Table of Contents

Section 3.2 - Saturation solvers for pure substances

According to the Gibbs phase rule, we only have 1 degree of freedom along the saturation curves for a pure substance. This means that we should be able to specify either pressure or temperature and determine the corresponding saturated point.

This can be solved in 2 ways, either as an optimisation problem formulated in just pressure or chemical potential, or a root-finding problem using both equations. Generally root-finding problems are easier to solve numerically, so that formulation is generally preferable.

Specification of equilibrium

We know at equilibrium we have the equivalence of temperature, pressure, and chemical potential between phases. For vapour-liquid equilibrium this could be written as

$$T^{\text{vap}} = T^{\text{liq}} \tag{1}$$
$$p^{\text{vap}} = p^{\text{liq}} \tag{2}$$
$$\mu^{\text{vap}} = \mu^{\text{liq}}\,. \tag{3}$$

As we are iterating using two variables, we need to select two of the three equations to "complete" our problem. While any two could be used, solving for temperature is far more expensive than solving for pressure and chemical potential. This is because equations of state are generally not analytically solvable for temperature, meaning they would require an additional inner loop to solve for this. Therefore we solve for equilibrium using equations (1) and (3).

This is a root-finding problem in $\mathbb{R}^2$, with an objective function F defined as

$$F : \mathbb{R}^2 \to \mathbb{R}^2$$

$$F : \begin{bmatrix} V^{\text{liq}} \\ V^{\text{vap}} \end{bmatrix} \to \begin{bmatrix} T^{\text{liq}} - T^{\text{vap}} = 0 \\ \mu^{\text{liq}} - \mu^{\text{vap}} = 0 \end{bmatrix}.$$

Scaling factors

When solving our problem, properly scaling our equations is very important. Part of this is due to conditioning [1], and part due to how we define convergence. In a root-finding problem, convergence is usually declared when the magnitude of the function falls below a specified value, written as

$$|f(x)| < \epsilon$$

where ϵ is a **user-defined tolerance**. By using these scaling factors, our variables are similarly-sized, and the system of equations is better behaved. The typical scaling factors for the thermodynamic variables are given by the table below.

Typical scaling factors for thermodynamic variables

	Cubic	SAFT
Pressure	P_C	$R \cdot \epsilon / (N_A \cdot \sigma^3)$
Temperature	T_C	ϵ
Molar Energies (e.g. μ, g, a)	$R \cdot T$	$R \cdot T$

using Clapeyron, we can access the pressure and temperature scaling factors using

```
ps = p_scale(model)
Ts = T_scale(model)
```

Another technique often used is using logs to scale variables scanning multiple orders of magnitude. This frequently comes up with volumes, with liquid and vapour volumes varying by up to 1000x.

Initial guesses

As with all numerical methods, good initial guesses are quite important. To obtain these, we can leverage the theory of corresponding states. This states that "all fluids, when compared at the same reduced temperature and reduced pressure, have approximately the same compressibility factor and all deviate from ideal gas behavior to about the same degree" [1]. This allows us to express the solution to the van der Waals EoS saturation curve in terms of **reduced variables**. We use a highly-accurate numerical approximation for this, with saturated volumes given by

$$v^{\text{sat. liq}} = 3b/c^{\text{sat. liq}}$$
$$v^{\text{sat. vap}} = 3b/c^{\text{sat. vap}}$$

where c is given by

$$c^{\text{sat. liq}} = 1 + 2(1-T_\text{r})^{1/2} + \frac{2}{5}(1-T_\text{r}) - \frac{13}{25}(1-T_\text{r})^{3/2} + 0.115(1-T_\text{r})^2$$

$$c^{\text{sat. vap}} = \begin{cases} 2(1 + \frac{2}{5}(1-T_\text{r}) + 0.161(1-T_\text{r})^2) - c^{\text{sat. liq}} & 0.25 < T_\text{r} \leq 1 \\ 2(\frac{3}{2} - \frac{4}{9}T_\text{r} - 0.15T_\text{r}^2) - c^{\text{sat. liq}} & 0 \leq T_\text{r} < 0.64 \end{cases}$$

When using this for an initial guess we take one final step by moving the initial guesses away from the van der Waals saturation curve. I chose to use a factor of 0.5 for the liquid volume and 2 for the vapour volume respectively. This is done to avoid the guesses falling *inside* the saturation curve of the fluid we're solving for, as that can lead to numerical instability and convergence issues. By choosing guesses that bracket, but don't lie too far from, the saturation curve, convergence is far more reliable.

