Table of Contents

Section 3.4 - The Flash Problem

The flash problem refers to knowing **when** a fluid will undergo a phase split, **how many phases** the fluid will split into, and the **composition** of each phase. We use these calculations all across chemical engineering, particularly in separation processes. For example, these are used when modelling flash drums, distillation columns and liquid-liquid extraction. They can also appear when simulating petroleum reservoirs and other complex liquid flow.

Key concepts

Before we begin, there are a few important ideas and equations we should mention.

Chemical equilibrium

At equilibrium, we have the **equivalence of chemical potential**, so that for a mixture with C components and P phases $(\alpha, \beta, \dots, P)$ we have an equivalence relation for every component *i* in *C*:

$$\mu_i^\alpha = \mu_i^\beta = \cdots = \mu_i^P$$

Identically, we have the **equivalent of fugacity**:

$$f_i^\alpha = f_i^\beta = \cdots = f_i^P$$

We can express the compositions at vapour liquid equilibrium (VLE) using **K factors**. These represent the distribution of each component between phases.

$$K_i = \frac{y_i}{x_i}$$

Using the equality of fugacity, we can also express this in terms of the fugacity coefficient

$$\begin{aligned} f_i^L &= x_i \varphi_i^L P \\ f_i^V &= y_i \varphi_i^V P \\ x_i \varphi_i^L \cancel{P} &= y_i \varphi_i^V \cancel{P} \\ K_i &= \frac{\varphi_i^L}{\varphi_i^V} \end{aligned}$$

The "trivial solution"

In flash problems we run the risk of converging to the so-called trivial solution. This is where each phase is identical, meaning the equality of chemical potential is inherently satisfied. In a situation where we know there should be a 2-phase region, it can be hard to avoid converging to this solution.

The Rachford-Rice equation

For the case where we have **composition independent** K values and a suspected 2-phase region, we can solve for the phase compositions and distribution using just material balances.

$$
\begin{aligned}
z_i F &= x_i L + y_i V \\
z_i &= x_i(1-\beta) + y_i \beta \\
\frac{z_i}{x_i} &= 1 - \beta + K_i \beta \\
x_i &= \frac{z_i}{1+\beta(K_i-1)} \\
y_i &= K_i x_i = \frac{K_i z_i}{1+\beta(K_i-1)}
\end{aligned}
$$

