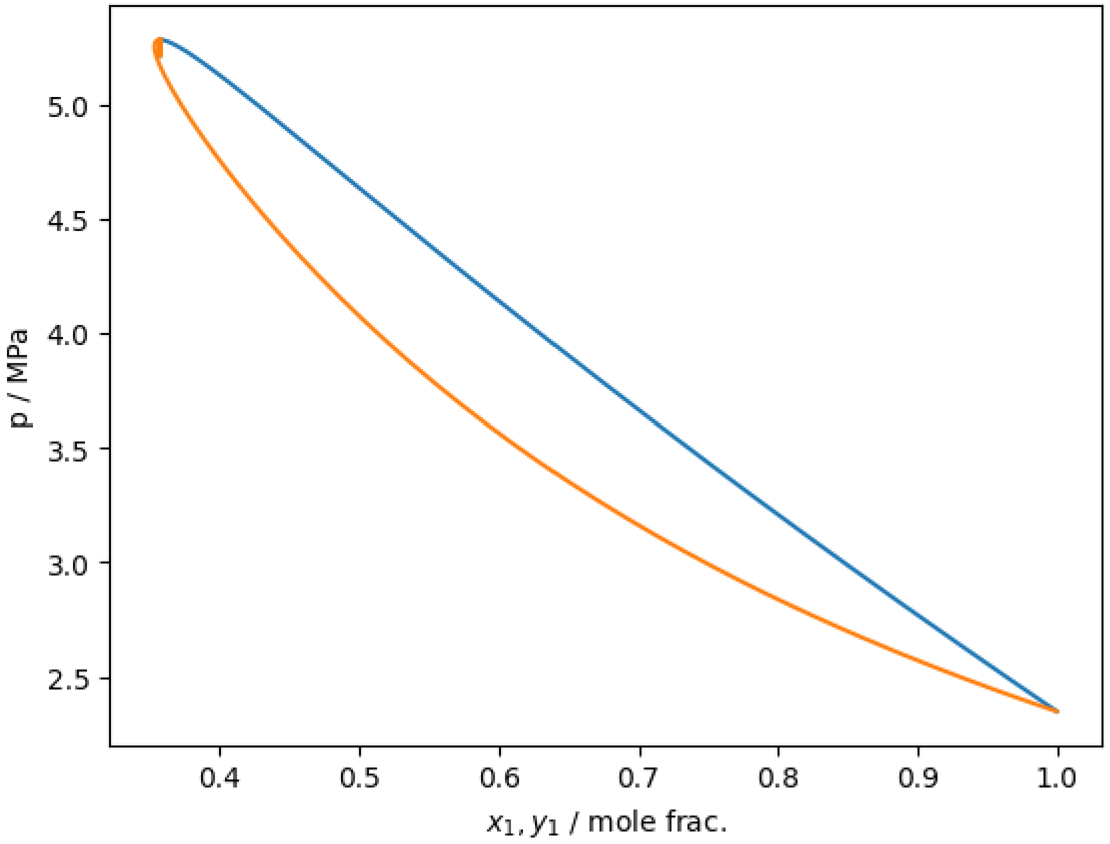
Isn't that exciting!

You can also provide an optional set of flags to the function to control other behaviors of the function, and switch between simple Euler and adaptive RK45 integration (the default)

The options class is here: `TVLEOptions`

Supercritical isotherms work approximately in the same manner

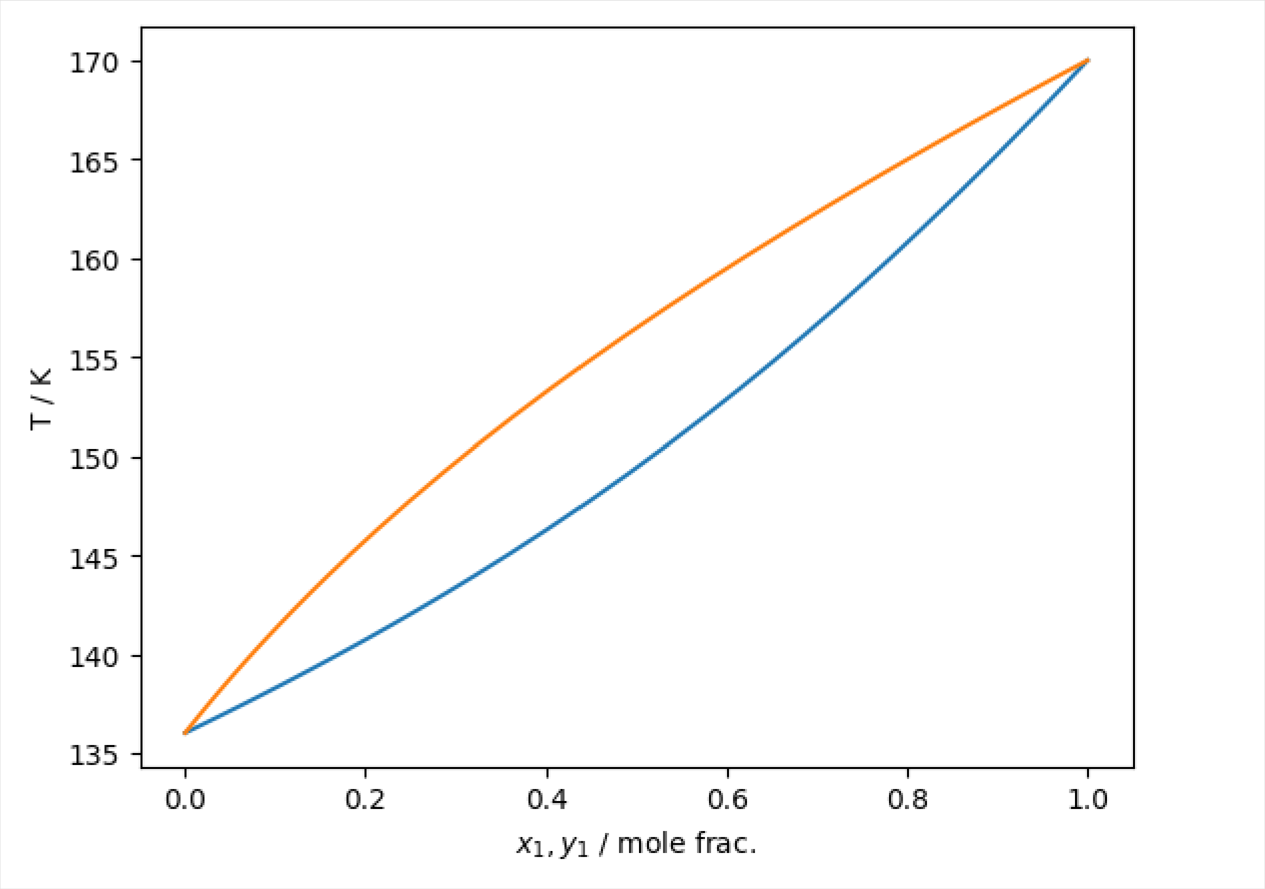
[6]:
```
Tc_K = [190.564, 154.581]
pc_Pa = [4599200, 5042800]
acentric = [0.011, 0.022]
model = teqp.canonical_PR(Tc_K, pc_Pa, acentric)
model1 = teqp.canonical_PR([Tc_K[0]], [pc_Pa[0]], [acentric[0]])
T = 170.0 # [K] # Note: above Tc of the second component
rhoL0, rhoV0 = model1.superanc_rhoLV(T) # start off at pure of the first component
j = model.trace_VLE_isotherm_binary(T, np.array([rhoL0, 0]), np.array([rhoV0, 0]))
df = pandas.DataFrame(j) # Now as a data frame
plt.plot(df['xL_0 / mole frac.'], df['pL / Pa']/1e6)
plt.plot(df['xV_0 / mole frac.'], df['pL / Pa']/1e6)
plt.gca().set(xlabel='$x_1,y_1$ / mole frac.', ylabel='p / MPa')
plt.show()
```



As of version 0.10.0, isobar tracing has been added to `teqp`. It operates in fundamentally the same fashion as the isotherm tracing and the same recommendations about starting at a pure fluid apply

The tracer function class is here: `trace_VLE_isobar_binary`

[7]:
```
Tc_K = [190.564, 154.581]
pc_Pa = [4599200, 5042800]
acentric = [0.011, 0.022]
model = teqp.canonical_PR(Tc_K, pc_Pa, acentric)
model1 = teqp.canonical_PR([Tc_K[0]], [pc_Pa[0]], [acentric[0]])
T = 170.0 # [K] # Note: above Tc of the second component
rhoL0, rhoV0 = model1.superanc_rhoLV(T) # start off at pure of the first component
p0 = rhoL0*model1.get_R(np.array([1.0]))*T*(1+model1.get_Ar01(T, rhoL0, np.array([1.
↪0])))
j = model.trace_VLE_isobar_binary(p0, T, np.array([rhoL0, 0]), np.array([rhoV0, 0]))
df = pandas.DataFrame(j) # Now as a data frame
plt.plot(df['xL_0 / mole frac.'], df['T / K'])
plt.plot(df['xV_0 / mole frac.'], df['T / K'])
plt.gca().set(xlabel='$x_1,y_1$ / mole frac.', ylabel='T / K')
plt.show()
```



5.3 VLLE

Following the approach described in Bell et al.: https://doi.org/10.1021/acs.iecr.1c04703

for the mixture of nitrogen + ethane, with the default thermodynamic model in teqp, which is the GERG-2008 mixing parameters (no departure function).

Two traces are made, and the intersection is obtained, this gives you the VLLE solution.