The derivation can be seen in The van der Waals equation: analytical and approximate solutions, the code is implemented below:

vdW_saturation_volume

```
vdW_saturation_volume(model, T)
```

Returns the saturation curve for a van der Waals with an equivalent critical point for `model`. Uses a derivation from 10.1007/s10910-007-9272-4 for the approximate solution to the saturation curve.

```
"""
    vdW_saturation_volume(model, T)

Returns the saturation curve for a van der Waals with an equivalent critical point
for ```model```. Uses a derivation from 10.1007/s10910-007-9272-4 for the
approximate solution to the saturation curve.
"""
function vdW_saturation_volume(model, T)
    Tc, pc, vc = crit_pure(model)
    Tr = T/Tc

    # Valid for 0 ≤ Tᵣ ≤ 1
    cL = 1 + 2(1-Tr)^(1/2) + 2/5*(1-Tr) - 13/25*(1-Tr)^(3/2) + 0.115*(1-Tr)^2

    if 0.25 < Tr ≤ 1
        # Valid for 0.25 < Tr ≤ 1
        cG = 2*(1 + 2/5*(1-Tr) + 0.161*(1-Tr)^2) - cL
    elseif 0 ≤ Tr < 0.64
        # Valid for 0 ≤ Tr < 0.64
        cG = 2*(3/2 - 4/9*Tr - 0.15Tr^2) - cL
    else
        @error "Invalid reduced temperature, Tr = $Tr. Use a value between 0 and 1"
    end
    b = vdW_b(pc, Tc)
    vL = 3b/cL
    vG = 3b/cG
    return [0.5*vL, 2*vG]
end
```

Implementation

Now, lets implement our saturation solver. We have two functions - our objective function, or the funcation to be zeroed, and the solver.

sat_p_objective

```
sat_p_objective(model, T, V)
```

Defines the objective function for a pure saturation solver. Returns a vector `[f1, f2]`, where `f1` is the pressure equation, and `f2` is the chemical potential equation.

```
"""
    sat_p_objective(model, T, V)

Defines the objective function for a pure saturation solver. Returns a vector
```[f1, f2]```, where ```f1``` is the pressure equation, and ```f2``` is the
chemical potential equation.
"""
function sat_p_objective(model, T, V)
    # Unpack input array
    Vl, Vv = V

    # Define in-line equations for pressure and chemical potential
    p(V) = pressure(model, V, T)
    μ(V) = VT_chemical_potential(model, V, T)

    # Calculate the objective function
    f1 = (p(Vl) - p(Vv))/p_scale(model)
    f2 = (μ(Vl) - μ(Vv))/(R*T)
    return [f1, f2[1]]
end
```
```

solve_sat_p

```
solve_sat_p(model, T; V0 = vdW_saturation_volume(model, T), itersmax=100, ab
stol=1e-10)
```

Solves an equation of state for the saturation pressure at a given temperature using Newton's method. By default uses the solution to the van der Waals equation for initial guesses. Returns (psat, *Vliq*, *Vvap*)

```
"""
    solve_sat_p(model, T; V0 = vdW_saturation_volume(model, T), itersmax=100,
    abstol=1e-10)

Solves an equation of state for the saturation pressure at a given temperature
using Newton's method. By default uses the solution to the van der Waals equation
for initial guesses. Returns (psat, V_liq, V_vap)
"""
function solve_sat_p(model, T; V0 = vdW_saturation_volume(model, T), itersmax=100,
abstol=1e-10)
    # Objective function accepting a vector of volumes, R²→R²
    f(logV) = sat_p_objective(model, T, exp10.(logV))
    # function returning the Jacobian of our solution, R²→R²ˣ²
    Jf(logV) = ForwardDiff.jacobian(f, logV)

    logV0 = log10.(V0)
    logVold = 0.0
    logV = logV0
    fx = 1.0
    fx0 = f(logV0)
    iters = 0
    # Iterate until converged or the loop has reached the maximum number of
    iterations
    while (iters < itersmax && all(abs.(fx) .> abstol))
        Jfx = Jf(logV) # Calculate the jacobian
        fx = f(logV) # Calculate the value of f at V
        d = -Jfx\fx # Calculate the newton step
        logVold = logV # Store current iteration
        logV = logV .+ d # Take newton step
        iters += 1 # Increment our iteration counter
    end
    # Show a warning if the solver did not converge (uses short circuit evaluation
    rather than if statement)
    iters == itersmax && @warn "solver did not converge in $(iters)
    iterations\nfV=$(fx)"

    V = exp10.(logV)
    p_sat = pressure(model, V[1], T)
    return (p_sat, V[1], V[2])
end
```

Now, we just need to define a model and test out our solver!