where β is the *vapour fraction*, expressed as

$$
\beta = \frac{L}{V}
$$

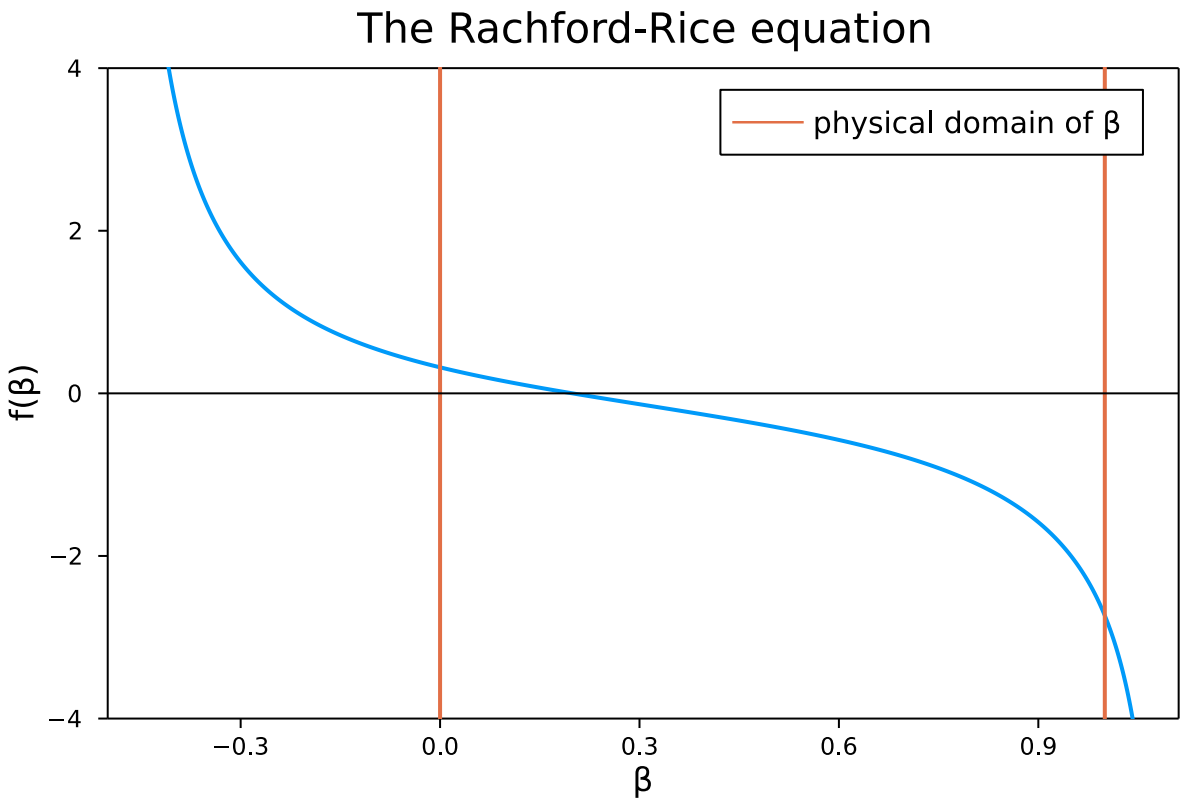
Since mole fractions sum to one,

$$
\sum x_i = \sum y_i = 1
$$

we can subtract each equation from each other to obtain our objective function, the Rachford-Rice equation.

$$
f(\beta) = \sum_i^{N_c} \frac{(K_i - 1) z_i}{1+\beta(K_i-1)} = 0
$$

This formulation carries a few advantages over the other possible formulations - primarily that it is a **monotonic** function, and has easily obtainable analytical derivatives. When K_i is known, it is then a univariate equation in β easily solvable in a variety of ways. The method used here is the **step limited Newton method** suggested by Michelsen.



From inspecting our equation, we can see that the equation will diverge to infinity when

$$\beta(K_i - 1) = -1$$

As we're summing across many K_i values, the interval where the Rachford-Rice function is well behaved is described by

$$\beta \in \left(\frac{1}{1 - K_{\text{max}}}, \frac{1}{1 - K_{\text{min}}} \right)$$

Another key feature to notice is that this bracket often falls outside of the physical domain of β

$$\beta_{\text{physical}} \in [0, 1]$$

We refer to the case where the solution to the Rachford-Rice equation falls outside of this physical domain as a **negative flash**. This implies that VLE does exist at the overall conditions but the material balance is not satisfied, so the mixture does not split into multiple phases.

Once we have the vapour fraction we can apply the material balances to calculate the compositions of our liquid and vapour phases.

Task: Solving the Rachford-Rice equation

Finish the functions below to solve for the phase distribution at the given K and z values.

Use the Rachford-Rice equation as defined above, and the first derivative

$$f(\beta) = \sum_{i}^{N_C} \frac{(K_i - 1)z_i}{1 + \beta(K_i - 1)} = 0$$

$$f'(\beta) = -\sum_{i}^{N_C} \frac{z_i(K_i - 1)^2}{(1 + \beta(K_i - 1))^2}$$

Also use the Newton method in 1D, remembering it is defined as

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)}$$

The step-limiting will be done by checking if

$$\beta_{\text{new}} \geq \beta_{\text{max}}$$

or

$$\beta_{\text{new}} \leq \beta_{\text{min}}$$

and if either of those cases is true, reduce the step size by half until β_{new} is within the domain.

$$d_{\text{new}} = \frac{d}{2}$$

where

$$d_0 = \frac{f(x)}{f'(x)}$$

Your function should accept a composition vector $\boldsymbol{z}$ and a K-factor vector $\boldsymbol{K}$, and return a scalar for the vapour fraction β.

`rachford_rice`

```
rachford_rice(z, K, β)
```

Evaluates the Rachford-Rice objective function and returns a tuple (f, f'), where f is the function value at the given point and f' is the derivative with respect to β.

```
"""
    rachford_rice(z, K, β)

Evaluates the Rachford-Rice objective function and returns a tuple (f, f′),
where f is the function value at the given point and f′ is the derivative with
respect to β.
"""
function rachford_rice(z, K, β)
    inner_vec = @. (K - 1) / (1 + β*(K - 1))
    f = sum(z.*inner_vec)
    f′ = -sum(z.*inner_vec.^2)
    return (f, f′)
end
```