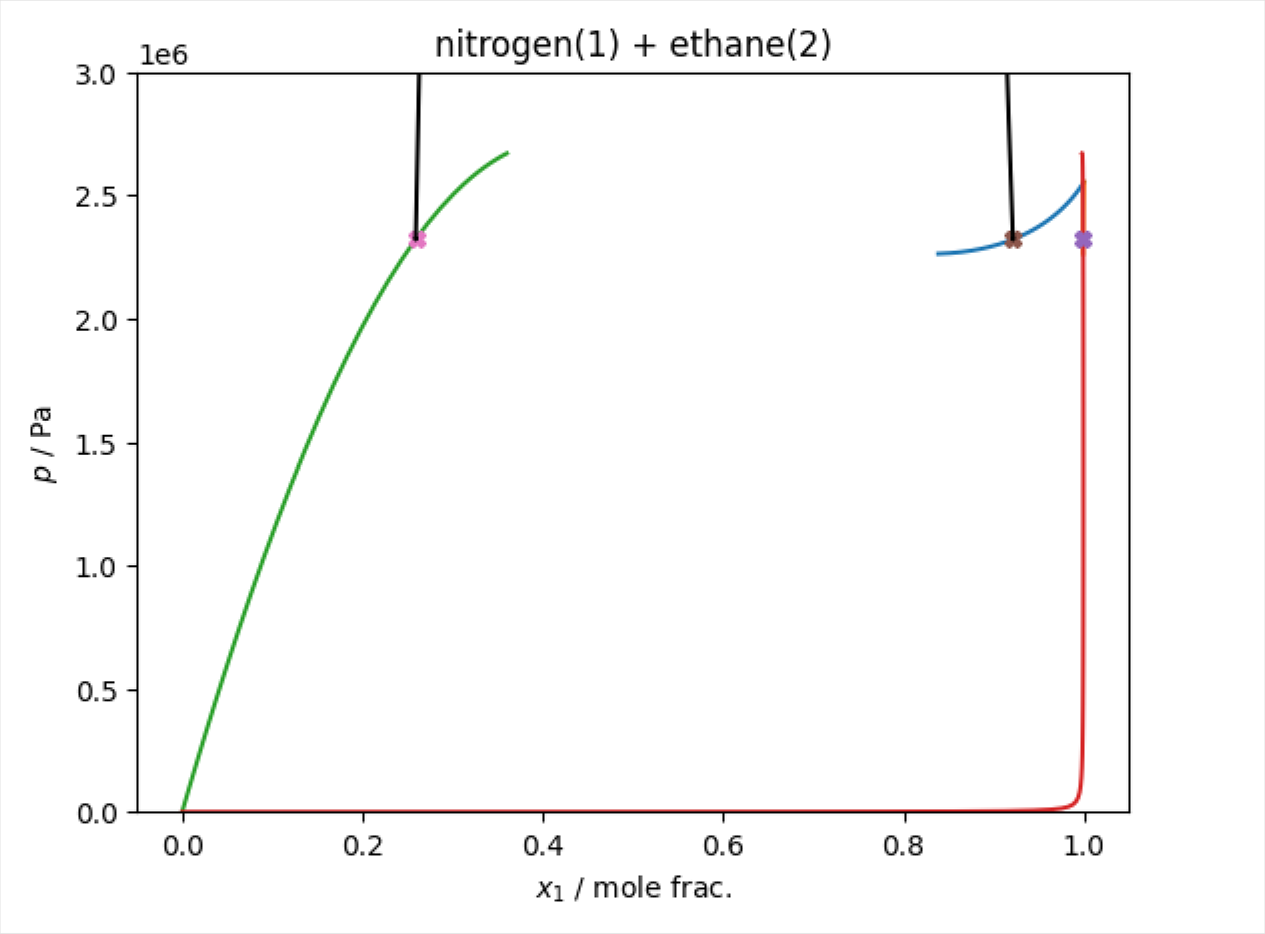
[1]:
```
import teqp, numpy as np, matplotlib.pyplot as plt, pandas

def get_traces(*, T, ipures):
    names = ['Nitrogen', 'Ethane']
    model = teqp.build_multifluid_model(names, teqp.get_datapath())
    pures = [teqp.build_multifluid_model([name], teqp.get_datapath()) for name in␣
↪names]
    traces = []
    for ipure in ipures:
        # Init at the pure fluid endpoint
        anc = pures[ipure].build_ancillaries()
        rhoLpure, rhoVpure = pures[ipure].pure_VLE_T(T, anc.rhoL(T), anc.rhoV(T), 10)

        rhovecL = np.array([0.0, 0.0])
        rhovecV = np.array([0.0, 0.0])
```

(continues on next page)

(continued from previous page)

```
        rhovecL[ipure] = rhoLpure
        rhovecV[ipure] = rhoVpure
        opt = teqp.TVLEOptions()
        opt.p_termination = 1e8
        opt.crit_termination=1e-4
        opt.calc_criticality=True
        j = model.trace_VLE_isotherm_binary(T, rhovecL, rhovecV, opt)
        traces.append(j)
    return model, traces

T = 120.3420
model, traces = get_traces(T=T, ipures=[0,1])
for trace in traces:
    df = pandas.DataFrame(trace)
    plt.plot(df['xL_0 / mole frac.'], df['pL / Pa'])
    plt.plot(df['xV_0 / mole frac.'], df['pV / Pa'])

# Do the VLLE solving
for soln in model.find_VLLE_T_binary(traces):
    print('rhovec / mol/m^3 | p / Pa')
    for rhovec in soln['polished']:
        rhovec = np.array(rhovec)
        rhotot = sum(rhovec)
        x = rhovec/rhotot
        p = rhotot*model.get_R(x)*T*(1+model.get_Ar01(T, rhotot, x))
        plt.plot(x[0], p, 'X')
        print(rhovec, p)

    # And also carry out the LLE trace for the two liquid phases
    j = model.trace_VLE_isotherm_binary(T, np.array(soln['polished'][1]), np.
↪array(soln['polished'][2]))
    df = pandas.DataFrame(j)
    plt.plot(df['xL_0 / mole frac.'], df['pL / Pa'], 'k')
    plt.plot(df['xV_0 / mole frac.'], df['pV / Pa'], 'k')

# Plotting niceties
plt.ylim(top=3e6, bottom=0)
plt.gca().set(xlabel='$x_1$ / mole frac.', ylabel='$p$ / Pa', title='nitrogen(1) +␣
↪ethane(2)')
plt.show()
```

```
rhovec / mol/m^3 | p / Pa
[3.66984834e+03 3.25893958e+00] 2321103.087319132
[19890.16767481  1698.86505766] 2321103.087318946
[ 5641.24690517 16140.85769908] 2321103.0873195715
```



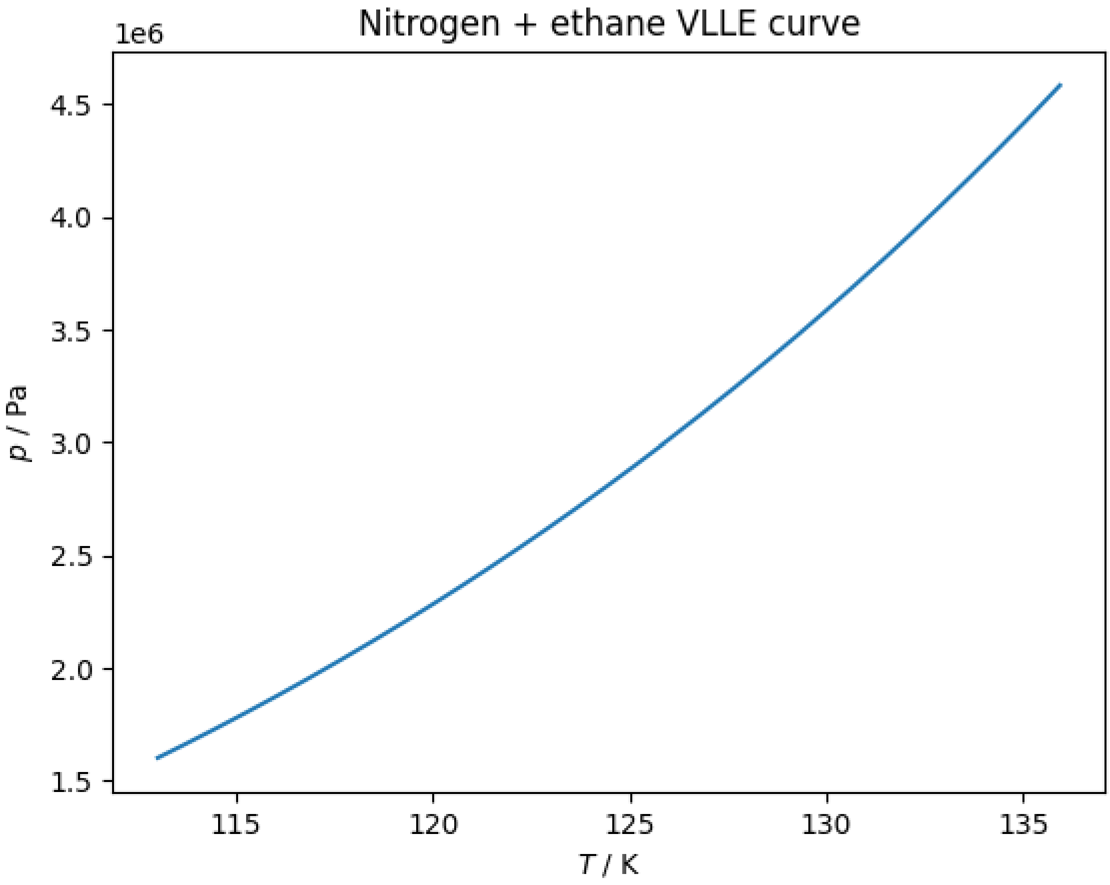
[2]:
```
# Trace from both pure fluid endpoints
T = 113
model, traces = get_traces(T=T, ipures = [0,1])

# Find the VLLE solution for the starting temperature
solns = model.find_VLLE_T_binary(traces)
rhovecV, rhovecL1, rhovecL2 = solns[0]['polished']

# Obtain the VLLE trace towards higher temperatures
opt = teqp.VLLETracerOptions()
a = lambda x: np.array(x)
VLLE = model.trace_VLLE_binary(T, a(rhovecV), a(rhovecL1), a(rhovecL2), opt)
df = pandas.DataFrame(VLLE)

# Add the pressure to the DataFrame
def add_ps(row, key):
    T = row['T / K']
    rhovec = np.array(row[key])
    rhotot = sum(rhovec)
    x = rhovec/rhotot
    p = rhotot*model.get_R(x)*T*(1+model.get_Ar01(T, rhotot, x))
    return p
df['p / Pa']  = df.apply(add_ps, axis=1, key='rhoV / mol/m^3')

# Plot the p-T curve
```

(continues on next page)

(continued from previous page)

```
plt.plot(df['T / K'], df['p / Pa'])
plt.gca().set(xlabel='$T$ / K', ylabel='$p$ / Pa');
plt.title('Nitrogen + ethane VLLE curve')
```

[2]:
```
Text(0.5, 1.0, 'Nitrogen + ethane VLLE curve')
```



5.4 VLLE @ constant pressure

Following the approach described in Bell et al.: https://doi.org/10.1021/acs.iecr.1c04703, but slightly different because the pressure is fixed rather than the temperature, but the same basic principles hold

for the mixture of nitrogen + ethane, with the default thermodynamic model in teqp, which is the GERG-2008 mixing parameters (no departure function).