▸ (2.44433e5, 0.00014626, 0.0116605)

```
begin
    # Specify our state
    cubic_model = PR(["hexane"])
    T = 373.15 # K
    # Solve the nonlinear system
    (p_sat, V_liq, V_vap) = solve_sat_p(cubic_model, T)
end
```

Solver Results

Temperature	373.15	K
Saturation pressure	244400.0	Pa
Liquid volume	1.46e-04	m^3
Vapour volume	1.17e-02	m^3

And we can compare it to the Clapeyron result

▸ (2.44433e5, 0.00014626, 0.0116605)

```
psat_Clapeyron, Vlsat_Clapeyron, Vvsat_Clapeyron = saturation_pressure(cubic_model, T)
```

true

```
p_sat ≈ psat_Clapeyron
```

true

```
V_liq ≈ Vlsat_Clapeyron
```

true

```
V_vap ≈ Vvsat_Clapeyron
```

Got it!

Splendid!

Building phase diagrams

Now we are able to determine the location of the saturation curve, how do we build up a graph of the entire phase boundary? We can call the solver we just wrote above again and again for different temperatures with the same initial guess, and for most cases it will probably converge. However, as before there are a particular number of difficulties near the critical point and the solver can become very sensitive to the initial guesses.

To help with this, we can reuse each previous result as the new guess to the solver. On top of being very important near the critical point, this technique is very important for speeding up the overall solver - the very good initial guesses obtained in this way allow for rapid convergence.

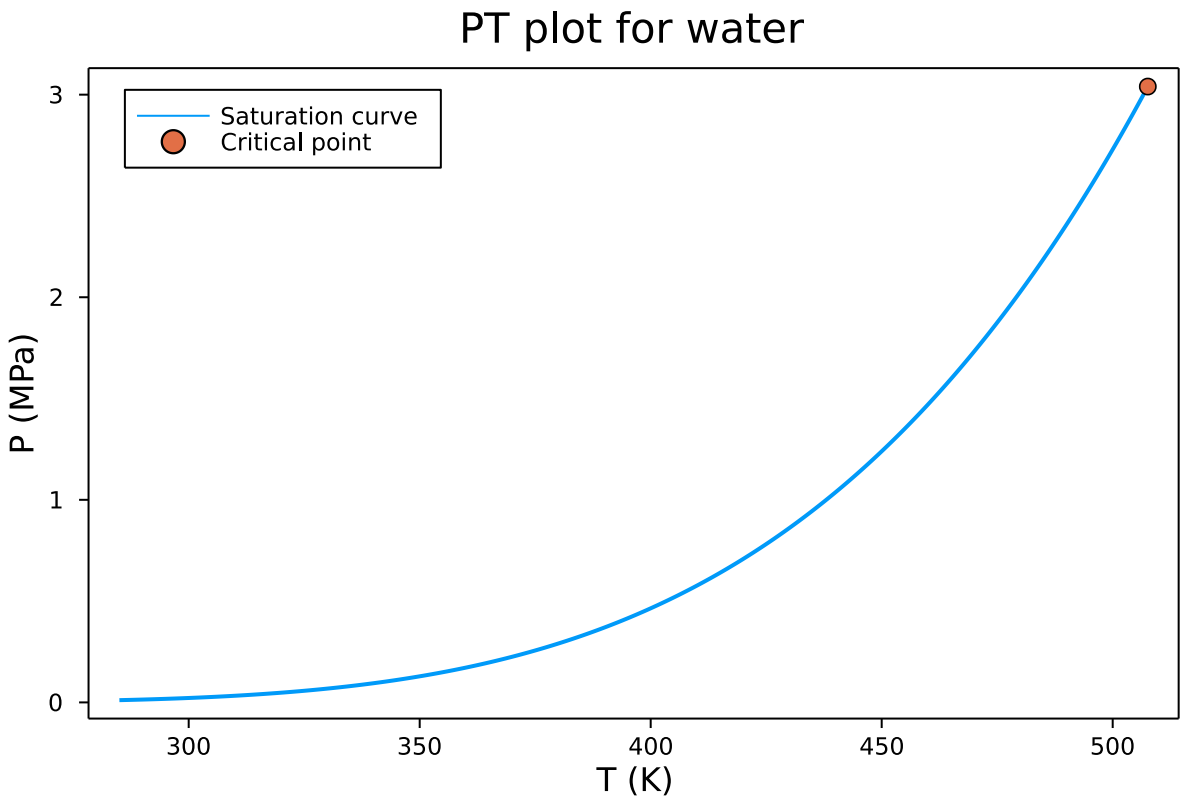
When building this, remember that the triple point is not represented by typical equations of state! As only liquid and vapour phases are captured, the only significant point we see on pure phase diagrams is the critical point.