Hint 1

[illegible]

solve_β

```
solve_β(z, K)
```

Solves the Rachford-Rice equation for the vapour fraction β using a step-limited Newton method.

```
"""
    solve_β(z, K)

Solves the Rachford-Rice equation for the vapour fraction β using a step-limited
Newton method.
"""
function solve_β(z, K)

    βmin = 1/(1-maximum(K))
    βmax = 1/(1-minimum(K))

    # Initial guess for β.
    β = (βmin + βmax)/2
    δβ = 1.0
    i = 0
    itersmax = 100
    # While the change is greater than our tolerance
    while i < itersmax && abs(δβ) > 1e-7
        i += 1
        f, f′ = rachford_rice(z, K, β)
        # Calculate the newton step
        d = f/f′

        # Keep β inside the limits of f(β)
        step_ok = false
        while !step_ok
            βnew = β - d
            if βnew > βmax || βnew < βmin # Reject newton step
                # println("reducing step size (i=$i)")
                d = 0.5*d
            else
                # println("continuing (i=$i, d=$d)")
                step_ok = true
                δβ = βnew - β
                β = βnew
            end
        end
    end
    if i == itersmax
        @warn "failed to converge in $i iterations\n δB = $δβ\n β = $β\n βmin =
        $βmin\n βmax = $βmax"
    end
    return β
end
```

Hint 2

Calculate your Newton step using

$$\delta = \frac{f}{f'}$$

You can write f' by typing \prime then pressing tab

Now lets calculate the value of β based off of your Rachford-Rice solver

```
0.788888888888889
```

```
begin
    z_RR = [0.8, 0.0, 0.2]
    K_RR = [3.0, 0.25, 0.10]
    β_RR = solve_β(z_RR, K_RR)
end
```

Got it!

Let's move on to the next section.

The Isothermal Flash

The procedure above set out how to solve for the phase distribution for the case where our K-factors are composition independent, but this approximation is only valid for very ideal species. To move to more general phase split calculations, we must account for this.

To do this we use the fact that K factors can be defined in two ways

$$K_i = \frac{y_i}{x_i}$$
$$K_i = \frac{\varphi_i^L(p, T, \mathbf{x})}{\varphi_i^V(p, T, \mathbf{y})}$$

allowing us to formulate the problem as

$$\mathbf{K} = f(\mathbf{K})$$

which we solve for the **fixed points**. We can use many ways to solve for the fixed points, but the most simple is **subsequent substitution**. This is when we iterate using

$$\mathbf{K}_{n+1} = f(\mathbf{K}_n)$$

until the value of $\mathbf{K}$ converges to a fixed point.

Using successive substitution, this algorithm is broken up into 4 stages

```
0. Specify the state
1. Calculate initial guesses

While not converged
2. Solve Rachford-Rice equation for x⃗, y⃗
3. Calculate new K⃗

When converged
4. Calculate final β, x⃗, y⃗
```

Where stages 2 and 3 are the va

0. Specifying our state

Before we begin any calculations, we need to specify a few things. We need to define the composition of our feed, our thermodynamic model, and the pressure and temperature we're conducting the flash at. Here, we're using **Peng Robinson** to model a 2-component mixture.

z specification

Component	Mole fraction
Methane	0.8
Hydrogen Sulfide	0.2

pT specification	
Pressure	60 bar
Temperature	255 K

1. Initial guesses - the Wilson equation

$$\ln K_i = \ln \frac{P_{c,i}}{P_i} + 5.373(1+\omega_i)\left(1 - \frac{T_{c,i}}{T}\right)$$

This is the same initialisation procedure we used for our **stability analysis** algorithm. Another possibility would be to use the result from stability analysis as an initial guess. That can be valuable as good initial guesses are particularly important when avoiding converging to the **trivial solution**.