Two traces are made, and the intersection is obtained, this gives you the VLLE solution.

[1]:
```
import teqp, numpy as np, matplotlib.pyplot as plt, pandas
import CoolProp.CoolProp as CP

names = ['Nitrogen', 'Ethane']
model = teqp.build_multifluid_model(names, teqp.get_datapath())
pures = [teqp.build_multifluid_model([name], teqp.get_datapath()) for name in names]
p = 29e5 # Pa
```

(continues on next page)

(continued from previous page)

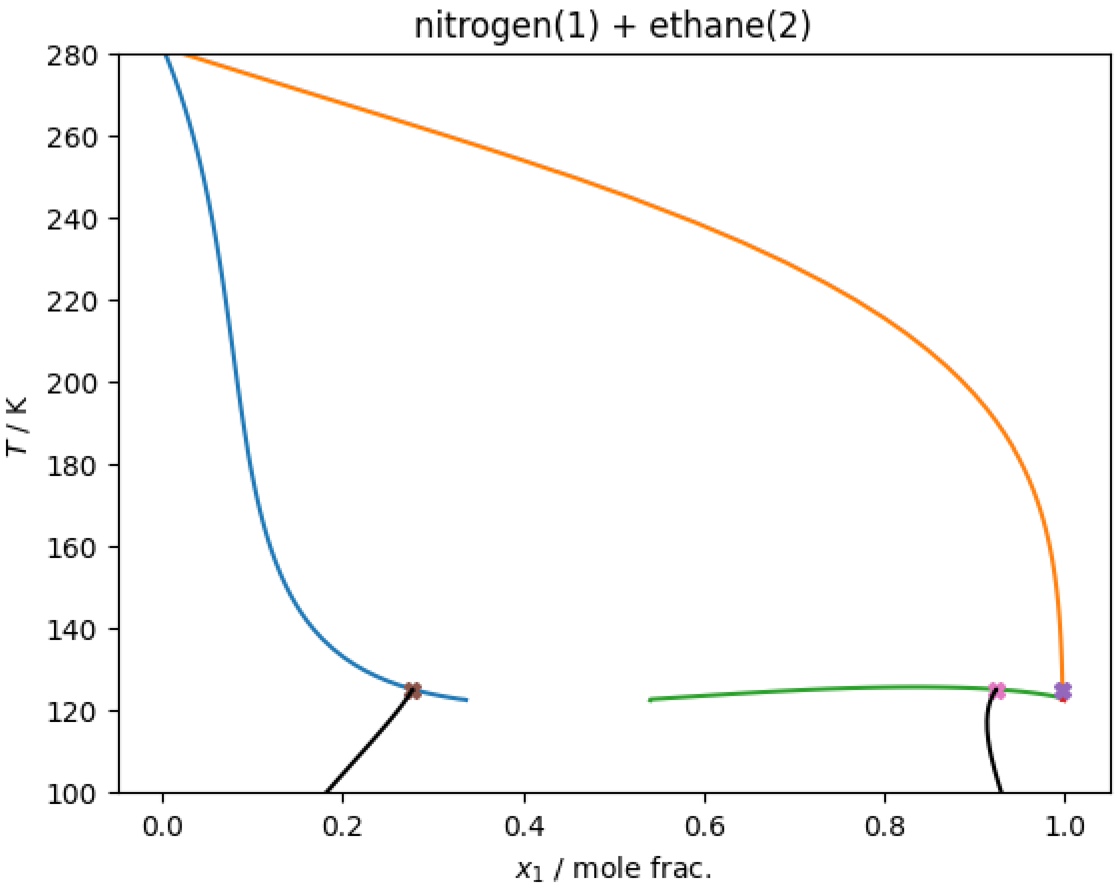
```
# Trace from both pure fluid endpoints
traces = []
for ipure in [1,0]:
    # Init at the pure fluid endpoint
    anc = pures[ipure].build_ancillaries()
    rhoLpure, rhoVpure = [CP.PropsSI('Dmolar','P',p,'Q',Q,names[ipure]) for Q in [0,
↪1]]
    T = CP.PropsSI('T','P',p,'Q',0,names[ipure])

    rhovecL = np.array([0.0, 0.0])
    rhovecV = np.array([0.0, 0.0])
    rhovecL[ipure] = rhoLpure
    rhovecV[ipure] = rhoVpure
    j = model.trace_VLE_isobar_binary(p, T, rhovecL, rhovecV)
    df = pandas.DataFrame(j)
    plt.plot(df['xL_0 / mole frac.'], df['T / K'])
    plt.plot(df['xV_0 / mole frac.'], df['T / K'])
    traces.append(j)

# Do the VLLE solving
for soln in model.find_VLLE_p_binary(traces):
    T = soln['polished'][-1]
    print('rhovec / mol/m^3 | T / K')
    for rhovec in soln['polished'][0:3]:
        rhovec = np.array(rhovec)
        rhotot = sum(rhovec)
        x = rhovec/rhotot
        p = rhotot*model.get_R(x)*T*(1+model.get_Ar01(T, rhotot, x))
        plt.plot(x[0], T, 'X')
        print(rhovec, T)

    # And also carry out the LLE trace for the two liquid phases
    opt = teqp.PVLEOptions()
    opt.integration_order = 5
    opt.init_dt = 1e-10
    # Or could be 1 depending on the initial integration direction, do not know the␣
↪direction
    # a priori because not starting at a pure fluid endpoint
    for init_dt in [-1]:
        opt.init_c = init_dt
        rhovecV, rhovecL1, rhovecL2, T = soln['polished']
        j = model.trace_VLE_isobar_binary(p, T, np.array(rhovecL1), np.
↪array(rhovecL2), opt)
        df = pandas.DataFrame(j)
        plt.plot(df['xL_0 / mole frac.'], df['T / K'], 'k')
        plt.plot(df['xV_0 / mole frac.'], df['T / K'], 'k')

# Plotting niceties
plt.ylim(top=280, bottom=100)
plt.gca().set(xlabel='$x_1$ / mole frac.', ylabel='$T$ / K', title='nitrogen(1) +␣
↪ethane(2)')
plt.show()
```

```
rhovec / mol/m^3 | T / K
[4921.97976373    9.6755684 ] 125.14729018874252
[ 6008.68040253 15630.22353351] 125.14729018874252
```

(continues on next page)

(continued from previous page)

```
[18948.39537895  1540.60935171] 125.14729018874252
```



```
[1]: import scipy.interpolate
import teqp
import numpy as np
import pandas
import matplotlib.pyplot as plt
teqp.__version__
```

```
[1]: '0.19.1'
```

5.5 Critical curves & points

5.5.1 Pure Fluids

Solving for the critical point involves finding the temperature and density that make

$$\left(\frac{\partial p}{\partial \rho}\right)_T = \left(\frac{\partial^2 p}{\partial \rho^2}\right)_T = 0$$

by 2D non-linear rootfinding. Newton steps are taken, and the analytic Jacobian is used (thanks to the ability to do derivatives with automatic differentiation). This is all handily wrapped up in the `solve_pure_critical` method which requires the user to provide guess values for temperature and density

[2]:
```
# Values taken from http://dx.doi.org/10.6028/jres.121.011
modelPR = teqp.canonical_PR([190.564], [4599200], [0.011])

# Solve for the critical point from a point close to the critical point
T0 = 192.0
# Critical compressibility factor of P-R is 0.307401308698.. (see https://doi.org/10.
↪1021/acs.iecr.1c00847)
rhoc = (4599200/(8.31446261815324*190.564))/0.3074
rho0 = rhoc*1.2345 # Perturb to make sure we are doing something in the solver
modelPR.solve_pure_critical(T0, rho0)
```

[2]:
```
(190.564, 9442.816240022832)
```

If you have a mixture, but want to obtain the critical point of a pure fluid of this mixture, you can specify the index of the component in the mixture, as well as the number of componnts in the mixture with something like:

`model.solve_pure_critical(T0, rho0, {"alternative_pure_index": 1, "alternative_length": 2})` so here, for the second fluid, with 0-based index of 1, in a two-component mixture

5.5.2 Mixtures

A pure fluid has a single vapor-liquid critical point, but mixtures are different:

- They may have multiple (or zero!) critical points for a given mixture composition
- The critical curves may not emanate from the pure fluid endpoints

When it comes to critical points, intuition from pure fluids is not helpful, or sometimes even counter-productive.

teqp has methods for working with the critical loci of binary mixtures (only binary mixtures, for now) and especially, methods for tracing the critical curves emanating from the pure fluid endpoints.

The tracing method in teqp is based explicitly on the isochoric thermodynamics formalism introduced by Ulrich Deiters and Sergio Quinones-Cisneros. It uses the Helmholtz energy density as the fundamental potential and all other properties are derived from it. For critical curves it is based upon the integration of sets of ordinary differential equations; the differential equations are in the form of derivatives of the molar concentrations of each component in the mixture with respect to an integration variable. The set of ODE is then integrated.