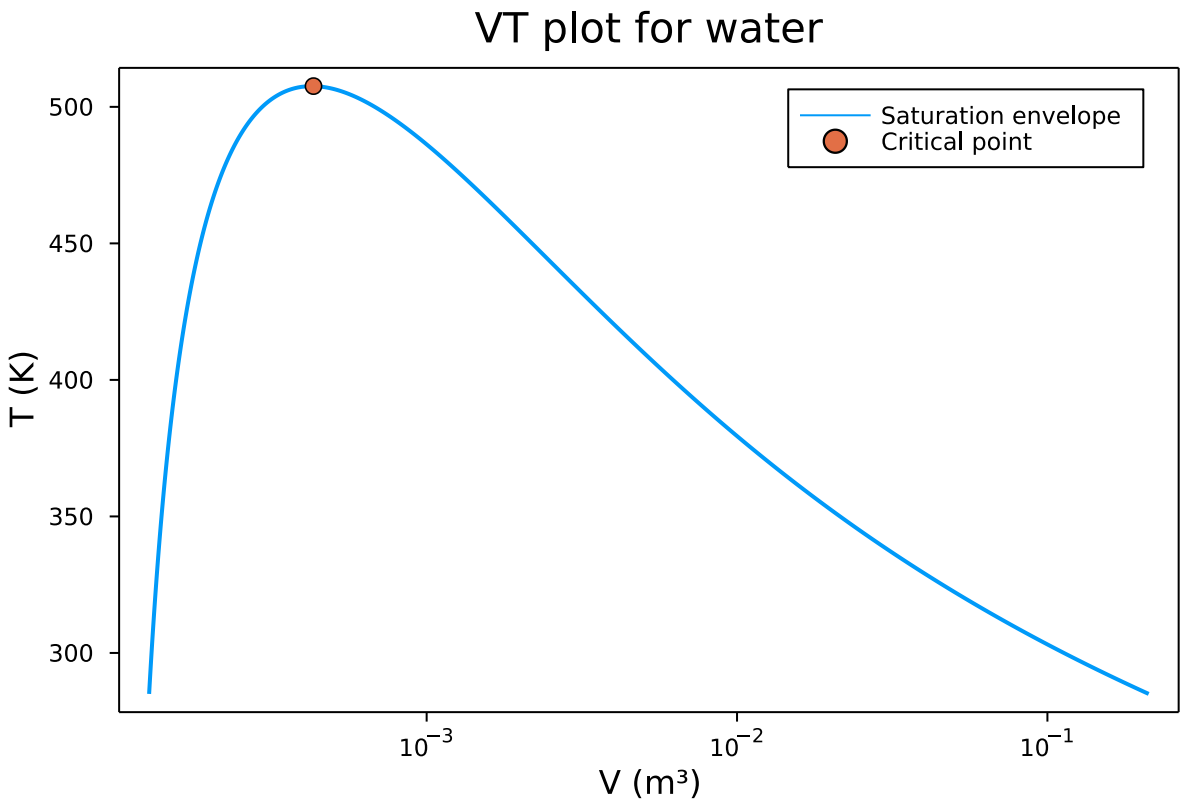
```
[[11099.7, 11218.4, 11338.1, 11458.9, 11580.7, 11703.6, 11827.6, 11952.7, 12078.9, ⋯ mo
```