Component	K-factor
methane	3.282
hydrogen sulfide	0.106

2. Rachford-Rice equation

For this, we will use the solver we just developed to calculate

$$\beta = f(\mathbf{K}, \mathbf{z})$$

Then use the material balances to calculate the phase compositions

$$x_i = \frac{z_i}{1 + \beta(K_i - 1)}$$
$$y_i = K_i x_i$$

3. Update K-factors

To calculate the next set of K-factors we use the expression

$$K_i = \frac{\varphi_i^L}{\varphi_i^V}$$

where φ can be obtained using Clapeyron with

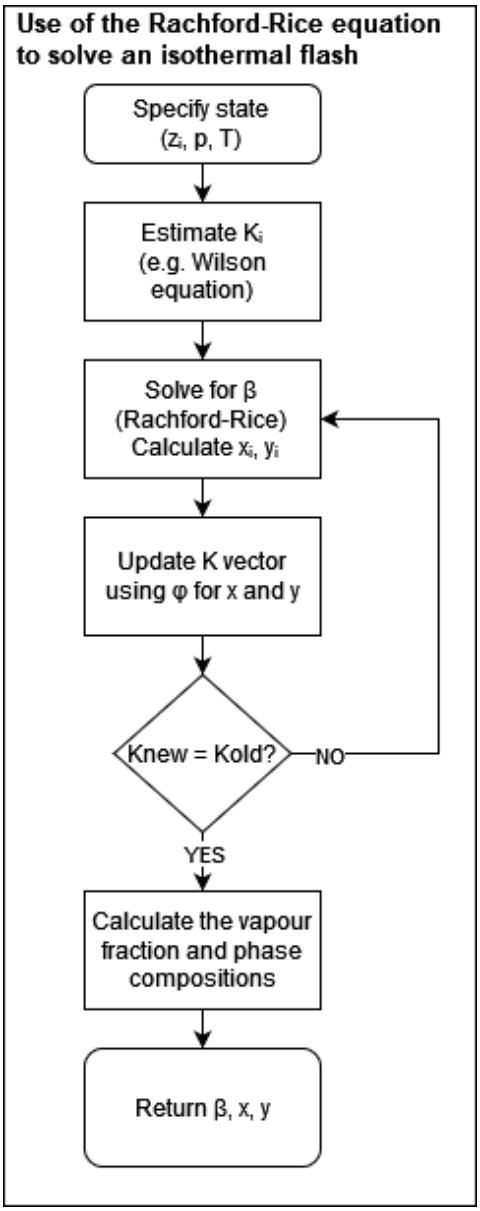
```
fugacity_coefficient(model, p, T, z; phase = :unknown)
```

4. Return values

We then apply the same procudure as in step 2 to calculate the final return values

$$(\beta,\ \mathbf{x},\ \mathbf{y})$$

Flowchart



Implementation

Below there is a working implementation of the flash described above. Try changing some parameters and see how the phase split changes!

Wilson_K_factor

```
Wilson_K_factor(pure_model, p, T)
```

returns a vector of K-factors predicted by the Wilson correlation for a pure component at a given pressure and temperature.

```
"""
    Wilson_K_factor(pure_model, p, T)

returns a vector of K-factors predicted by the Wilson correlation for a pure component at a given pressure and temperature.
"""
function Wilson_K_factor(pure_model, p, T)
    Tc, pc, vc = crit_pure(pure_model)
    ω = acentric_factor(pure_model)

    # Calculate K-factors
    K = exp(log(pc/p) + 5.373*(1+ω)*(1-Tc/T))
    return K
end
```

update_K_factors

```
update_K_factors(model, p, T, x, y)
```

For use in VLE flash calculations. Updates the K-factor vector using fugacity coefficients. Returns the new vector of K-factors.

```
"""
    update_K_factors(model, p, T, x, y)
For use in VLE flash calculations. Updates the K-factor vector using fugacity coefficients. Returns the new vector of K-factors.
"""
function update_K_factors(model, p, T, x, y)
    φˡ = fugacity_coefficient(model, p, T, x; phase=:liquid)
    φᵛ = fugacity_coefficient(model, p, T, y; phase=:vapour)
    K = φˡ./φᵛ
    return K
end
```

solve_flash

```
solve_flash(model, p, T, z, K0; maxiters=100, abstol=1e-9)
```

Solves the isothermal-isobaric two-phase flash problem using successive substitution

```
"""
    solve_flash(model, p, T, z, K0; maxiters=100, abstol=1e-9)
Solves the isothermal-isobaric two-phase flash problem using successive substitution
"""
function solve_flash(model, p, T, z, K0; maxiters=100, abstol=1e-10)
    iters = 0
    K_norm = 1.0
    K = K0
    while K_norm > abstol && iters < maxiters
        iters += 1

        β = solve_β(z, K)
        x = @. z/(1 + β*(K - 1))
        y = @. K*x

        Knew = update_K_factors(model, p, T, x, y)
        K_norm = norm(Knew .- K)
        K = Knew
    end

    # Make sure to issue a warning if the flash failed to converge!
    iters == maxiters && @warn "did not converge in $iters iterations"

    β = solve_β(z, K)
    x = @. z/(1 + β*(K - 1))
    y = @. K*x
    x = x./sum(x)
    y = y./sum(y)
    return (β, x, y)
end
```