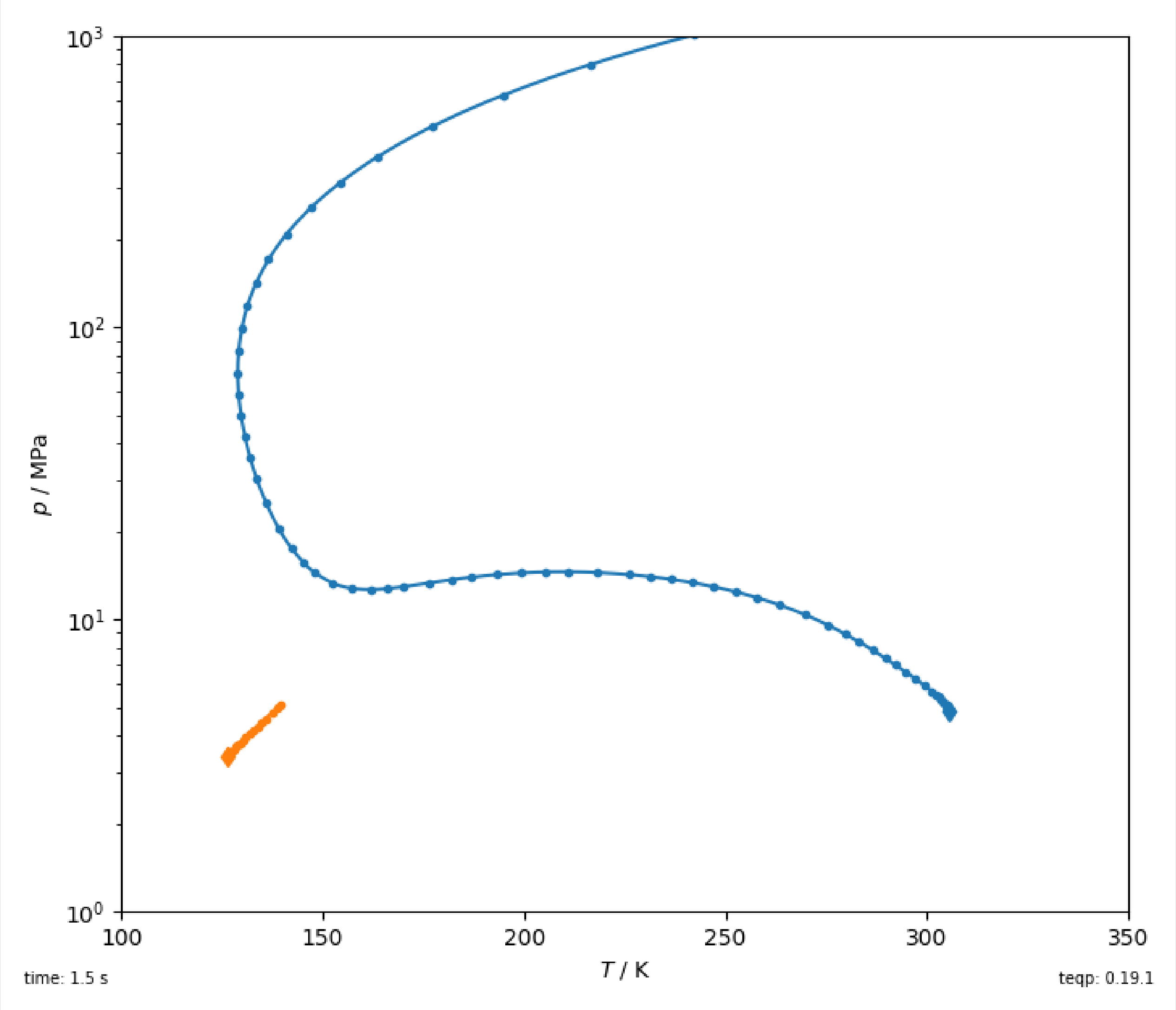
Here is an example of the construction of the critical curves emanating from the pure fluid endpoints for the mixture nitrogen + ethane.

[3]:
```
import timeit
import numpy as np
import matplotlib.pyplot as plt
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 arclength tracing parameter
    o.rel_err = 1e-8
```

(continues on next page)

(continued from previous page)

```
    o.abs_err = 1e-5
    o.integration_order = 5
    o.calc_stability = True
    o.polish = True
    curveJSON = model.trace_critical_arclength_binary(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

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, '.')

    # And interpolate to smooth out the curve using the arclength
    # parameter (which must be monotonically increasing) as
    # the interpolation variable
    tinterp = np.linspace(df['t'].min(), df['t'].max(), 10000)
    Tinterp = scipy.interpolate.interp1d(df['t'], df['T / K'], kind='cubic')(tinterp)
    pinterp = scipy.interpolate.interp1d(df['t'], df['p / Pa'], kind='cubic')(tinterp)
    plt.plot(Tinterp, pinterp/1e6, color=line.get_color())

    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);
```



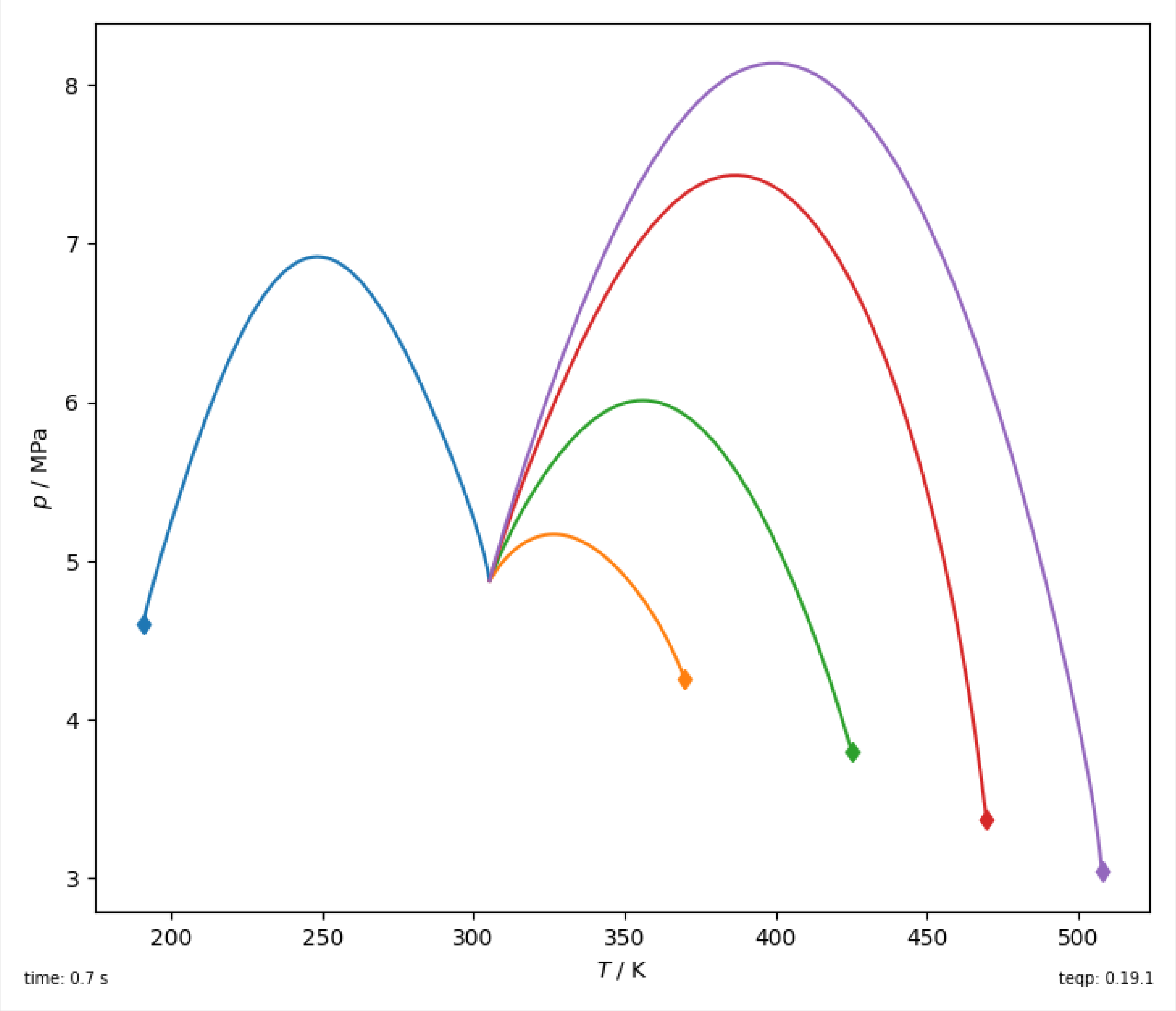
And now for something a bit more interesting: ethane + alkane critical curves

```
[4]: import timeit
import numpy as np
import matplotlib.pyplot as plt
import pandas
import teqp

def get_critical_curve(names, ipure):
    """ Return curve as pandas DataFrame """
    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()
#     print(dir(o))
    o.init_dt = 1.0 # step in the parameter
    o.rel_err = 1e-6 # relative error on the step
    o.abs_err = 1e-6 # absolute error on the step
    o.max_dt = 100 # cap the size of the allowed step
```

(continues on next page)

(continued from previous page)

```
    o.calc_stability = True
    o.polish = True
    curveJSON = model.trace_critical_arclength_binary(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

fig, ax = plt.subplots(1,1,figsize=(7, 6))
tic = timeit.default_timer()
name0 = 'ETHANE'
for othername in ['METHANE','PROPANE','BUTANE','PENTANE','HEXANE']:
    for ipure in [1]:
        df = get_critical_curve([name0, othername], ipure)
        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.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);
```



5.5.3 Pure fluid EOS with nonanalytic terms

For the highest accuracy EOS for normal water and carbon dioxide, there are non-analytic terms that prevent the initialization of the critical tracing at the pure fluid critical point. Instead, one can start close to, but not *AT*, the pure fluid endpoint. After deciding on that starting composition, one solves for the critical point and then traces away from it.

You might need to either do tracing in two parts, one with init_c=+1 and then init_c=-1, or one tracing might be good enough.

Here is an example:

[5]:
```
def get_critical_curve_composition(names, T0, rhovec0, init_c=-1):
    """ Trace the critical curve from a fixed point along it """
    o = teqp.TCABOptions()
#     print(dir(o))
    o.init_dt = 1.0 # step in the parameter
    o.rel_err = 1e-6 # relative error on the step
    o.abs_err = 1e-6 # absolute error on the step
```

(continues on next page)