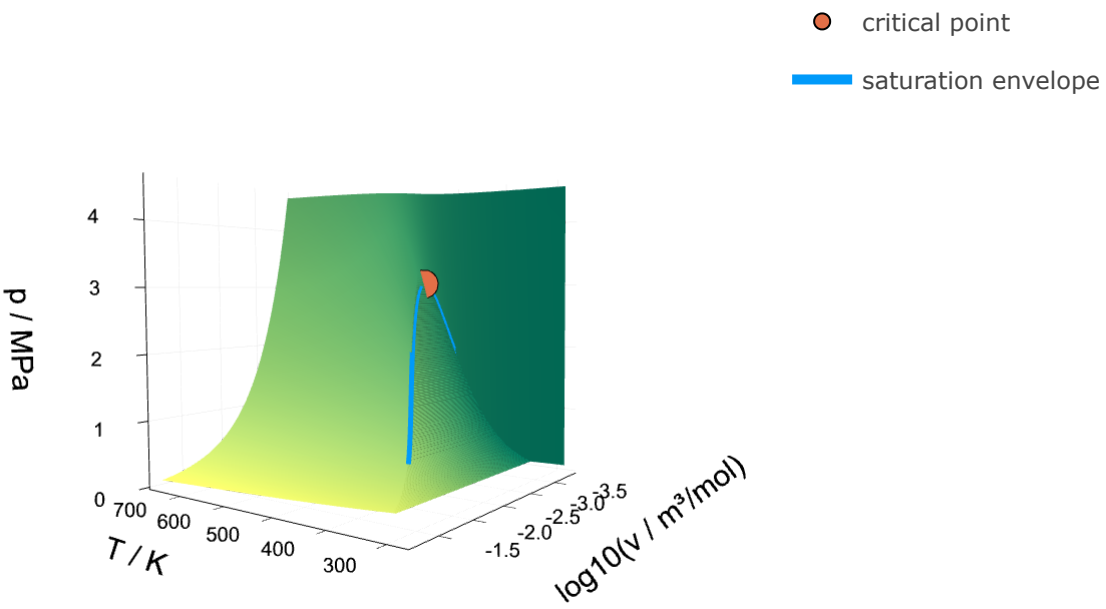
```
begin
    # The vector of temperature values
    crit_temp, _, _ = crit_pure(cubic_model)
    T_vec = range(285.0, 0.9999crit_temp, length=1000)
    # Preallocate our pressure and volume vectors
    p_vec = zeros(length(T_vec))
    Vl_vec = zeros(length(T_vec))
    Vv_vec = zeros(length(T_vec))

    # Create initial guess
    V0 = vdW_saturation_volume(cubic_model, T)
    for (i, T) in enumerate(T_vec)
        try
            sat = solve_sat_p(cubic_model, T; V0=V0)

            if ~any(isnan.(sat))
                (p_vec[i], Vl_vec[i], Vv_vec[i]) = sat
                V0 = [Vl_vec[i], Vv_vec[i]] # Store previous iteration for new guess
            end
        catch
            continue
        end
    end
    T_vec = T_vec[.!iszero.(p_vec)]
    filter!.(!iszero, [p_vec, Vl_vec, Vv_vec])
end
```


PT plot for water
Saturation curve
Critical point
P (MPa)
T (K)
0
1
2
3
300
350
400
450
500


VT plot for water
Saturation envelope
Critical point
T (K)
V (m³)
300
350
400
450
500
10^−3
10^−2
10^−1



Critical point solver

Formulation

Above, we traced the pure saturation envelope between two user-decided points. If one of these falls above the critical point, the saturation solver will either fail or converge to a trivial solution. If we want to determine the end point of our saturation curve before beginning calculation, how should we go about that?

For a cubic equation of state the critical temperature and pressure are an input to the EoS, meaning for a pure substance you will always know the critical point beforehand. This is not the case for other equations of state, meaning it must be solved for numerically.

The conditions for the critical point in a pure substance are:

$$f_1(v, T) = \left(\frac{\partial p(v,T)}{\partial v}\right)_T = 0$$
$$f_2(v, T) = \left(\frac{\partial^2 p(v,T)}{\partial v^2}\right)_T = 0$$

This would usually require analytical derivatives, as calculation with finite differences are both inaccurate and expensive, especially for higher order derivatives. However, automatic differentiation allows us to easily determine the exact derivatives in the objective function, as well as the higher order derivatives needed to solve this with Newton's method. Note that although solver methods that do not rely on derivatives exist, it is faster to use derivative information if it is available.

Initial guesses

The final issue we must resolve is the generation of **initial guesses**. Luckily, a critical point solver isn't too sensitive to initial guesses.