$z_{methane} =$ 0.8

▸ [0.8, 0.2]

```
begin
    comps = ["methane", "hydrogen sulfide"] # Define the components
    model = PR(comps) # Create our fluid model

    pure_models = split_model.(model) # Create a vector of models for each of our
    pure substances

    # Describe our state
    p = 55e5 # Pa
    T = 255.0 # K
    z = [z_methane, 1-z_methane]
end
```

```
(0.963565, [0.198242, 0.801758], [0.822754, 0.177246])
```

```
begin
    K0 = Wilson_K_factor.(pure_models, p, T)
    β_flash, x_flash, y_flash = solve_flash(model, p, T, z, K0)
end
```

Component	Mole fraction	
	Liquid	Vapour
methane	0.1982	0.8228
hydrogen sulfide	0.8018	0.1772

$$\beta = 0.9636$$

And, as always, we can compare this result to Clapeyron

```
(2×2 Matrix{Float64}:, 2×2 Matrix{Float64}: , -1.23725)
  0.198242  0.801758    0.0072229  0.0292118
  0.822754  0.177246    0.792777   0.170788
```

```
x_Clapeyron, n_Clapeyron, G = tp_flash(model, p, T, z, RRTPFlash())
```

```
true
```

```
x_Clapeyron[1,:] ≈ x_flash
```

```
true
```

```
x_Clapeyron[2,:] ≈ y_flash
```

Got it!

Awesome!

Convergence of the two-phase flash

Subsequent substitution converges very quickly for situations where the state is far from the critical point and when the K-factors are weakly dependent on composition. When we fall outside of these cases, this method can take hundreds of iterations to converge.

To visualise this, we can keep track of our convergence measure (the Euclidian norm of change in $\mathbf{K}$) and plot this against the number of iterations.

Here we're looking at three different numerical methods

1. Successive substitution
2. Accelerated successive substitution **[1]**
3. Newton's method

In the figure below we show the convergence speed of three different iteration procedures for the flash calculation. We see the Newton algorithm converges very quickly, followed by the accelerated successive substitution, then finally the standard successive substitution procedure we wrote earlier.

However, before analysing their real-world performance, we should have an idea of how they may be expected to perform.

Successive substitution has **linear** convergence. This means that each successive error is approximately proportional,

$$||\boldsymbol{x}_{i+1} - \boldsymbol{x}^*|| \leq c||\boldsymbol{x}_i - \boldsymbol{x}^*||$$

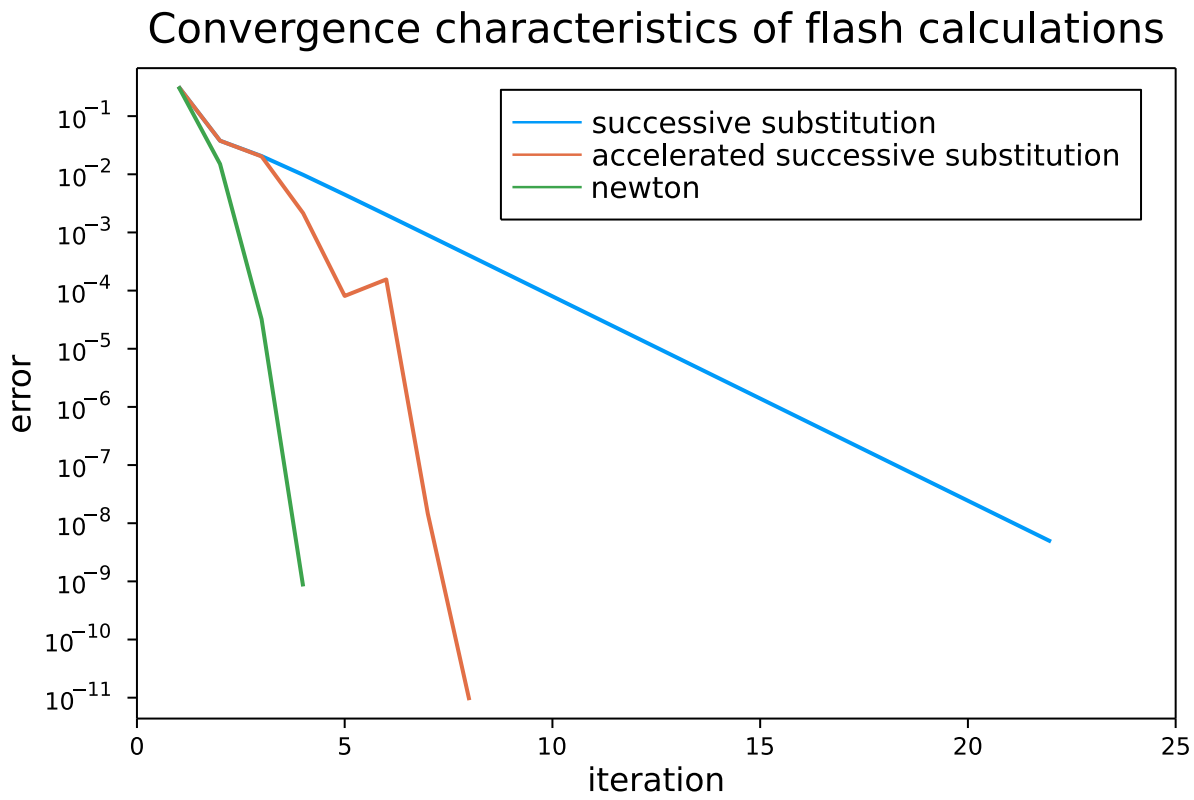
where $\boldsymbol{x}^*$ represents the final value, and $\boldsymbol{x}_i, \boldsymbol{x}_{i+1}$ represent iterations.