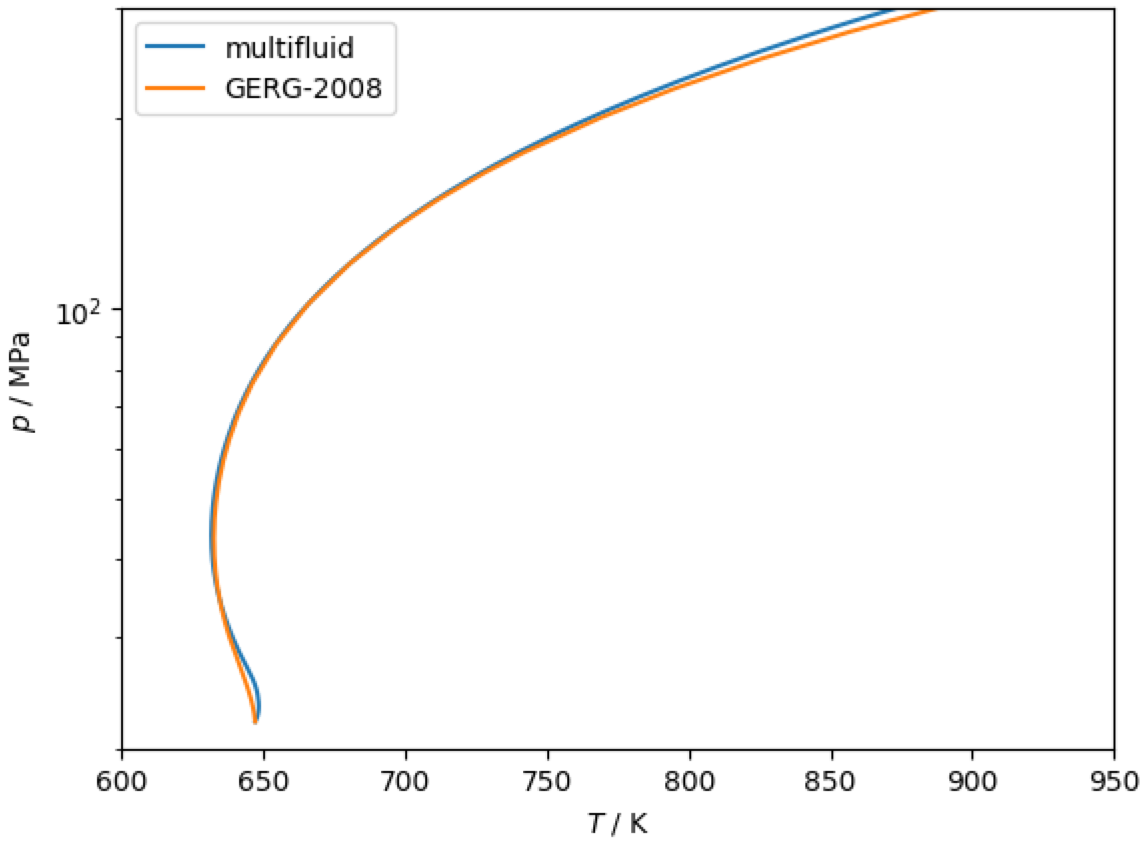
(continued from previous page)

```
    o.max_dt = 100 # cap the size of the allowed step
    o.calc_stability = True
    o.polish = True
    o.init_c = init_c # You might need to swap the initial tracing direction by␣
↪making this +1.0
    curveJSON = model.trace_critical_arclength_binary(T0, rhovec0, '', 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

# Tracing with multi-fluid from an endpoint with non-analytic terms
model = teqp.build_multifluid_model(["Water", "Methane"], teqp.get_datapath())

x0 = 1-1e-6 # ever so slightly away from the pure fluid
molefrac = np.array([x0, 1-x0])

# Solve for the actual critical point at this mole fraction with scipy
y0 = [model.get_Tcvec()[0], 1/model.get_vcvec()[0]]
residual = lambda y: model.get_criticality_conditions(y[0], y[1]*molefrac)
res = scipy.optimize.fsolve(residual, y0)
T = res[0]
rho0 = res[1]
rhovec0 = rho0*molefrac

# Now trace from this point
curve = get_critical_curve_composition(model, T0=T, rhovec0=rhovec0)
plt.plot(curve['T / K'], curve['p / Pa']/1e6, label='multifluid')

# With GERG-2008, things are much more straightforward...
model = teqp.make_model({'kind': 'GERG2008resid', 'model': {'names': ['water','methane
↪']}})

def get_critical_curve_simple(model, ipure, T0, rho0):
    """ Trace from a pure fluid... """
    rhovec0 = np.array([0, 0])
    rhovec0[ipure] = rho0
    o = teqp.TCABOptions()
    o.init_dt = 1.0 # step in the arclength tracing parameter
    o.rel_err = 1e-8
    o.abs_err = 1e-5
    o.integration_order = 5
    o.calc_stability = True
    o.polish = True
    curveJSON = model.trace_critical_arclength_binary(T0, rhovec0, '', 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

for ifluid in [0]:
    Tci = model.get_Tcvec()[ifluid]
    vci = model.get_vcvec()[ifluid]
    df = get_critical_curve_simple(model, ipure=ifluid, T0=Tci, rho0 = 1.0/vci)
    plt.plot(df['T / K'], df['p / Pa']/1e6, label='GERG-2008')

plt.gca().set(xlabel='$T$ / K', ylabel='$p$ / MPa')
```

(continues on next page)

(continued from previous page)

```
plt.yscale('log')
plt.xlim(600, 950)
plt.ylim(20, 300)
plt.legend(loc='best');
```



5.6 Information

The algorithms are written in a very generic way; they take an instance of a thermodynamic model, and the necessary derivatives are calculated from this model with automatic differentiation (or similar). In that way, implementing a model is all that is required to enable its use in the calculation of critical curves or to trace the phase equilibria. Determining the starting values, on the other hand, may require model-specific assistance, for instance with superancillary equations.

CHAPTER

SIX

EXAMPLES

6.1 The teqp paper in I&ECR

A few minor changes have been made:

- The `get_splus` method requires the molar concentrations to be a numpy array (to avoid copies) (as of version 0.14.0
- The top-level methods `teqp.xxx` have been deprecated, and the methods attached to the instance are preferred
- The `radial_dist` field must always be provided

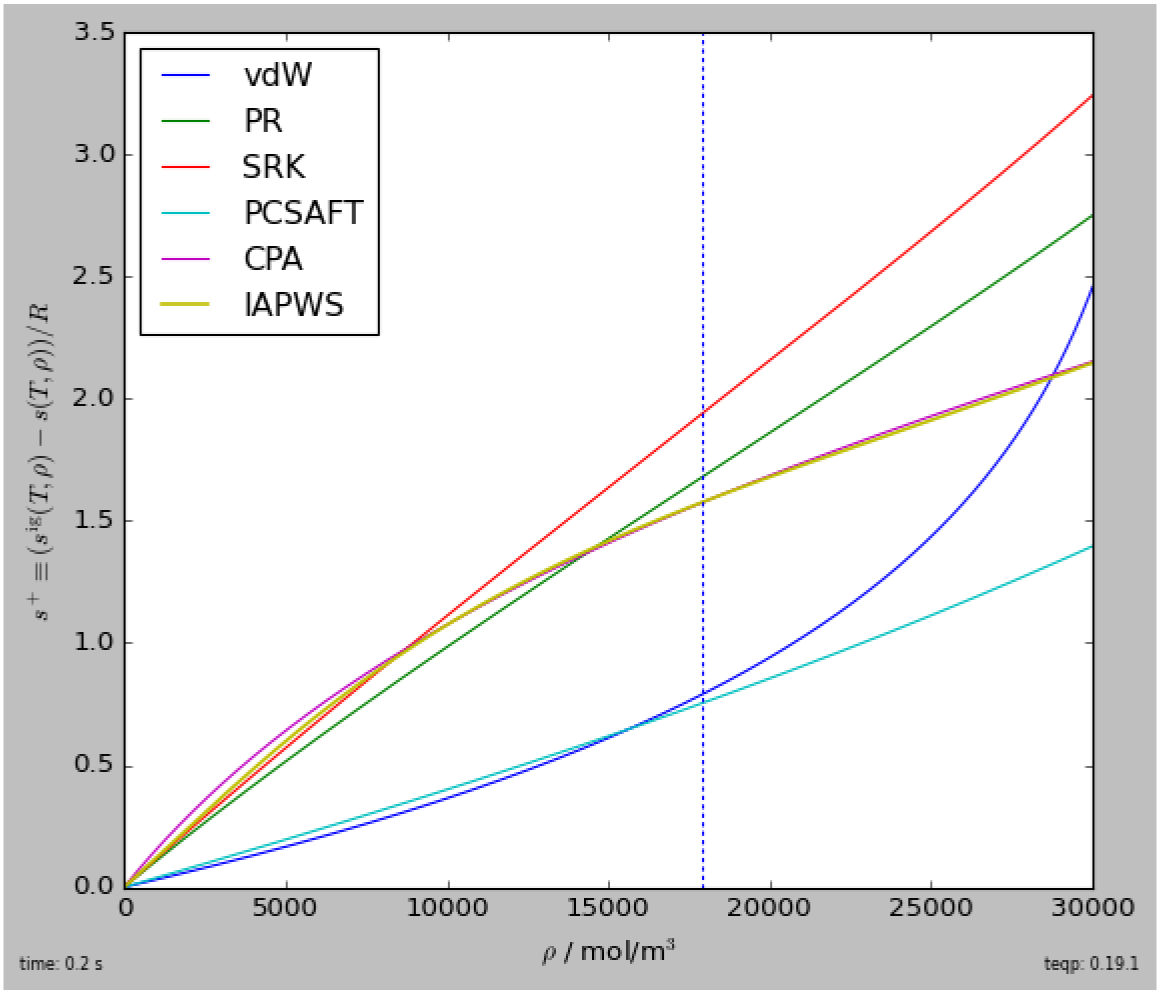
[1]:

```
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, "radial_dist
↪": "CS"}

    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))
```

(continues on next page)

(continued from previous page)

```
    ]

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([model.get_splus(T, np.array([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'$s^+\equiv (s^{\rm ig}(T,\rho)-
↪s(T,\rho))/R$')
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.show()
```