The volume guess is an empirically chosen packing fraction of $\mathbf{0.3}$. This corresponds to

$$V_0^{\text{liq}} = \frac{1}{0.3} \cdot \frac{\pi}{6} \cdot N_A \cdot \sigma^3$$

as defined in Section 3.1.

The temperature guess corresponds to

$$T_0 = 2 \cdot T^{\text{scale}}$$

where for SAFT,

$$T^{\text{scale}} = \epsilon \, .$$

Implementation

Rather than write the entire solver by hand, we will now rely on Newton's method exported by the NLsolve library. The format of this function is:

```
nlsolve(f!, initial_x, autodiff = :forward, method = :newton)
```

Note that **!** signifies an in-place function. This means that rather than returning a value, it updates the first value passed to the new values.

critical_objective!

```
critical_objective!(model, F, x)
```

Defines the objective function for a critical point solver. Returns a vector `[f1, f2]`, where `f1` is the first pressure derivative, and `f2` is the second pressure derivative.

```
"""
    critical_objective!(model, F, x)

Defines the objective function for a critical point solver. Returns a vector
```[f1, f2]```, where ```f1``` is the first pressure derivative, and ```f2``` is
the second pressure derivative.
"""
function critical_objective!(model, F, x)
    V = x[1]
    T = x[2]
    p(V) = pressure(model, V, T)/p_scale(model)
    f1(V) = ForwardDiff.derivative(p, V)
    f2(V) = ForwardDiff.derivative(f1, V)
    F .= [f1(V), f2(V)]
end
```

## critical_point_guess

```
critical_point_guess(model::SAFTModel)
```

Generates initial guesses for the critical point of a fluid. Scales the limit of the packing fraction and temperature scale factor using empirical factors.

```
"""
    critical_point_guess(model::SAFTModel)

Generates initial guesses for the critical point of a fluid. Scales the limit of
the packing fraction and temperature scale factor using empirical factors.
"""
function critical_point_guess(model::SAFTModel)
    σ = model.params.sigma.diagvalues[1]
    seg = model.params.segment.values[1]
    # η = 0.3
    V0 = 1/0.3 * 1.25 * π/6 * N_A * σ^3 * seg
    return [log10(V0), 2.0]
end
```
```

solve_critical_point

```
solve_critical_point(model)
```

Directly solves for the critical point of an equation of state using automatic differentiation and Newton's method.

```
"""
    solve_critical_point(model)

Directly solves for the critical point of an equation of state using automatic differentiation and Newton's method.
"""
function solve_critical_point(model)
    Ts = T_scale(model)
    # Generate initial guesses
    x0 = critical_point_guess(model)

    # Solve system for critical point
    res = nlsolve((F, x) -> critical_objective!(model, F, [exp10(x[1]), Ts*x[2]]), x0, autodiff = :forward, xtol=1e-9, method=:newton)

    # Extract answer
    Vc = exp10(res.zero[1])
    Tc = Ts*res.zero[2]
    # Calculate pressure
    pc = pressure(model, Vc, Tc)
    # Return values
    return (pc, Tc, Vc)
end
```

We can now use our function to calculate the critical point

```
(3.66201e7, 697.378, 5.43514e-5)
```

```
begin
    model_SAFT = PCSAFT(["water"])
    pc, Tc, Vc = solve_critical_point(model_SAFT)
end
```

Solver Results

Critical Temperature	697.38	K
Critical Volume	5.435e-5	m^3
Critical Pressure	36.62	MPa

As with the volume solver, we can compare this to the implementation within Clapeyron using the `crit_pure` function.

```
(Tc, pc, Vc) = crit_pure(model)
```

```
(697.378, 3.66201e7, 5.43514e-5)
```

```
Tc_Clapeyron, pc_Clapeyron, Vc_Clapeyron = crit_pure(model_SAFT)
```

```
true
```

```
Tc ≈ Tc_Clapeyron
```

```
true
```

```
pc ≈ pc_Clapeyron
```

```
true
```

```
Vc ≈ Vc_Clapeyron
```

Got it!

Keep it up!

And we've converged correctly on the right answer!

Footnotes

[1]:

Equation conditioning generally refers to the sensitivity of the output to small pertubations in the input. Often in numerical methods, a highly sensitive (i.e. poorly conditioned) problem can result in the loss of precision. The condition number is often discussed for both matrices and for functions.

[2]:

If you are using a model with volume translation, you should also account for that in your initial guess.