Accelerated successive substitution has **superlinear** convergence. This is when our convergence is faster than linear, but not yet quadratic. The acceleration procedure uses the last *m* iterations, in this case 5, to extrapolate each step and converge faster.

Finally, Newton's method has **quadratic** convergence. Here, the error within each successive iteration obeys

$$||\boldsymbol{x}_{i+1} - \boldsymbol{x}^*|| \leq c(||\boldsymbol{x}_i - \boldsymbol{x}^*||)^2 \, .$$

```
rootfinding_obj_func! (generic function with 1 method)
```



Benchmarks

Successive substitution

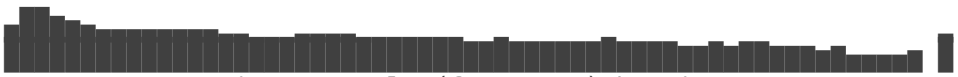
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  102.800 μs …   9.627 ms  ┊ GC (min … max):  0.00% … 96.48%
 Time  (median):     117.300 μs               ┊ GC (median):     0.00%
 Time  (mean ± σ):   145.978 μs ± 371.515 μs  ┊ GC (mean ± σ):  10.58% ±  4.11%
```

```
 103 μs          Histogram: log(frequency) by time        272 μs <

 Memory estimate: 138.69 KiB, allocs estimate: 1219.
```

Accelerated successive substitution [1]

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  59.400 μs …  10.772 ms  ┊ GC (min … max): 0.00% … 98.40%
 Time  (median):     64.200 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   80.650 μs ± 273.137 μs  ┊ GC (mean ± σ):  9.41% ±  2.77%
```



```
 59.4 μs         Histogram: log(frequency) by time       156 μs <

 Memory estimate: 61.75 KiB, allocs estimate: 518.
```

Newton

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  190.600 μs …  56.686 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     218.200 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   254.599 μs ± 657.839 μs  ┊ GC (mean ± σ):  3.82% ± 2.88%
```

```
  191 μs        Histogram: log(frequency) by time        483 μs <

 Memory estimate: 67.14 KiB, allocs estimate: 852.
```

From the error-iteration graph above we can see that our real-world convergence represents what we expected. However, the benchmarks then show that Newton method in this case performs worse than even successive substitution!

This occurs because of the increased complexity of Newton's method. Requiring a full Jacobian for each iteration, as well as the subsequent linear solve, there is considerably more computational cost for each iteration.

Attempting to get the "best of both worlds", Michelsen proposed a combination algorithm, leveraging the advantages of both accelerated successive substitution and the Newton method. As we can see accelerated successive substitution converges very quickly for most situations, we perform up to 15 iterations of this method, then switch to a faster converging, but slower to evalute, method like Newton.

Footnotes

[1]:

> Here we use Anderson acceleration to speed up the convergence rate of our fixpoint iteration. This is slightly different to the methods usually used in literature, e.g. by Michelsen. There you could expect to see some form of eigenvalue extrapolation, either dominant eigenvalue method (DEM) or general dominant eigenvalue method (GDEM). Another "accelerated successive substitution" method is described by *Risnes et al*, and is described here.