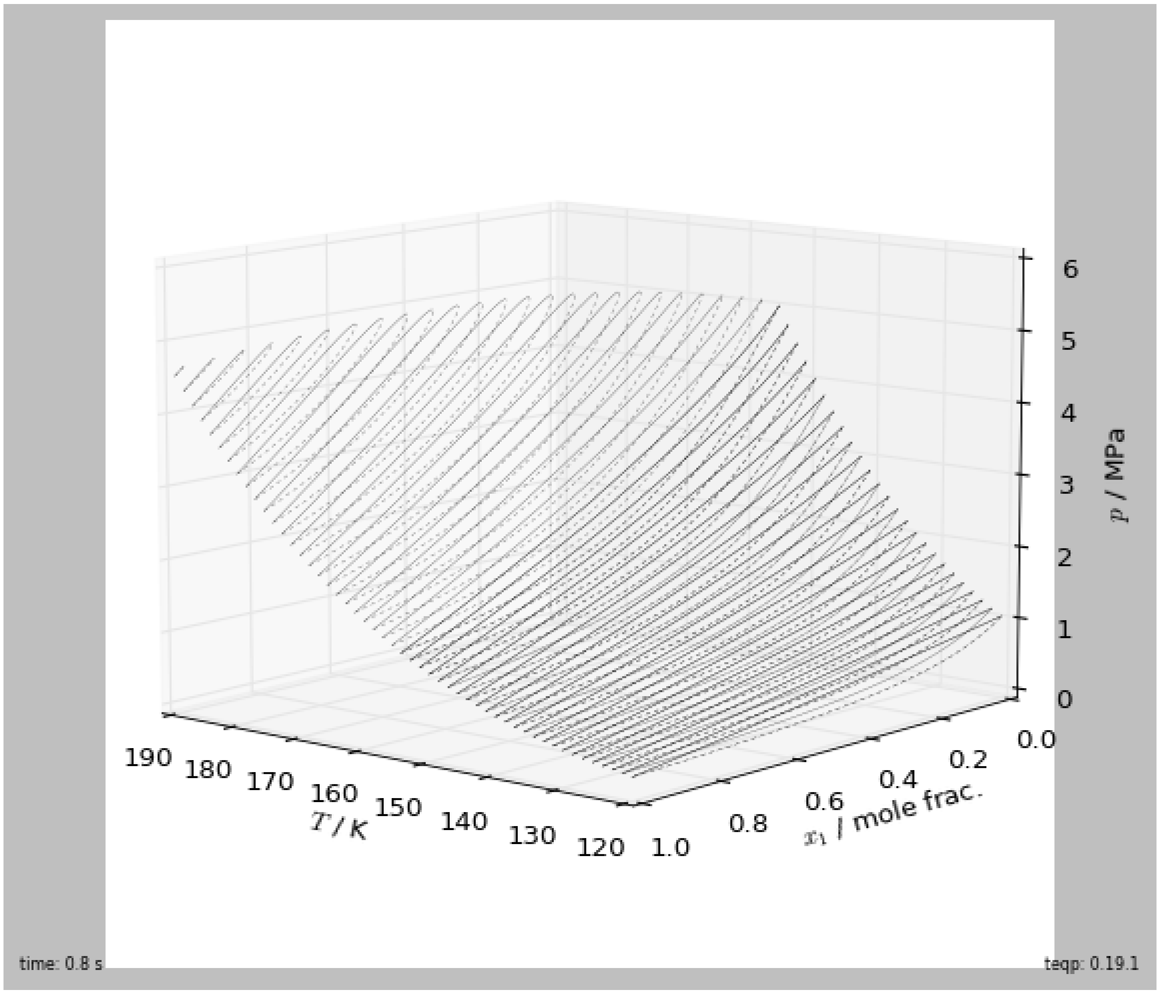
[2]:

```
import json, timeit
```

(continues on next page)

(continued from previous page)

```
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(model.trace_VLE_isotherm_binary(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()
```



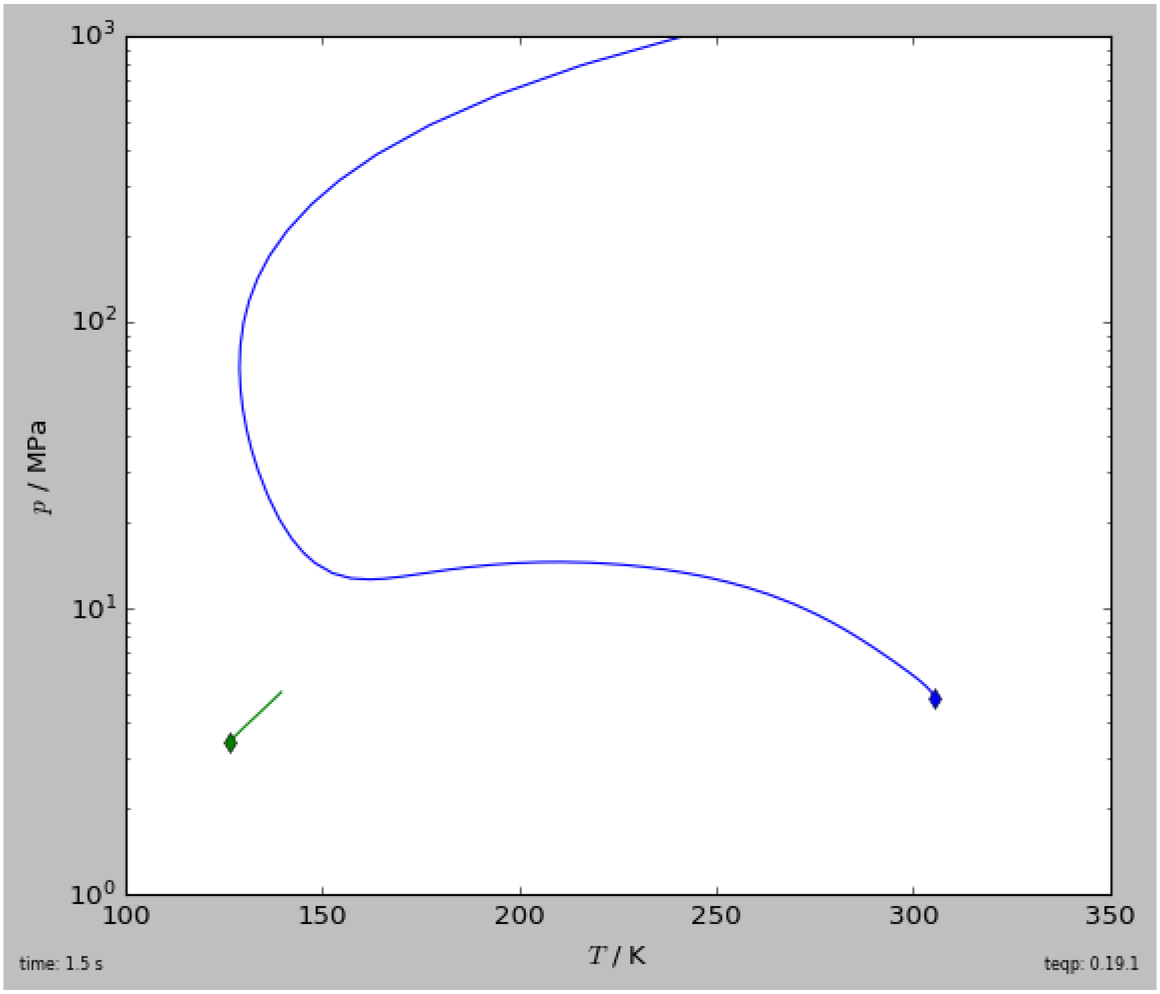
[3]:

```
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 = model.trace_critical_arclength_binary(T0, rho0, '', o)
```

(continues on next page)

(continued from previous page)

```
    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.show()
```


10^3
10^2
10^1
10^0
p / MPa
100
150
200
250
300
350
T / K
time: 1.5 s
teqp: 0.19.1

CHAPTER
SEVEN

FITTING

7.1 Multi-fluid Parameter Fitting

Here is an example of fitting the β_T and γ_T values for the binary pair of propane+n-dodecane with the multi-fluid model. It uses differential evolution to do the global optimization, which is probably overkill in this case as the problem is 2D and other algorithms like Nelder-Mead or even approximate Hessian methods would probably be fine.

In any case, it takes a few seconds to run (when the actual optimization is uncommented), demonstrating how one can fit model parameters with existing tooling from the scientific python stack.

[1]:
```
import json
import teqp, numpy as np, pandas, matplotlib.pyplot as plt
import scipy.interpolate, scipy.optimize

import pandas
data = pandas.read_csv('VLE_data_propane_dodecane.csv')
```

[2]:
```
def cost_function(parameters:np.ndarray, plot:bool=False):

    # Fitting some parameters and fixing the others
    betaV, gammaV = 1.0, 1.0
    betaT, gammaT = parameters

    # betaT, gammaT, betaV, gammaV = parameters

    BIP = [{
        'function': '',
        'BibTeX': 'thiswork',
        'CAS1': '112-40-3',
        'CAS2': '74-98-6',
        'F': 0.0,
        'Name1': 'n-Dodecane',
        'Name2': 'n-Propane',
        'betaT': betaT,
        'betaV': betaV,
        'gammaT': gammaT,
        'gammaV': gammaV
    }]
    model = teqp.build_multifluid_model(["n-Dodecane", "n-Propane"], teqp.get_
↪datapath(),
        BIPcollectionpath=json.dumps(BIP)
    )
    ancs = [model.build_ancillaries(ipure) for ipure in [0,1]]
```

(continues on next page)

(continued from previous page)

```
    cost = 0.0

    # The 0-based index of the fluid to start from. At this temperature, only one␣
↪fluid
    # is subcritical, so it has to be that one, but in general you could start
    # from either one.
    ipure = 0

    for T in [419.15, 457.65]:
        # Subset the experimental data to match the isotherm
        # being fitted
        dfT = data[np.abs(data['T / K90'] - T) < 1e-3]

        if plot:
            plt.plot(1-dfT['x[0] / mole frac.'], dfT['p / Pa']/1e6, 'X')
            plt.plot(1-dfT['y[0] / mole frac.'], dfT['p / Pa']/1e6, 'X')

        try:
            # Get the molar concentrations of the pure fluid
            # at the starting point
            anc = ancs[ipure]
            rhoL0 = np.array([0, 0.0])
            rhoV0 = np.array([0, 0.0])
            rhoL0[ipure] = anc.rhoL(T)
            rhoV0[ipure] = anc.rhoV(T)

            # Now we do the trace and convert retuned JSON
            # data into a DataFrame
            df = pandas.DataFrame(model.trace_VLE_isotherm_binary(T, rhoL0, rhoV0))

            if plot:
                plt.plot(df['xL_0 / mole frac.'], df['pL / Pa']/1e6)
                plt.plot(df['xV_0 / mole frac.'], df['pL / Pa']/1e6)

            # Interpolate trace at experimental pressures along this
            # isotherm to get composition from the current model
            # The interpolators are set up to put in NaN for out
            # of range values
            x_interpolator = scipy.interpolate.interp1d(
                df['pL / Pa'], df['xL_0 / mole frac.'],
                fill_value=np.nan, bounds_error=False
            )
            y_interpolator = scipy.interpolate.interp1d(
                df['pL / Pa'], df['xV_0 / mole frac.'],
                fill_value=np.nan, bounds_error=False
            )
            # The interpolated values for the compositions
            # along the trace at experimental pressures
            x_model = x_interpolator(dfT['p / Pa'])
            y_model = y_interpolator(dfT['p / Pa'])
            if plot:
                plt.plot(x_model, dfT['p / Pa']/1e6, '.')

            # print(x_model, (1-dfT['x[0] (-)']))

            errTx = np.sum(np.abs(x_model-(1-dfT['x[0] / mole frac.'])))
```

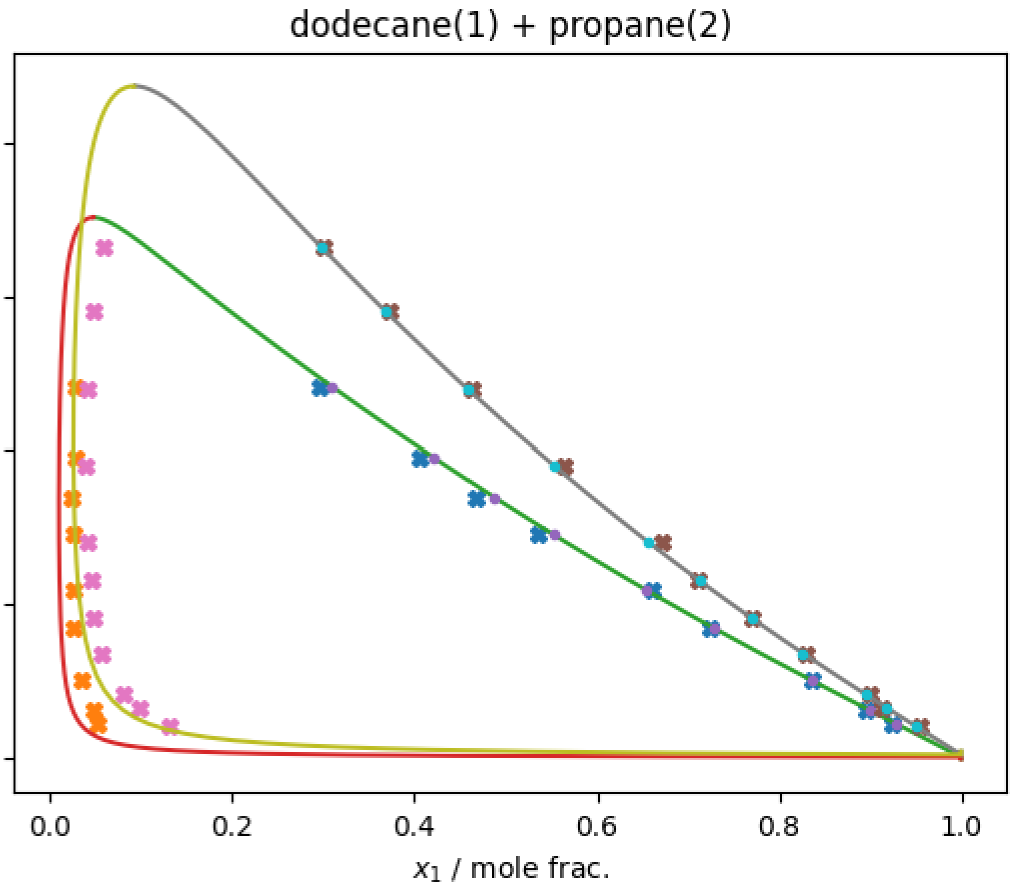
(continues on next page)

(continued from previous page)

```
            errTy = np.sum(np.abs(y_model-(1-dfT['y[0] / mole frac.'])))

            # If any point *cannot* be interpolated, throw out the model,
            # returning a large cost function value.
            #
            # Note: you might need to be more careful here,
            # if the points are close to the critical point, a good model might
            # (but not usually), undershoot the critical point of the
            # real mixture
            #
            # Also watch out for values of compositons in the data that are␣
↪placeholders
            # with a value of nan, which will pollute the error calculation
            if not np.isfinite(errTx):
                return 1e6
            if not np.isfinite(errTy):
                return 1e6
            cost += errTx + errTy

        except BaseException as BE:
            print(BE)
            pass
    if plot:
        plt.title(f'dodecane(1) + propane(2)')
        plt.xlabel('$x_1$ / mole frac.'); plt.ylabel('$p$ / MPa')
        plt.savefig('n-Dodecane+propane.pdf')
        plt.show()

    return cost
```

[3]:

```
# The final parameter values, will be overwritten if
# optimization call is uncommented
x = [1.01778992, 1.17318854]

# Here is the code used to do the optimization, uncomment to run it
# Note: it is commented out because it takes too long to run on doc builder
#
# res = scipy.optimize.differential_evolution(
#     cost_function,
#     bounds=((0.9, 1.5), (0.75, 1.5)),
#     disp=True,
#     polish=False
# )
# print(res)
# x = res.x

cost_function(x, plot=True)
```



```
[3]: 0.47041664920218196
```

CHAPTER

EIGHT

TEQP

8.1 teqp package

8.1.1 Submodules

8.1.2 teqp.teqp module

TEQP: Templated Equation of State Package

class `teqp.teqp.`**`AbstractModel`**

Bases: `pybind11_object`

`build_Psi_Hessian_autodiff`(*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, n]]

`build_Psir_Hessian_autodiff`(*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, n]]

`build_Psir_gradient_autodiff`(*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

`build_d2PsirdTdrhoi_autodiff`(*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

`dpsatdT_pure`(*self:* teqp.teqp.AbstractModel, *T: float*, *rhoL: float*, *rhoV: float*) → float

`eigen_problem`(*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*, *alignment_v0: numpy.ndarray[numpy.float64[m, 1]] | None = None*) → teqp::EigenData

`extrapolate_from_critical`(*self:* teqp.teqp.AbstractModel, *Tc: float*, *rhoc: float*, *T: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]] | None = None*) → numpy.ndarray[numpy.float64[2, 1]]

`find_VLLE_T_binary`(*self:* teqp.teqp.AbstractModel, *traces: List[json]*, *options:* teqp.teqp.VLLEFinderOptions | *None = None*) → List[json]

`find_VLLE_p_binary`(*self:* teqp.teqp.AbstractModel, *traces: List[json]*, *options:* teqp.teqp.VLLEFinderOptions | *None = None*) → List[json]

get_ATrhoXi(*self: teqp.teqp.AbstractModel*, *T: float*, *NT: int*, *rhomolar: float*, *Nrho: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*, *i: int*, *NXi: int*) → float

get_ATrhoXiXj(*self: teqp.teqp.AbstractModel*, *T: float*, *NT: int*, *rhomolar: float*, *Nrho: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*, *i: int*, *NXi: int*, *j: int*, *NXj: int*) → float

get_ATrhoXiXjXk(*self: teqp.teqp.AbstractModel*, *T: float*, *NT: int*, *rhomolar: float*, *Nrho: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*, *i: int*, *NXi: int*, *j: int*, *NXj: int*, *k: int*, *NXk: int*) → float

get_Ar00(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar00n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar01(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar01n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar02(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar02n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar03(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar03n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar04(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar04n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar05n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar06n(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_Ar10(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar11(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar12(*self: teqp.teqp.AbstractModel*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar13 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar14 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar20 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar21 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar22 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar23 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Ar24 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Arxy (*self:* teqp.teqp.AbstractModel, *NT: int*, *ND: int*, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_AtaudeltaXi (*self:* teqp.teqp.AbstractModel, *tau: float*, *Ntau: int*, *delta: float*, *Ndelta: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*, *i: int*, *NXi: int*) → float

get_AtaudeltaXiXj (*self:* teqp.teqp.AbstractModel, *tau: float*, *Ntau: int*, *delta: float*, *Ndelta: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*, *i: int*, *NXi: int*, *j: int*, *NXj: int*) → float

get_AtaudeltaXiXjXk (*self:* teqp.teqp.AbstractModel, *tau: float*, *Ntau: int*, *delta: float*, *Ndelta: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*, *i: int*, *NXi: int*, *j: int*, *NXj: int*, *k: int*, *NXk: int*) → float

get_B12vir (*self:* teqp.teqp.AbstractModel, *T: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_B2vir (*self:* teqp.teqp.AbstractModel, *T: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_Bnvir (*self:* teqp.teqp.AbstractModel, *Nderiv: int*, *T: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → Dict[int, float]

get_R (*self:* teqp.teqp.AbstractModel, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_chempotVLE_autodiff (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_criticality_conditions (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[2, 1]]

get_dchempotdT_autodiff (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_deriv_mat2 (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[3, 3]]

get_dmBnvirdTm (*self:* teqp.teqp.AbstractModel, *Nderiv: int*, *NTderiv: int*, *T: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_dp_dT_crit (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_dpsat_dTsat_isopleth (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecL: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_drhovec_dT_crit (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_drhovecdT_psat (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecL: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV: numpy.ndarray[numpy.float64[m, 1]]*) → Tuple[numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1]]]

get_drhovecdp_Tsat (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecL: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV: numpy.ndarray[numpy.float64[m, 1]]*) → Tuple[numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1]]]

get_fugacity_coefficients (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_minimum_eigenvalue_Psi_Hessian (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_neff (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_partial_molar_volumes (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → numpy.ndarray[numpy.float64[m, 1]]

get_pr (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → float

get_pure_critical_conditions_Jacobian (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *alternative_pure_index: int | None = None*, *alternative_length: int | None = None*) → Tuple[numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, n]]]

get_splus (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovec: numpy.ndarray[numpy.float64[m, 1]]*) → float

mix_VLE_Tp (*self:* teqp.teqp.AbstractModel, *T: float*, *p_given: float*, *rhovecL0: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV0: numpy.ndarray[numpy.float64[m, 1]]*, *options:* teqp.teqp.MixVLETpFlags | *None = None*) → *teqp.teqp.MixVLEReturn*

mix_VLE_Tx (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecL0: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV0: numpy.ndarray[numpy.float64[m, 1]]*, *xspec: numpy.ndarray[numpy.float64[m, 1]]*, *atol: float*, *reltol: float*, *axtol: float*, *relxtol: float*, *maxiter: int*) → Tuple[*teqp.teqp.VLE_return_code*, numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1]]]

mix_VLLE_T (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecVinit: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecL1init: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecL2init: numpy.ndarray[numpy.float64[m, 1]]*, *atol: float*, *reltol: float*, *axtol: float*, *relxtol: float*, *maxiter: int*) → Tuple[teqp::VLLE::VLLE_return_code, numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1]]]

mixture_VLE_px (*self:* teqp.teqp.AbstractModel, *p_spec: float*, *xmolar_spec: numpy.ndarray[numpy.float64[m, 1]]*, *T0: float*, *rhovecL0: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV0: numpy.ndarray[numpy.float64[m, 1]]*, *options:* teqp.teqp.MixVLEpxFlags | *None = None*) → Tuple[*teqp.teqp.VLE_return_code*, float, numpy.ndarray[numpy.float64[m, 1]], numpy.ndarray[numpy.float64[m, 1]]]

pure_VLE_T (*self:* teqp.teqp.AbstractModel, *T: float*, *rhoL: float*, *rhoV: float*, *max_iter: int*, *molefrac: numpy.ndarray[numpy.float64[m, 1]]* | *None = None*) → numpy.ndarray[numpy.float64[2, 1]]

solve_pure_critical (*self:* teqp.teqp.AbstractModel, *T: float*, *rho: float*, *flags: json* | *None = None*) → Tuple[float, float]

trace_VLE_isobar_binary (*self:* teqp.teqp.AbstractModel, *p: float*, *T0: float*, *rhovecL0: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV0: numpy.ndarray[numpy.float64[m, 1]]*, *options:* teqp.teqp.PVLEOptions | *None = None*) → json

trace_VLE_isotherm_binary (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecL0: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecV0: numpy.ndarray[numpy.float64[m, 1]]*, *options:* teqp.teqp.TVLEOptions | *None = None*) → json

trace_VLLE_binary (*self:* teqp.teqp.AbstractModel, *T: float*, *rhovecV: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecL1: numpy.ndarray[numpy.float64[m, 1]]*, *rhovecL2: numpy.ndarray[numpy.float64[m, 1]]*, *options:* teqp.teqp.VLLETracerOptions | *None = None*) → json

trace_critical_arclength_binary (*self:* teqp.teqp.AbstractModel, *T0: float*, *rhovec0: numpy.ndarray[numpy.float64[m, 1]]*, *path: str* | *None = None*, *options:* teqp.teqp.TCABOptions | *None = None*) → json

class teqp.teqp.**IterationMatrices**

Bases: `pybind11_object`

property J

property v

property vars

class teqp.teqp.**MixVLEReturn**

Bases: `pybind11_object`

property T

property initial_r

property message

property num_fev

property num_iter

property r

property return_code

property rhovecL

property rhovecV

property success

class `teqp.teqp.`**MixVLETpFlags**

Bases: `pybind11_object`

property atol

property axtol

property maxiter

property reltol

property relxtol

class `teqp.teqp.`**MixVLEpxFlags**

Bases: `pybind11_object`

property atol

property axtol

property maxiter

property reltol

property relxtol

class `teqp.teqp.`**MultiFluidVLEAncillaries**

Bases: `pybind11_object`

property pL

property pV

property rhoL

property rhoV

class `teqp.teqp.`**NRIterator**

Bases: `pybind11_object`

get_T(*self:* teqp.teqp.NRIterator) → float

get_molefrac(*self:* teqp.teqp.NRIterator) → numpy.ndarray[numpy.float64[m, 1]]

get_rho(*self:* teqp.teqp.NRIterator) → float

get_vals(*self:* teqp.teqp.NRIterator) → numpy.ndarray[numpy.float64[m, 1]]

get_vars (*self:* teqp.teqp.NRIterator) → List[str]

take_step (*self:* teqp.teqp.NRIterator) → numpy.ndarray[numpy.float64[m, 1]]

take_steps (*self:* teqp.teqp.NRIterator, *arg0: int*) → None

class `teqp.teqp.`**PVLEOptions**

Bases: `pybind11_object`

property abs_err

property calc_criticality

property crit_termination

property init_c

property init_dt

property integration_order

property max_dt

property max_steps

property polish

property polish_exception_on_fail

property polish_reltol_rho

property rel_err

property terminate_unstable

property verbosity

class `teqp.teqp.`**SAFTCoeffs**

Bases: `pybind11_object`

property BibTeXKey

property Qstar2

property epsilon_over_k

property m

property mustar2

property nQ

property name

property nmu

property sigma_Angstrom

class `teqp.teqp.`**TCABOptions**

Bases: `pybind11_object`

property T_tol

property abs_err

property calc_stability

property init_c

property init_dt

property integration_order

property max_dt

property max_step_count

property polish

property polish_exception_on_fail

property polish_reltol_T

property polish_reltol_rho

property pure_endpoint_polish

property rel_err

property skip_dircheck_count

property small_T_count

property stability_rel_drho

property verbosity

class `teqp.teqp.`**TVLEOptions**

Bases: `pybind11_object`

property abs_err

property calc_criticality

property crit_termination

property init_c

property init_dt

property integration_order

property max_dt

property max_steps

property p_termination

property polish

property polish_exception_on_fail

property polish_reltol_rho

property rel_err

property terminate_unstable

property verbosity

class `teqp.teqp.`**VLEAncillary**

Bases: `pybind11_object`

property T_r

property Tmax

property Tmin

class `teqp.teqp.`**VLE_return_code**

Bases: `pybind11_object`

Members:

unset

xtol_satisfied

functol_satisfied

maxiter_met

maxfev_met

notfinite_step

functol_satisfied = <VLE_return_code.functol_satisfied: 2>

maxfev_met = <VLE_return_code.maxfev_met: 3>

maxiter_met = <VLE_return_code.maxiter_met: 4>

property name

notfinite_step = <VLE_return_code.notfinite_step: 5>

unset = <VLE_return_code.unset: 0>

property value

xtol_satisfied = <VLE_return_code.xtol_satisfied: 1>

class `teqp.teqp.`**VLLEFinderOptions**

Bases: `pybind11_object`

property max_steps

property rho_trivial_threshold

class `teqp.teqp.`**VLLETracerOptions**

Bases: `pybind11_object`

property T_limit

property abs_err

property init_dT

property max_dT

property max_polish_steps

property max_step_count

property max_step_retries

property polish

property rel_err

property terminate_composition

property terminate_composition_tol

property verbosity

teqp.teqp.**attach_model_specific_methods**(*arg0: object*) → None

teqp.teqp.**build_alias_map**(*root: str*) → Dict[str, str]

teqp.teqp.**build_ancillaries**(*model:* teqp.teqp.AbstractModel, *Tc: float*, *rhoc: float*, *Tmin: float*, *flags: json | None = None*) → *teqp.teqp.MultiFluidVLEAncillaries*

teqp.teqp.**collect_component_json**(*identifiers: List[str]*, *root: str*) → List[json]

teqp.teqp.**convert_CoolProp_idealgas**(*arg0: str*, *arg1: int*) → json

teqp.teqp.**convert_FLD**(*component: str*, *name: str*) → json

teqp.teqp.**convert_HMXBNC**(*path: str*) → Tuple[json, json]

teqp.teqp.**get_BIPdep**(*BIPcollection: json*, *identifiers: List[str]*, *flags: json = None*) → Tuple[json, bool]

teqp.teqp.**get_departure_json**(*name: str*, *root: str*) → json

8.1.3 Module contents

teqp.**AmmoniaWaterTillnerRoth**()

teqp.**CPAfactory**(*spec*)

teqp.**IdealHelmholtz**(*model*)

teqp.**PCSAFTEOS**(*coeffs*, *kmat=None*)

teqp.**build_LJ126_TholJPCRD2016**()

teqp.**build_Psi_Hessian_autodiff**(*model*, **args*, ***kwargs*)

teqp.**build_Psir_Hessian_autodiff**(*model*, **args*, ***kwargs*)

teqp.**build_Psir_gradient_autodiff**(*model*, **args*, ***kwargs*) `teqp.build_d2PsirdTdrhoi_autodiff(model, *args, **kwargs)`

`teqp.build_multifluid_ecs_mutant(*args, **kwargs)`

`teqp.build_multifluid_model(components, coolprop_root, BIPcollectionpath='', flags={}, departurepath='')`

`teqp.build_multifluid_mutant(*args, **kwargs)`

`teqp.canonical_PR(Tc_K, pc_Pa, acentric, kmat=None)`

`teqp.canonical_SRK(Tc_K, pc_Pa, acentric, kmat=None)`

`teqp.deprecated_caller(model, *args, **kwargs)`

`teqp.eigen_problem(model, *args, **kwargs)`

`teqp.extrapolate_from_critical(model, *args, **kwargs)`

`teqp.find_VLLE_T_binary(model, *args, **kwargs)`

`teqp.get_B2virget_B12vir(model, *args, **kwargs)`

`teqp.get_chempotVLE_autodiff(model, *args, **kwargs)`

`teqp.get_criticality_conditions(model, *args, **kwargs)`

`teqp.get_datapath()`

Get the absolute path to the folder containing the root of multi-fluid data

`teqp.get_dchempotdT_autodiff(model, *args, **kwargs)`

`teqp.get_dpsat_dTsat_isopleth(model, *args, **kwargs)`

`teqp.get_drhovec_dT_crit(model, *args, **kwargs)`

`teqp.get_drhovecdT_psat(model, *args, **kwargs)`

`teqp.get_drhovecdp_Tsat(model, *args, **kwargs)`

`teqp.get_fugacity_coefficients(model, *args, **kwargs)`

`teqp.get_minimum_eigenvalue_Psi_Hessian(model, *args, **kwargs)`

`teqp.get_partial_molar_volumes(model, *args, **kwargs)`

`teqp.get_pr(model, *args, **kwargs)`

`teqp.get_pure_critical_conditions_Jacobian(model, *args, **kwargs)`

`teqp.get_splus(model, *args, **kwargs)`

`teqp.make_model(*args, **kwargs)`

This function is in two parts; first the make_model function (renamed to _make_model in the Python interface) is used to make the model and then the model-specific methods are attached to the instance

`teqp.make_vdW1(a, b)`

`teqp.mix_VLE_Tx(model, *args, **kwargs)`

`teqp.mix_VLLE_T(model, *args, **kwargs)` `teqp.`**`mixture_VLE_px`**(*model*, **args*, ***kwargs*)

`teqp.`**`pure_VLE_T`**(*model*, **args*, ***kwargs*)

`teqp.`**`solve_pure_critical`**(*model*, **args*, ***kwargs*)

`teqp.`**`tolist`**(*a*)

`teqp.`**`trace_VLE_isobar_binary`**(*model*, **args*, ***kwargs*)

`teqp.`**`trace_VLE_isotherm_binary`**(*model*, **args*, ***kwargs*)

`teqp.`**`trace_critical_arclength_binary`**(*model*, **args*, ***kwargs*)

`teqp.`**`vdWEOS`**(*Tc_K*, *pc_Pa*)

`teqp.`**`vdWEOS1`**(**args*)

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX