PTFlash: A vectorized and parallel deep learning framework for two-phase flash calculation
Jingang Qu, Thibault Faney, Jean-Charles de Hemptinne, Soleiman Yousef, Patrick Gallinari

To cite this version:

HAL Id: hal-03659647
https://hal.science/hal-03659647v3
Submitted on 10 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License
PTFlash: A vectorized and parallel deep learning framework for two-phase flash calculation*

Jingang Qu*a,b,*, Thibault Faneyb, Jean-Charles de Hemptinneb, Soleiman Yousefb, Patrick Gallinari*a,c

aSorbonne Université, CNRS, ISIR, F-75005 Paris, France
bIFPEN, France
cCriteo AI Lab, Paris, France

Abstract

Phase equilibrium calculations are an essential part of numerical simulations of multi-component multi-phase flow in porous media, accounting for the largest share of the computational time. In this work, we introduce a fast and parallel framework, PTFlash, that vectorizes algorithms required for two-phase flash calculation using PyTorch, and can facilitate a wide range of downstream applications. Vectorization promotes parallelism and consequently leads to attractive hardware-agnostic acceleration. In addition, to further accelerate PTFlash, we design two task-specific neural networks, one for predicting the stability of given mixtures and the other for providing estimates of the distribution coefficients, which are trained offline and help shorten computation time by sidestepping stability analysis and reducing the number of iterations to reach convergence.

The evaluation of PTFlash was conducted on three case studies involving hydrocarbons, CO₂ and N₂, for which the phase equilibrium was tested over a large range of temperature, pressure and composition conditions, using the Soave-Redlich-Kwong (SRK) equation of state. We compare PTFlash with an

---

*This is the manuscript accepted by Fuel journal. For the formal publication, please refer to https://doi.org/10.1016/j.fuel.2022.125603

**The share link https://authors.elsevier.com/a/1fj0m3iM4IIcE allows free access to the article for 50 days from 7 Sep 2022.

*Corresponding author

Email address: jingang.qu@sorbonne-universite.fr (Jingang Qu)
in-house thermodynamic library, *Carnot*, written in C++ and performing flash calculations one by one on CPU. Results show speed-ups of up to two order of magnitude on large scale calculations, while maintaining perfect precision with the reference solution provided by *Carnot*.

*Keywords:* Flash calculation, Two-phase equilibrium, Vectorization, Deep learning

1. Introduction

Numerical simulation of multi-component multi-phase flow in porous media is an essential tool for many subsurface applications, from reservoir simulation to long term CO$_2$ storage. A core element of the simulator for such applications is to determine the phase distribution of a given fluid mixture at equilibrium, also known as flash calculation. Starting with the seminal work of Michelsen [1, 2], researchers have developed robust and efficient algorithms for isothermal two-phase flash calculation. These algorithms have been implemented in the IFPEN thermodynamic C++ library *Carnot*.

Nonetheless, flash calculations still account for the majority of simulation time in a large range of subsurface applications [3, 4]. In most simulators, flash calculations are performed for each grid cell at each time step. Moreover, since modern simulators tend to require higher and higher grid resolutions up to billions of grid cells [5], the share of computing time due to flash calculations is expected to increase as well. In this context, speeding up flash calculations has drawn increasing research interest.

Some efforts have been made to accelerate flash calculations. [6–8] proposed a reduction method aiming to reduce the number of independent variables by leveraging the sparsity of the binary interaction parameter matrix, resulting in a limited speed-up [4]. [9] introduced the shadow region method using the results of previous time steps to initiate the current one, which assumes that the changes in pressure, temperature, and composition of a given block are small between two adjacent time steps in typical compositional reservoir simulation.
presented tie-line based methods, which approximate the results of flash calculations through linear interpolation between existing tie-lines and can be seen as a kind of look-up table. In [11–16], the authors focused on the use of machine learning, which provides a collection of techniques that can effectively discover patterns and regularities in data. They used support vector machine [17], relevance vector machine [18] and neural networks [19] to directly predict equilibrium phases and provide more accurate initial estimates for flash calculations. In [5, 20], researchers focused on developing faster parallel linear solvers, with [5] mentioning specifically that the vectorization of partial equation of state (EOS) related operations would lead to faster execution.

In this work, we introduce PTFlash, a framework for two-phase flash calculation based on the SRK equation of state [21]. PTFlash is built on the deep learning framework PyTorch [22] and consists in two main elements, namely the vectorization of algorithms and the use of neural networks. First, we perform a complete rewrite of two-phase flash calculation algorithms of Carnot using PyTorch. This enables the systematic vectorization of the complex iterative algorithms implemented in Carnot, allowing in turn to efficiently harness modern hardware with the help of, e.g., Advanced Vector Extensions AVX for Intel CPUs [23] and CUDA for Nvidia GPUs [24]. Note that vectorization of complex iterative algorithms with branching is not straightforward and needs specific care. Second, we replace repetitive and time consuming parts of the original algorithms with deep neural networks trained on the exact solution. More specifically, one neural network is used to predict the stability of given mixtures, and the other is used to provide initial estimates for the iterative algorithms. Once well trained, neural networks are seamlessly incorporated into PTFlash. These two elements allow PTFlash to provide substantial speed-ups compared to Carnot, especially so in the context of flow simulations where parallel executions of flash calculations for up to a billion grid cells are needed.

The rest of this article is organized as follows. In Section 2 we introduce the fundamentals of isothermal two-phase flash calculation and present three case studies. In Section 3 we explain how to efficiently vectorize flash calcu-
2. Isothermal two-phase flash calculation

In this section, we introduce the essential concepts of isothermal two-phase flash calculation. In the following, without loss of generality, we consider the equilibrium between the liquid and vapor phases.

2.1. Problem setting

We consider a mixture of \( N_c \) components. Given pressure \((P)\), temperature \((T)\) and feed composition \((z = (z_1, \ldots, z_{N_c}))\), the objective of flash calculation is to determine the system state at equilibrium: single phase or coexistence of two phases. In the latter case, we need to additionally compute the molar fraction of vapor phase \(\theta_V\), the composition of the liquid phase \(x\) and that of the vapor phase \(y\). These properties are constrained by the following mass balance equations:

\[
x_i(1 - \theta_V) + y_i \theta_V = z_i, \quad \text{for } i = 1, \ldots, N_c \tag{1a}
\]

\[
\sum_{i=1}^{N_c} x_i = \sum_{i=1}^{N_c} y_i = 1 \tag{1b}
\]

In addition, the following equilibrium condition should be satisfied:

\[
\frac{\varphi^L_i(P, T, x)}{\varphi^V_i(P, T, y)} = \frac{y_i}{x_i} \tag{2}
\]

where the superscripts \(L\) and \(V\) refer to the liquid and vapor phases, respectively, and \(\varphi_i\) is the fugacity coefficient of component \(i\), which is a known nonlinear function of \(P, T\) and the corresponding phase composition. This function depends on an equation of state that relates pressure, temperature and volume. In this work, we use the SRK equation of state \cite{21} and solve it using an iterative
approach \[25\] rather than the analytical solution of the cubic equation, e.g., Cardano’s formula, which may be subject to numerical errors in certain edge cases \[26\]. For more details, see Appendix A.

2.2. Numerical solver

Equations 1 and 2 form a non-linear system, which is generally solved in a two-stage procedure. First, we establish the stability of a given mixture via stability analysis (Section 2.2.1). If the mixture is stable, only one phase exists at equilibrium. Otherwise, two phases coexist. Second, we determine \( \theta, x \) and \( y \) at equilibrium through phase split calculations (Section 2.2.2).

2.2.1. Stability analysis

A mixture of composition \( z \) is stable at specified \( P \) and \( T \) if and only if its total Gibbs energy is at the global minimum, which can be verified through the reduced tangent plane distance \[1\]:

\[
\text{tpd}(w) = \sum_{i=1}^{N_c} w_i (\ln w_i + \ln \phi_i(w) - \ln z_i - \ln \phi_i(z))
\]

where \( w \) is a trial phase composition. If \( \text{tpd}(w) \) is non-negative for any \( w \), the mixture is stable. This involves a constrained minimization problem, which is generally reframed as an unconstrained one:

\[
\text{tm}(W) = \sum_{i=1}^{N_c} W_i (\ln W_i + \ln \phi_i(W) - \ln z_i - \ln \phi_i(z) - 1)
\]

where \( \text{tm} \) is the modified tangent plane distance and \( W \) is mole numbers. To locate the minima of \( \text{tm} \), we first use the successive substitution method accelerated by the Dominant Eigenvalue Method (DEM) \[27\], which iterates:

\[
\ln W_i^{(k+1)} = \ln z_i + \ln \phi_i(z) - \ln \phi_i(W^{(k)})
\]

It is customary to initiate the minimization with two sets of estimates, that is, vapor-like estimate \( W_i = K_i z_i \) and liquid-like estimate \( W_i = z_i/K_i \), where \( K_i \)

\[5\]
is the distribution coefficients, defined as \( y_i / x_i \) and initialized via the Wilson approximation \[21\], as follows:

\[
\ln K_i = \ln \left( \frac{P_{c,i}}{P} \right) + 5.373(1 + \omega_i) \left( 1 - \frac{T_{c,i}}{T} \right) \tag{6}
\]

where \( T_{c,i} \) and \( P_{c,i} \) refer to the critical temperature and pressure of component \( i \), respectively, and \( \omega_i \) is the acentric factor.

Once converging to a stationary point (i.e., \( \max(|\partial\tau_m/\partial W|) < 1.0 \times 10^{-6} \)) or a negative \( \tau_m \) is found, successive substitution stops. If this does not happen after a fixed number of iterations (9 in our work), especially in the vicinity of critical points, we resort to a second-order optimization technique, i.e., the trust-region method \[28\], to minimize \( \tau_m(W) \), which we describe in Appendix B.1. In addition, based on the results of stability analysis, we can re-estimate \( K_i \) more accurately as \( z_i / W_i^L \) if \( \tau_m < \tau_m^V \) or \( W_i^V / z_i \) otherwise, where the superscripts \( V \) and \( L \) denote the results obtained using the vapor-like and liquid-like estimates, respectively.

### 2.2.2. Phase split calculations

Substituting \( K_i = y_i / x_i \) into Equation \[1\] yields the following Rachford-Rice equation \[29\]:

\[
f_{RR}(\theta_V, K) = \sum_{i=1}^{N_c} \frac{(K_i - 1)z_i}{1 + (K_i - 1)\theta_V} = 0 \tag{7}
\]

Given \( K = (K_1, \ldots, K_{N_c}) \), we solve the above equation using the method proposed by \[30\] to get \( \theta_V \), which is detailed in Appendix C.1.

To obtain \( \theta_V \), \( x \) and \( y \) at equilibrium, phase split calculations start with the accelerated successive substitution method, as illustrated in Figure \[1\] and the corresponding convergence criterion is \( \max(|K_i^{(k+1)} / K_i^{(k)} - 1|) < 1.0 \times 10^{-8} \). If successive substitution fails to converge after a few iterations (9 in our work), we use the trust-region method to minimize the reduced Gibbs energy:

\[
G = \sum_{i=1}^{N_c} n_i^L (\ln x_i + \ln \varphi_i^L) + \sum_{i=1}^{N_c} n_i^V (\ln y_i + \ln \varphi_i^V) \tag{8}
\]
\[ n_i^L = x_i (1 - \theta_V) \] and \[ n_i^V = y_i \theta_V \] are the mole numbers of liquid and vapor phases, respectively. The convergence criterion is \[ \max(|\partial G / \partial n_i^V|) < 1.0 \times 10^{-8} \]. For more details, see Appendix B.2.

2.2.3. Strategy for two-phase flash calculation

We basically adopt the rules of thumb proposed by Michelsen in the book \[31\] to implement two-phase flash calculation, as shown in Figure 2. In the flowchart, we first initialize the distribution coefficients \( K_i \) using the Wilson approximation. Subsequently, in order to avoid computationally expensive stability analysis, we carry out the successive substitution of phase split calculations 3 times, which will end up with 3 possible cases: (1) \( \theta_V \) is out of bounds (0, 1) during iterations. (2) None of \( \Delta G \), \( \text{tpd}(x) \) and \( \text{tpd}(y) \) are negative, where \( \text{tpd}(x) \) and \( \text{tpd}(y) \) are reduced tangent plane distances using current vapor and liquid phases as trial phases, and \( \Delta G = \theta_V \times \text{tpd}(x) + (1 - \theta_V) \times \text{tpd}(y) \). (3) Any of \( \Delta G \), \( \text{tpd}(x) \) and \( \text{tpd}(y) \) is negative.

For the first two cases, we cannot be sure of the stability of the given mixture, thus continuing with stability analysis. For the third case, we can conclude that the given mixture is unstable, thereby sidestepping stability analysis. Finally, if two phases coexist, we perform phase split calculations to get \( \theta_V \), \( x \) and \( y \) at equilibrium.
2.3. Case studies

Here, we introduce three case studies involving hydrocarbons, CO\(_2\) and N\(_2\), whose properties are shown in Table 1. In this work, we only consider the binary interaction parameter (BIP) between CH\(_4\) and CO\(_2\), which is 0.0882. The BIPs between the others are 0. The first case study focuses on a system of two components (CH\(_4\) and C\(_6\)H\(_{14}\)), and the second one involves four components (CH\(_4\), C\(_2\)H\(_6\), C\(_3\)H\(_8\) and C\(_4\)H\(_{10}\)). For these two case studies, the ranges of pressure and temperature are 0.1MPa - 10MPa and 200K - 500K, respectively, and we consider the entire compositional space, i.e., 0 < \(z_i\) < 1 for \(i = 1, \ldots, N_c\). The third case study includes all 9 components in Table 1. The bounds of pressure and temperature are 5MPa - 25MPa and 200K - 600K, respectively. In addition, from a practical perspective, given that some mixtures do not exist in nature, rather than considering the entire compositional space, we specify four different compositional ranges, as shown in Table 2, each of which represents one.
Table 1: Component properties

<table>
<thead>
<tr>
<th>Component</th>
<th>$P_c$ (MPa)</th>
<th>$T_c$ (K)</th>
<th>$\omega$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CH_4$</td>
<td>4.6</td>
<td>190.55</td>
<td>0.0111</td>
</tr>
<tr>
<td>$C_2H_6$</td>
<td>4.875</td>
<td>305.43</td>
<td>0.097</td>
</tr>
<tr>
<td>$C_3H_8$</td>
<td>4.268</td>
<td>369.82</td>
<td>0.1536</td>
</tr>
<tr>
<td>$n$-$C_4H_{10}$</td>
<td>3.796</td>
<td>425.16</td>
<td>0.2008</td>
</tr>
<tr>
<td>$n$-$C_5H_{12}$</td>
<td>3.3332</td>
<td>467.15</td>
<td>0.2635</td>
</tr>
<tr>
<td>$C_6H_{14}$</td>
<td>2.9688</td>
<td>507.4</td>
<td>0.296</td>
</tr>
<tr>
<td>$C_7H^{+}_{16}$</td>
<td>2.622</td>
<td>604.5</td>
<td>0.3565</td>
</tr>
<tr>
<td>$CO_2$</td>
<td>7.382</td>
<td>304.19</td>
<td>0.225</td>
</tr>
<tr>
<td>$N_2$</td>
<td>3.3944</td>
<td>126.25</td>
<td>0.039</td>
</tr>
</tbody>
</table>

of the common reservoir fluid types, namely wet gas, gas condensate, volatile oil, and black oil. Figure 3(a) shows phase diagrams of four typical reservoir fluids at fixed compositions, as defined in Appendix D and we can see that the more heavy hydrocarbons there are, the lower the pressure range of the phase envelope and the less volatile the fluid is.

Table 2: Four fluid types characterized by different compositional ranges

<table>
<thead>
<tr>
<th>Component</th>
<th>Wet gas</th>
<th>Gas condensate</th>
<th>Volatile oil</th>
<th>Black oil</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CH_4$</td>
<td>80% - 100%</td>
<td>60% - 80%</td>
<td>50% - 70%</td>
<td>20% - 40%</td>
</tr>
<tr>
<td>$C_2H_6$</td>
<td>2% - 7%</td>
<td>5% - 10%</td>
<td>6% - 10%</td>
<td>3% - 6%</td>
</tr>
<tr>
<td>$C_3H_8$</td>
<td>$\leq$ 3%</td>
<td>$\leq$ 4%</td>
<td>$\leq$ 4.5%</td>
<td>$\leq$ 1.5%</td>
</tr>
<tr>
<td>$n$-$C_4H_{10}$</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 3%</td>
<td>$\leq$ 3%</td>
<td>$\leq$ 1.5%</td>
</tr>
<tr>
<td>$n$-$C_5H_{12}$</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 1%</td>
</tr>
<tr>
<td>$C_6H_{14}$</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 2%</td>
</tr>
<tr>
<td>$C_7H^{+}_{16}$</td>
<td>$\leq$ 1%</td>
<td>5% - 10%</td>
<td>10% - 30%</td>
<td>45% - 65%</td>
</tr>
<tr>
<td>$CO_2$</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 3.5%</td>
<td>$\leq$ 2%</td>
<td>$\leq$ 0.1%</td>
</tr>
<tr>
<td>$N_2$</td>
<td>$\leq$ 0.5%</td>
<td>$\leq$ 0.5%</td>
<td>$\leq$ 0.5%</td>
<td>$\leq$ 0.5%</td>
</tr>
</tbody>
</table>
2.4. Data generation

To efficiently sample input data including $P$, $T$ and $z$, we first use Latin Hypercube Sampling (LHS) technique to take space-filling samples \cite{22}. Subsequently, for $P$ and $T$, we linearly transform the uniform distribution $U(0,1)$ to the expected ranges. For $z$ subject to $\sum z_i = 1$, we transform a set of $U(0,1)$ into the Dirichlet distribution $Dir(\alpha)$ whose support is a simplex, as follows:

\begin{align}
  x_i & \sim i.i.d. \ U(0,1) \text{ using LHS} \\
  y_i &= \Gamma(\alpha_i, 1).ppf(x_i) \\
  z_i &= \frac{y_i}{\sum_{i=1}^{N_c} y_i}
\end{align}

where $\alpha = (\alpha_1, \ldots, \alpha_{N_c})$ is the concentration parameters of the Dirichlet distribution and controls its mode, $\Gamma(\alpha_i, 1)$ is the Gamma distribution, $ppf$ represents the percent-point function, also known as the quantile function, and $z = (z_1, \ldots, z_{N_c}) \sim Dir(\alpha)$.

For the first two case studies, the concentration parameters are $\alpha = 1$, i.e., all-ones vector. For the third case study, we adjust $\alpha$ for different fluid types to make the probability of each compositional range as large as possible, as shown.
in Table 3. Figure 3(b) presents the marginal distribution of $z_i$ for black oil.

In summary, we sample $z$ using Equation 9 with different $\alpha$ specified in Table 3 and then we single out the acceptable samples located in the compositional ranges defined in Table 2. In the following, unless otherwise specified, four fluid types are always equally represented.

Table 3: Concentration parameters $\alpha$ for different fluid types in Table 2

<table>
<thead>
<tr>
<th>Fluid Type</th>
<th>$\alpha_1$ for $CH_4$</th>
<th>$\alpha_2$ for $C_2H_6$</th>
<th>$\alpha_7$ for $C_7H_{16}$</th>
<th>$\alpha_i$ for others</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wet gas</td>
<td>100</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Gas condensate</td>
<td>40</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Volatile oil</td>
<td>55</td>
<td>8</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>Black oil</td>
<td>25</td>
<td>4</td>
<td>40</td>
<td>1</td>
</tr>
</tbody>
</table>

Eventually, the samples of $P$, $T$ and $z$ are concatenated together to form the complete input data.

3. Vectorization of two-phase flash calculation

We vectorize the two-phase flash so that it takes as inputs $P = (P_1, \cdots, P_n)$, $T = (T_1, \cdots, T_n)$ and $z = (z_1, \cdots, z_n)$, where $P$ and $T$ are vectors, $z$ is a matrix, and $n$ denotes the number of samples processed concurrently and is often referred to as the batch dimension.

In recent years, Automatic Vectorization (AV) has emerged and developed\(^1\), e.g., JAX\(^3\), which can automatically vectorize a function through the batching transformation that adds a batch dimension to its input. In this way, the vectorized function can process a batch of inputs simultaneously rather than processing them one by one in a loop. However, AV comes at the expense of performance to some extent and is slower than well-designed manual vectorization, which vectorizes a function by carefully revamping its internal operations.

\(^1\)At the time of writing, PyTorch team released a fledgling library, `functorch`, which takes inspiration from JAX and supports Automatic Vectorization.
to accommodate to a batch of inputs. For example, matrix-vector products for a batch of vectors can be directly replaced with a matrix-matrix product. In addition, flash calculation has an iterative nature and complicated control flow, which is likely to result in the failure of AV. Consequently, for finer-grained control, more flexibility, and better performance, we manually vectorize all algorithms involved in flash calculation, including the solution of the SRK equation of state and the Rachford-Rice equation, stability analysis and phase split calculations.

To achieve efficient vectorization, one difficulty is asynchronous convergence, that is, for each algorithm, the number of iterations required to reach convergence generally varies for different samples, which hinders vectorization and parallelism. To alleviate this problem, we design a general-purpose paradigm, synchronizer, to save converged results in time at the end of each iteration and then remove the corresponding samples in order not to waste computational resources on them in the following iterations, which is achieved by leveraging a one-dimensional Boolean mask encapsulating convergence information to efficiently access data in vectors and matrices, as follows:

\[
X^{(k+1)} \leftarrow f(X^{(k)}) \quad (10a)
\]
\[
\text{Save } X^{(k+1)}[\text{mask}] \text{ to } \tilde{X} \quad (10b)
\]
\[
X^{(k+1)} \leftarrow X^{(k+1)}[\sim \text{mask}] \quad (10c)
\]
\[
k \leftarrow k + 1 \quad (10d)
\]

where \(k\) is the number of iterations, \(f(X)\) is a vectorized iterated function taking as input \(X \in \mathbb{R}^{n \times m}\) (\(n\) is the batch dimension, i.e., number of samples, and \(m\) is the dimension of \(X\)), \(\tilde{X}\) is a placeholder matrix used to save converged results, mask is a Boolean vector where True means convergence, and \(\sim\) denotes the logical NOT operator. The number of unconverged samples gradually decreases as a result of incremental convergence. For the full version of synchronizer, refer to Appendix E.1 We can use synchronizer to wrap and vectorize
any iterative algorithm. For instance, we illustrate how to perform vectorized
stability analysis in Appendix E.2.

The efficiency of synchronizer may be questioned because previously con-
verged samples are still waiting for un converged ones before moving to the next
step. This is true, but the situation is not as pessimistic since we try to shorten
the waiting time as much as possible. For example, if successive substitution
fails to converge quickly, we immediately use the trust-region method. In any
case, the delay caused by waiting is insignificant compared to the acceleration
due to vectorization. Furthermore, we leverage neural networks to provide more
accurate initial estimate $X^{(0)}$ so that all samples converge as simultaneously as
possible, thereby reducing asynchrony, which we will present in Section 4.

Once all algorithms are well vectorized, another problem is how to globally
coordinate different subroutines. To this end, we add barrier synchronization
to the entry points of stability analysis and phase split calculations in Figure
2, which can avoid any subroutine connected to it proceeding further until all
others terminate and arrive at this barrier.

We also optimized the code using TorchScript [22], allowing for more efficient
execution through algebraic peephole optimizations and fusion of some opera-
tions, and more practical asynchronous parallelism without the Python global
interpreter lock [34], whereby vapor-like and liquid-like estimates are dealt with
in parallel in stability analysis.

4. Acceleration of flash calculation using neural networks

To further accelerate flash calculation, we create and train two task-specific
neural networks, classifier and initializer. The classifier is used to predict the
probability $p$ that a given mixture is stable, i.e., $p = \text{classifier}(P, T, z)$, which
involves a binary classification problem. It can predict the stability of most
samples, thereby bypassing stability analysis and saving time. The initializer
is able to initialize $K_i$ more accurately than the Wilson approximation, i.e.,
$\ln K_i = \text{initializer}(P, T, z)$, which relates to a regression problem. It can reduce
the number of iterations required to reach convergence and alleviate the asyn-
chronous convergence we introduced before. Note that the hyper-parameters
of neural networks presented below, e.g., the number of units and layers, are
dedicated to the case study containing 9 components. Nonetheless, the basic
architecture of neural networks and the training methods can be generalized to
any case.

4.1. Classifier

4.1.1. Architecture

As shown in Figure 4(a), the classifier has 3 hidden layers with 32 neurons
and using the SiLU activation function \( [35–37] \). The output layer has only one
neuron and uses the sigmoid activation function compressing a real number to
the range \((0, 1)\). The input \( x \) consists of \( P, T \) and \( z \), and the output is the
probability \( p \) that a given mixture is stable. The scaling layer standardizes the
inputs as \( (x - u) / s \), where \( u \) and \( s \) are the mean and standard deviation of \( x \)
over the training set. To train the classifier, we use the binary cross-entropy
(bce), which is the de-facto loss function for binary classification problems and
defined as:

\[
bce(y, p) = y \ln p + (1 - y) \ln(1 - p)
\]  \( (11) \)

where \( y \) is either 0 for unstable mixtures or 1 for stable ones.

The architecture of the classifier is obtained by tuning hyper-parameters us-
ing Tree-Structured Parzen Estimator optimization algorithm \( [38] \) with Asyn-
chronous Successive Halving algorithm \( [39] \) as an auxiliary tool to early stop
less promising trials. We create a dataset containing 100,000 samples (80% for
training and 20% for validation), and then tune the hyper-parameters of the
classifier with 150 trials to minimize the loss on the validation set (we use Adam
\( [40] \) as optimizer and the batch size is 512), as shown in Figure 4(b). We can
see that SiLU largely outperforms other activation functions.
(a) The architecture of classifier for the 9-component mixture

Figure 4: Figure (a) shows the architecture of the classifier. Figure (b) is a parallel coordinates plot used to visualize the results of tuning hyper-parameters of the classifier, where lr stands for learning rate. The colors of lines are mapped to the value of the loss.

4.1.2. Training

We first generate one million samples in the way described in Section 2.3 and then feed them to PTFlash to determine stability (no need for phase split calculations), which takes about 2 seconds. Subsequently, these samples are divided into the training (70%), validation (15%) and test (15%) sets. To train the classifier, we set the batch size to 512 and use Adam with Triangular Cyclic Learning Rate (CLR) [41, 42], which periodically increases and decreases the learning rate during training, as shown in Figure 5(a). [43] claimed that CLR helps to escape local minima and has the opportunity to achieve superb performance using fewer epochs and less time. We found that Adam with and without CLR achieved similar performance, but the former converged five times faster than the latter. Early stopping is also used to avoid overfitting [44]. The total training time is about 5 minutes using Nvidia RTX 3080. The final performance of the classifier on the test set is bce = 0.002 and accuracy = 99.93%. For a more intuitive understanding of performance, Figure 5(b) shows the contours of probabilities predicted by the classifier, where the blue contour of $p = 0.5$. 
basically coincides with the phase envelope. In the zoomed inset, the additional
green and yellow contours correspond to $p=0.02$ and 0.98, respectively.

![Graph of cyclic learning rate](image1)

(a) Cyclic learning rate of the classifier

![Graph illustrating contours of probabilities](image2)

(b) Prediction of the classifier for volatile oil

Figure 5: Figure (a) shows how the learning rate varies cyclically. Figure (b) illustrates the contours of probabilities predicted by the classifier for volatile oil at fixed composition. The red and gray correspond to the two-phase and monophasic regions, respectively.

4.2. Initializer

4.2.1. Architecture

The input of the initializer includes $P$, $T$, and $z$, and its output is $\ln K_i$. The initializer has 1 hidden layer and 3 residual blocks, as shown in Figure 6. Each residual block has 2 hidden layers and a shortcut connection adding the input of the first hidden layer to the output of the second [45]. All hidden layers have 64 neurons and use the SiLU activation function. The output layer has $N_c$ neurons without activation function. The wide shortcut, proposed in [46], enables neural networks to directly learn simple rules via it besides deep patterns through hidden layers, which is motivated by the fact that the inputs, such as $P$ and $T$, are directly involved in the calculation of $K_i$. The concat layer concatenates the input layer and the outputs of the last residual block (the concatenation means putting two matrices $A \in \mathbb{R}^{d_1 \times d_2}$ and $B \in \mathbb{R}^{d_1 \times d_3}$ together to form a new one $C \in \mathbb{R}^{d_1 \times (d_2+d_3)}$). In addition, the targets of the initializer are $\ln K_i$ instead of $K_i$, since $K_i$ varies in different orders of magnitude, which
hampers the training of the initializer, whereas \( \ln K_i \) does not.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{initializer_architecture.png}
\caption{The architecture of initializer for the 9-component mixture}
\end{figure}

We found that the convergence of phase split calculations is robust if \( K_i \) predicted by the initializer can lead to more accurate values of the vapor fraction \( \theta_V \), especially around critical points where calculations are quite sensitive to initial \( K_i \) and prone to degenerate into trivial solutions. As a consequence, the loss function used to train the initializer consists of two parts, one is the mean absolute error (mae) in terms of \( K_i \) and the other is mae in terms of \( \theta_V \), as follows:

\begin{align}
\text{mae}(\ln K, \ln \hat{K}) &= \frac{1}{N} \sum_{i=1}^{N} |\ln K_i - \ln \hat{K}_i| \quad (12a) \\
\text{mae}(\theta_V, \hat{\theta}_V) &= |\theta_V - \hat{\theta}_V| \quad (12b)
\end{align}

where \( \ln K \) is the ground truth, \( \ln \hat{K} \) is the prediction of the initializer, \( \theta_V \) is the vapor fraction at equilibrium, and \( \hat{\theta}_V \) is obtained by solving the Rachford-Rice equation given \( z \) and the prediction \( \hat{K} \).

4.2.2. Training

We generate one million samples in the two-phase region (\( K_i \) is not available at the monophasic region), which are divided into the training (70%), validation
and test (15%) sets. The training of the initializer is carried out in two stages. First, we train it to minimize mae(ln $K_i$, ln $\hat{K}_i$), using Adam with CLR and setting the batch size to 512. Second, after the above training, we further train it to minimize mae(ln $K_i$, ln $\hat{K}_i$) + mae($\theta_{V_i}$, $\hat{\theta}_{V_i}$), using Adam with a small learning rate $1.0 \times 10^{-5}$. Here, $\partial \hat{\theta}_{V_i} / \partial \hat{K}_i$ is required during backpropagation and can be simply computed via PyTorch’s automatic differentiation, which, however, differentiates through all the unrolled iterations, since we solve the Rachford-Rice equation in an iterative manner we described in Appendix C.1. Instead, we can make use of the implicit function theorem [47] to directly obtain $\partial \hat{\theta}_{V_i} / \partial \hat{K}_i$ by using the derivative information at the solution point of the Rachford-Rice equation, as follows:

$$\partial \hat{\theta}_{V_i} / \partial \hat{K}_i = -[\partial_{\theta_{V_i}} f_{RR}(\hat{\theta}_{V_i}, \hat{K}_i)]^{-1} \partial_K f_{RR}(\hat{\theta}_{V_i}, \hat{K}_i)$$  \hspace{1cm} (13)$$

This way is obviously more efficient and avoids differentiation through iterations.

Finally, the performance of the initializer on the test set is mae = 9.66e-4 in terms of ln $K_i$ and mae = 1.86e-3 in terms of $K_i$.

4.3. Strategy for accelerating flash calculation using neural networks

As shown in Figure 7, given $P$, $T$ and $z$, we first use the classifier to predict $p$. Next, based on two predefined thresholds, $p_l$ and $p_r$, satisfying $p_l \leq p_r$, the given mixture is thought of as unstable if $p \leq p_l$ or stable if $p \geq p_r$. If $p_l < p < p_r$, we will use stability analysis to avoid unexpected errors. Here, we can adjust $p_l$ and $p_r$ to trade reliability for speed. In general, less errors occur with smaller $p_l$ and greater $p_r$, but probably taking more time on stability analysis, and vice versa. A special case is $p_l = p_r = p_c$, where $p_c$ could be a well-calibrated probability or simply set to 0.5, which means that we completely trust the classifier (i.e., stable if $p \geq p_c$ or unstable otherwise), and no extra stability analysis is required. For the initializer, it serves both stability analysis when $p_l < p < p_r$ and phase split calculations.
Neural networks can also be used individually. If only the classifier is available, one may initialize $K_i$ via the Wilson approximation rather than the initializer in Figure 7. If only the initializer is available, one may use it to initialize $K_i$ in Figure 2.

![Flowchart](image)

Figure 7: Acceleration of flash calculation using neural networks

5. Results

In this section, we will compare our proposed framework for vectorized flash calculation, PTFlash, with Carnot, an in-house thermodynamic library developed by IFP Energies Nouvelles and based on C++. Carnot performs two-phase flash calculation in the manner shown in Figure 2 but can only handle samples one at a time on CPU. Regarding the hardware, CPU is Intel 11700F and GPU is NVIDIA RTX 3080 featuring 8704 CUDA cores and 10G memory. Note that since using multiple cores renders the frequency quite unstable due to heat accumulation, we only use one core of CPU so that the frequency can be stabilized at 4.5GHz, which allows for a consistent criterion for measuring the execution time.
PTFlash and Carnot gave identical results (coincidence to 9 decimal places under double-precision floating-point format) because they use exactly the same convergence criteria for all iterative algorithms. In the following, we will focus on comparing their speeds.

5.1. Vectorized flash calculation

We compare the execution time of different methods for flash calculation with respect to the workload quantified by the number of samples $n$, as shown in Figures 8. Due to GPU memory limitations, the maximum number of samples allowed is 10, 5, and 1 million for the three case studies, respectively. We can see that all three figures exhibit the same behavior. When the workload is relatively low, e.g., $n < 1000$, Carnot wins by large margins, and CPU is also preferable based on the fact that PTFlash runs much faster on CPU than on GPU. On the one hand, PyTorch has some fixed overhead in the setup of the working environment, e.g., the creation of tensors. On the other hand, when GPU is used, there are some additional costs of CPU-GPU communication and synchronization. When $n$ is small, these overheads dominate. As proof, we can see that the time of PTFlash on GPU hardly changes as $n$ varies from 100 to $10^4$. In contrast, the time of Carnot is almost proportional to $n$.

As the workload increases, the strength of PTFlash on GPU emerges and becomes increasingly prominent. For the three case studies, PTFlash on GPU is 163.4 (2 components), 106.3 (4 components) and 50.5 (9 components) times faster than Carnot at the maximum number of samples. This suggests that PTFlash on GPU is more suitable for large scale computation. Interestingly, we can observe that PTFlash on CPU also outperforms Carnot when the workload is relatively heavy, e.g., $n > 10^3$. In fact, thanks to Advanced Vector Extensions, vectorization enables fuller utilization of CPU’s computational power.

We notice that there is a lack of a comprehensive and unified benchmark for the runtime of flash calculation in the literature. Here, we give an article with a case study similar to ours for readers’ reference [48], which claimed that the total computation time of flash calculations is 10 seconds for one million samples.
of a 9-component mixture. However, it is worth pointing out that the sampling
method, convergence criteria and algorithm implementation in this reference
article are different from ours. In our work, these aspects are consistent for
both Carnot and PTFlash to ensure a fair comparison.

Next, we focus on the mixture of 9 components and analyze the performance
of PTFlash for this case study. Table 4 is a performance profiler of PTFlash
on GPU at $n = 10^6$, which records the running time of each subroutine of
flash calculations. As a complement, Figures 9 dissect phase split calculations
by tracking the total elapsed time and the convergence percentage up to each
iteration, as well as the mean of critical distances $d_c$ of converged samples at
each iteration, where $d_c$ is defined as:

$$d_c = \sqrt{\sum_{i=1}^{N_c} \ln K_i^2}$$  \hspace{1cm} (14)$$

The closer to critical points, the smaller $d_c$. In other words, $d_c$ indicates the
closeness to critical points.

The observations of Figures 9 are summarized as follows: (1) In Figure 9(a) the slope of time with respect to the number of iterations is decreasing because
the workload is reduced due to incremental convergence. (2) In Figure 9(b) for the samples that do not converge after successive substitution, the majority of
them (92.67%) converge after 3 iterations of the trust-region method. (3) In Figure 9(c) $d_c$ decreases during iterations, which means that samples close to
critical points converge last and also confirms that convergence is slow around
critical points.

Table 4: Performance profiler of PTFlash on GPU (Figure 2) for the mixture of 9 components at $n = 10^6$ in Figure 8(c)

<table>
<thead>
<tr>
<th># of samples</th>
<th>ss of phase split</th>
<th>Stability analysis</th>
<th>Phase split calculations</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ss</td>
<td>tr</td>
<td>ss</td>
</tr>
<tr>
<td>$10^6$</td>
<td>625645</td>
<td>130715</td>
<td>625645</td>
</tr>
<tr>
<td>Convergence</td>
<td>37.44% 1</td>
<td>79.11%</td>
<td>100%</td>
</tr>
<tr>
<td>Max number of iterations</td>
<td>3</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>Total time</td>
<td>0.4565s</td>
<td>0.4136s</td>
<td>0.3417s</td>
</tr>
</tbody>
</table>

1 \(37.44\%\) is the percentage of samples for which any of \(\Delta G, tpd_x\) and \(tpd_y\) is negative after 3 attempts of successive substitution, as described in Section 2.2.3.
2 The total time of stability analysis is less than the sum of the times of all subroutines because vapor-like and liquid-like estimates are handled concurrently.

The above analysis gives us a general understanding of PTFlash, but in fact
it is not easy to analyze PTFlash comprehensively because each subroutine also contains iterative algorithms, such as solving the SRK equation of state and the Rachford-Rice equation. Nevertheless, given the information already obtained, we know that we need to shorten the time of stability analysis and reduce the number of iterations in order to accelerate PTFlash, which is exactly the role of the classifier and initializer.

5.2. Deep-learning-powered vectorized flash calculation

We trained neural networks following Section 4 for the mixture of 9 components. Here, we will explore the effect of neural networks. First of all, we set
\(p_l = 0.02\) and \(p_r = 0.98\) as the thresholds of stability and instability, which are carefully chosen so that no misclassification occurs. In Figure 8(c), we can see that \(NN-PTFlash\) outpaces \(PTFlash\) on both CPU (2.7x speed-up) and GPU (2.2x speed-up). In addition, \(NN-PTFlash\) on GPU runs almost 110.7 times faster than \(Carnot\) at \(n = 10^6\).

Table 5 is the performance profiler of \(NN-PTFlash\) on GPU. We can see that the classifier is able to precisely determine the stability of the vast majority of samples (99.42%), which significantly relieves the burden of stability analysis and saves time. In addition, compared to phase split calculations of \(PTFlash\), the convergence percentage of successive substitution increases from 45.88% to 67.40%, and the overall time is also greatly reduced, which is attributed to better initial \(K_i\) provided by the initializer.

<table>
<thead>
<tr>
<th></th>
<th>Stability analysis</th>
<th>Phase split calculations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vapor-like estimate</td>
<td>liquid-like estimate</td>
</tr>
<tr>
<td>classifier</td>
<td>ss</td>
<td>tr</td>
</tr>
<tr>
<td># of samples</td>
<td>(10^6)</td>
<td>5818</td>
</tr>
<tr>
<td>Convergence</td>
<td>99.42% (^1)</td>
<td>81.56%</td>
</tr>
<tr>
<td>Max number of iterations</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>Total time</td>
<td>0.0005s</td>
<td>0.1365s</td>
</tr>
</tbody>
</table>

ss: successive substitution  \(1\) 99.42% includes 58.38% predicted as stable (i.e., \(p > p_r\)) and 41.04% predicted as unstable (i.e., \(p < p_l\)).

We also performed ablation studies to compare the contributions of the classifier and initializer by using them individually. For instance, when handling 1 million samples for the case study containing 9 components, \(NN-PTFlash\) with only the classifier on GPU takes 1.88s. However, the attempt to use the ini-
tializer alone fails because we found its outputs may reach unreasonably large
values (e.g., $1.0 \times 10^{15}$) for stable mixtures far away from the boundary between the
single-phase and two-phase regions, which leads to numerical overflow. From
machine learning terminology, this is the out-of-distribution generalization prob-
lem, since the initializer is trained on the two-phase region and may suffer from
large predictive errors when used within the single-phase region. Nonetheless,
there is no problem when the initializer works in tandem with the classifier be-
cause remaining samples located in the single-phase region are fairly close to the
boundary after filtering through the classifier, as shown in Figure 5(b). In any
case, based on the fact that $NN$-$PTFlash$ using only the classifier always lags
behind that using both, we can conclude that both the classifier and initializer
play an important role in speeding up flash calculations.

5.3. Discussion

The results show that the systematic and exhaustive vectorization of two-
phase flash calculation does result in attractive speed-ups when large scale com-
putation is involved, e.g., the number of samples to process is on the order of
millions. Importantly, this speed-up does not come at the cost of accuracy and
stability like [11, 12, 14, 15] which are subject to the unreliability of machine
learning models. In addition, we can see that neural networks, such as the
classifier and initializer, really make a big difference.

Due to GPU memory limitations, the number of samples $n$ is limited in
Figures 8. Nonetheless, we can see that the slopes of time with respect to $n$ differ
significantly between different methods. The time of $Carnot$ is proportional to
$n$, in contrast, the time of $PTFlash$ on GPU is increasing slowly. Therefore,
it is reasonable to believe that the speed advantage of $PTFlash$ on GPU will
become increasingly prominent if $n$ continues to grow.

Using PyTorch has several benefits in addition to its simplicity and flexibil-
ity. First, we can seamlessly incorporate neural networks into $PTFlash$. Second,
any subroutine of $PTFlash$ is fully differentiable through automatic differenti-
ation, and we can also leverage the implicit function theorem for efficient dif-
ferentiation, as we did in Section 4.2.2. Third, PyTorch’s highly optimized and ready-to-use multi-GPU parallelization largely circumvents the painstaking hand-crafted effort.

PTFlash also has several limitations. First, PTFlash is based on the SRK equation of state, which is relatively simple and sufficient for mixtures containing hydrocarbons and non-polar components, but does not take into account the effect of hydrogen bonding and falls short of adequacy for cross-associating mixtures having polar components, such as water and alcohol [49]. In this case, more advanced but also more complicated equations of state should be employed, such as the SAFT equation of state [50–55] or the CPA equation of state [56, 57]. However, vectorization of these complicated equations of state is far more difficult than that of cubic equations of state. To alleviate this problem, we plan to use neural networks to directly predict the fugacity coefficients. In this way, we can calculate the fugacity coefficients in a vectorized fashion, regardless of the equation of state used. Second, PTFlash consumes a large amount of GPU memory, badly limiting its use on much larger batches of data. We need to optimize PTFlash to reduce the consumption of GPU memory, e.g., by leveraging the sparsity and symmetry of matrices. Third, PTFlash does not support multi-phase equilibrium. Last but not least, neural networks are subject to the out-of-distribution generalization problem. If pressure and temperature are out of predefined ranges used to train neural networks, predictive performance will deteriorate dramatically. Furthermore, once the components of the mixture change, we need to create new neural networks and train them from scratch.

6. Conclusion

In this work, we presented a fast and parallel framework, PTFlash, for two-phase flash calculation based on PyTorch, which efficiently vectorizes algorithms and gains attractive speed-ups at large scale calculations. Two neural networks were used to predict the stability of given mixtures and to initialize the distrib-
tion coefficients more accurately than the Wilson approximation, which greatly accelerate PTFlash. In addition, PTFlash has much broader utility compared to the aforementioned methods which are mainly tailored to compositional reservoir simulation.

We compared PTFlash with Carnot, an in-house thermodynamic library, and we investigated three case studies containing 2, 4 and 9 components with maximum number of samples of 10, 5 and 1 million, respectively. The results showed that PTFlash on GPU is 163.4, 106.3 and 50.5 times faster than Carnot at the maximum number of samples for these three cases, respectively.

In the future, we will optimize PTFlash to reduce the consumption of GPU memory and extend our work to vectorize more advanced equations of state and support multi-phase equilibrium. We will also explore the feasibility of using neural networks to directly predict the fugacity coefficients, which can serve as an alternative to numerically solving equations of state. In addition, we will validate PTFlash on more hardware suitable for parallel computing, e.g., TPU. Last but not least, we will apply our work to downstream applications, e.g., compositional reservoir simulation.

7. Acknowledgements

We acknowledge the financial support from French National Research Agency (ANR) through the projects DL4CLIM ANR-19-CHIA-0018-01 and DEEP- NUM ANR-21-CE23-0017-02.
Appendix A. SRK equation of state and its solution

The SRK equation of state describes the relationship between pressure \( P \), temperature \( T \) and volume \( V \) in the following mathematical form [21]:

\[
P = \frac{RT}{V - b} - \frac{a\alpha}{V(V + b)} \quad (A.1)
\]

where \( R \) is the gas constant, \( a\alpha \) refers to the temperature-dependent energy parameter, and \( b \) denotes the co-volume parameter. We employ the van der Waals mixing rules and the classical combining rules to calculate \( a\alpha \) and \( b \), as follows:

\[
a\alpha = \sum_{i=1}^{N_c} \sum_{j=1}^{N_c} c_i c_j (a\alpha)_{ij} \quad (A.2a)
\]

\[
(a\alpha)_{ij} = (1 - k_{ij}) \sqrt{(a\alpha)_i(a\alpha)_j} \quad (A.2b)
\]

\[
b = \sum_{i=1}^{N_c} c_i b_i \quad (A.2c)
\]

\[
a_i = \frac{0.42748 \cdot R^2 (T_{c,i})^2}{P_{c,i}} \quad (A.2d)
\]

\[
b_i = \frac{0.08664 \cdot R T_{c,i}}{P_{c,i}} \quad (A.2e)
\]

\[
\alpha_i = \left[ 1 + m_i \left( 1 - \sqrt{\frac{T}{T_{c,i}}} \right) \right]^2 \quad (A.2f)
\]

\[
m_i = 0.480 + 1.574 \omega_i - 0.176 \omega_i^2 \quad (A.2g)
\]

where the subscripts \( i \) and \( j \) refer to the components \( i \) and \( j \), respectively, \( c_i \) denotes the mole fraction of the component \( i \) in the phase considered, \( k_{ij} \) is the binary interaction parameter between the components \( i \) and \( j \), \( a_i \) and \( b_i \) are two substance-specific constants related to the critical temperature \( T_{c,i} \) and critical pressure \( P_{c,i} \), and \( \omega_i \) is the acentric factor. We reformulate Equation A.1 as a cubic equation in terms of the compressibility factor \( Z \):

\[
f_{srk}(Z) = Z^3 - Z^2 + \rho_1 Z - \rho_0 = 0 \quad (A.3)
\]
where \( \rho_0 = AB \) and \( \rho_1 = A - B(1 + B) \), in which \( A = a\alpha P/(R^2T^2) \) and \( B = bP/(RT) \). To find the roots of \( f_{srk}(Z) \), we utilize an iterative approach based on Halley’s method \[25\], as follows:

\[
Z^{(k+1)} = Z^{(k)} - \frac{f_{srk}(Z^{(k)})}{f'_{srk}(Z^{(k)})} \left[ 1 - \frac{f_{srk}(Z^{(k)})}{f'_{srk}(Z^{(k)})} \cdot \frac{f''_{srk}(Z^{(k)})}{2f'_{srk}(Z^{(k)})} \right]^{-1} \tag{A.4}
\]

The above iteration starts with a liquid-like guess and converges to a real root \( Z_0 \) (The convergence criterion is \( |Z^{(k+1)}/Z^{(k)} - 1| < 1.0e-8 \), and then we deflate the cubic equation as:

\[
f_{srk}(Z) = (Z - Z_0)(Z^2 + pZ + q) = 0 \tag{A.5}
\]

where \( p = Z_0 - 1 \) and \( q = pZ_0 + \rho_1 \). If \( p^2 < 4q \), only one real root \( Z_0 \) exists, otherwise, there are three real roots and the other two are \( -p/2 \pm \sqrt{p^2 - 4q}/2 \). In the latter case, we assign the smallest root to the liquid phase and the biggest one to the vapor phase. Subsequently, the root corresponding to the lowest Gibbs energy will be chosen. When \( Z \) is known, the fugacity coefficients \( \varphi_i \) are calculated as follows:

\[
\ln \varphi_i(P, T, c) = \frac{b_i}{b}(Z - 1) - \ln(Z - B) + \frac{A}{B} \left( \frac{b_i}{b} - \frac{2}{a\alpha} \sum_{j=1}^{N_c} (a\alpha)_{ij} c_j \right) \ln(1 + B/Z) \tag{A.6}
\]

where \( c \) is the composition of the phase considered. In addition, the derivatives of the fugacity coefficients with respect to mole numbers, which are necessary for the trust-region methods of stability analysis and phase split calculations, are calculated explicitly rather than through PyTorch’s automatic differentiation, which requires retaining intermediate results and consumes prohibitive memory at large scale computation.
Appendix B. Trust-region method

When the successive substitution fails to converge quickly, particularly around critical points for which liquid and vapor phases are almost indistinguishable, we will switch to the trust-region method with restricted steps, which is a second-order optimization technique, to achieve faster convergence.

In the following, the problem formulations are taken from Michelsen and Mollerup’s book [31], but the concrete implementation of the trust-region method, such as how to adjust the trust-region size and calculate the step size, is adapted from [28].

Appendix B.1. Trust-region method for stability analysis

The objective function to be minimized is the modified tangent plane distance [1]:

\[
    tm(W) = \sum_{i=1}^{N_c} W_i (\ln W_i + \ln \varphi_i(W) - \ln z_i - \ln \varphi_i(z) - 1)
\]

The minimization is accomplished by iterating the following equations:

\[
    \beta^{(k)} = 2 \sqrt{W^{(k)}} \quad (B.1a)
\]

\[
    (H^{(k)} + \eta^{(k)} I) \cdot \Delta \beta + g^{(k)} = 0 \quad \text{s.t.} \quad \|\Delta \beta\| \leq \Delta_{max}^{(k)} \quad (B.1b)
\]

\[
    \beta^{(k+1)} = \beta^{(k)} + \Delta \beta \quad (B.1c)
\]

\[
    W^{(k+1)} = \left( \frac{\beta^{(k+1)}}{2} \right)^2 \quad (B.1d)
\]

where \( I \) is the identity matrix, \( g \) and \( H \) are the gradient and Hessian matrix of \( tm \) with respect to \( \beta \), respectively, and are calculated as follows:

\[
    g_i = \sqrt{W_i} (\ln W_i + \ln \varphi_i(W) - \ln z_i - \ln \varphi_i(z)) \quad (B.2a)
\]

\[
    H_{ij} = \sqrt{W_i W_j} \frac{\partial \ln \varphi_i}{\partial W_i} + \sigma_{ij} \left( 1 + \frac{g_i}{\beta_i} \right) \quad \text{where} \quad \sigma_{ij} = 1 \iff i = j \quad (B.2b)
\]
In addition, $\eta$ is the trust-region size used to guarantee the positive definite-
ness of $H + \eta I$ and to tailor the step size to meet $\|\Delta \beta\| \leq \Delta_{\text{max}}$, where $\Delta_{\text{max}}$
is adjusted during iterations depending on the match between the actual reduc-
tion $\delta_{tm} = tm^{(k+1)} - tm^{(k)}$ and the predicted reduction based on the quadratic
approximation $\hat{\delta}_{tm} = \Delta \beta^T g + \frac{1}{2} \Delta \beta^T H \Delta \beta$, using the following heuristic rules:

$$
\Delta_{\text{max}}^{(k)} = \begin{cases} 
\frac{\Delta_{\text{max}}^{(k)}}{2}, & \text{if } |\delta_{tm}/\hat{\delta}_{tm}| \leq 0.25 \\
2\Delta_{\text{max}}^{(k)}, & \text{if } |\delta_{tm}/\hat{\delta}_{tm}| \geq 0.75 \\
\Delta_{\text{max}}^{(k)}, & \text{otherwise}
\end{cases} 
$$

(B.3)

The convergence criterion of Equation B.1 is $\max(|g|) < 1.0 \times 10^{-6}$.

Appendix B.2. Trust-region method for phase split calculations

The objective function to be minimized is the reduced Gibbs energy:

$$
G = \sum_{i=1}^{N_c} \sum_{L} n_{Li}^L (\ln x_i + \ln \phi_{Li}^L) + \sum_{i=1}^{N_c} \sum_{V} n_{Vi}^V (\ln y_i + \ln \phi_{Vi}^V)
$$

where $n_{Li}^L = x_i (1 - \theta_V)$ and $n_{Vi}^V = y_i \theta_V$ are the mole numbers of liquid and vapor
phases, respectively. We choose $n_{Vi}^V$ as the independent variable and perform
the following iteration:

$$
\begin{align*}
(\tilde{H}^{(k)} + \eta^{(k)} \cdot D \left( \frac{z}{xy} \right)) \cdot \Delta n^V + \tilde{g}^{(k)} &= 0 \\
\text{s.t. } \|\Delta n^V\| &\leq \tilde{\Delta}_{\text{max}}^{(k)} \\
n^{V,k+1} &= n^{V,k} + \Delta n^V
\end{align*}
$$

(B.4a, B.4b)

where $\tilde{H}^{(k)}$ and $\tilde{g}^{(k)}$ are the gradient and hessian matrix of $G$ with respect to
$n_{Vi}^V$, respectively, and are calculated as follows:

$$
\tilde{H}_{ij} = \frac{1}{\theta_V (1 - \theta_V)} \left( \frac{z_i}{x_i y_i} \sigma_{ij} - 1 + \theta_V \frac{\partial \ln \phi_{Li}^L}{\partial n_{Lj}^L} + (1 - \theta_V) \frac{\partial \ln \phi_{Vi}^V}{\partial n_{Vj}^V} \right)
$$

(B.5a, B.5b)

In addition, $D(\cdot)$ is a diagonal matrix with diagonal entries in parentheses.

The above iteration stops if $\max(|\tilde{g}|) < 1.0 \times 10^{-8}$. Here, the trust-region method is
implemented in the same way as in stability analysis.
Appendix C. The Rachford-Rice equation

Appendix C.1. Solution of the Rachford-Rice equation

The Rachford-Rice equation is as follows:

$$f_{RR}(\theta_V, K) = \sum_{i=1}^{N_c} \frac{(K_i - 1)z_i}{1 + (K_i - 1)\theta_V} = 0$$

Given $K$, the solution of the above equation amounts to finding an appropriate zero yielding all non-negative phase compositions. Concretely, we adopt the method proposed by [30], which transforms $f_{RR}$ into a helper function $h_{RR}$ which is more linear in the vicinity of the zero:

$$h_{RR}(\theta_V, K) = (\theta_V - \alpha_l) \cdot (\alpha_r - \theta_V) \cdot f_{RR}(\theta_V) = 0 \ (C.1)$$

where $\alpha_l = 1/(1 - \max(K_i))$ and $\alpha_r = 1/(1 - \min(K_i))$. The above equation is solved by alternating between the Newton method and the bisection method used when the Newton step renders $\theta_V$ out of the bounds which contain the zero and become narrower during iterations. When the Newton step size is smaller than $1.0e-8$, the iteration stops.

Appendix C.2. Calculation of $\partial\theta_V/\partial K$ using the implicit function theorem

Based on the implicit function theorem [47], we can calculate $\partial\theta_V/\partial K$ in an efficient way. We first differentiate the Rachford-Rice equation with respect to $K$ (note that $\theta_V$ is an implicit function of $K$) and get:

$$\partial_{\theta_V} f_{RR}(\theta_V, K) \times \partial\theta_V/\partial K + \partial_K f_{RR}(\theta_V, K) = 0 \ (C.2)$$

We rearrange the above equation and get Equation 13 as follows:

$$\partial\theta_V/\partial K = -[\partial_{\theta_V} f_{RR}(\theta_V, K)]^{-1} \partial_K f_{RR}(\theta_V, K)$$

Moreover, since $\partial_{\theta_V} f_{RR}(\theta_V, K)$ is a scalar, we can further reduce the above equation to:

$$\partial\theta_V/\partial K = -\frac{\partial_K f_{RR}(\theta_V, K)}{\partial_{\theta_V} f_{RR}(\theta_V, K)} \ (C.3)$$

For the sake of brevity, we have simplified some details. For more details and a defense of the above derivation, refer to [47].
### Appendix D. Some typical reservoir fluid compositions

<table>
<thead>
<tr>
<th></th>
<th>Wet gas</th>
<th>Gas condensate</th>
<th>Volatile oil</th>
<th>Black oil</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CH_4$</td>
<td>92.46%</td>
<td>73.19%</td>
<td>57.6%</td>
<td>33.6%</td>
</tr>
<tr>
<td>$C_2H_6$</td>
<td>3.18%</td>
<td>7.8%</td>
<td>7.35%</td>
<td>4.01%</td>
</tr>
<tr>
<td>$C_3H_8$</td>
<td>1.01%</td>
<td>3.55%</td>
<td>4.21%</td>
<td>1.01%</td>
</tr>
<tr>
<td>n-$C_4H_{10}$</td>
<td>0.52%</td>
<td>2.16%</td>
<td>2.81%</td>
<td>1.15%</td>
</tr>
<tr>
<td>n-$C_5H_{12}$</td>
<td>0.21%</td>
<td>1.32%</td>
<td>1.48%</td>
<td>0.65%</td>
</tr>
<tr>
<td>$C_6H_{14}$</td>
<td>0.14%</td>
<td>1.09%</td>
<td>1.92%</td>
<td>1.8%</td>
</tr>
<tr>
<td>$C_7H_{16}^+$</td>
<td>0.82%</td>
<td>8.21%</td>
<td>22.57%</td>
<td>57.4%</td>
</tr>
<tr>
<td>$CO_2$</td>
<td>1.41%</td>
<td>2.37%</td>
<td>1.82%</td>
<td>0.07%</td>
</tr>
<tr>
<td>$N_2$</td>
<td>0.25%</td>
<td>0.31%</td>
<td>0.24%</td>
<td>0.31%</td>
</tr>
</tbody>
</table>
Appendix E. Vectorized algorithms

Appendix E.1. Synchronizer
Algorithm 1: PyTorch pseudo-code of synchronizer to save converged results after iteration and remove the corresponding samples

**Input:** Vectorized iterated function $f(X, O)$, initial estimate $X^{(0)}$, other $f$-related inputs $O$, convergence criterion $C$, maximum number of iterations $K$

1. **Initialization**
   2. Set the number of iterations $k \leftarrow 1$
   3. Generate a vector $i$ containing indices from $0$ to $n - 1$
      
      /* $n$ is the number of samples and indexing starts from $0$. */
   4. Create a placeholder matrix $\bar{X}$ of the same shape as $X^{(0)}$

5. while $k \leq K$ do
   6. $X^{(k+1)} \leftarrow f(X^{(k)}, O)$
   7. mask $\leftarrow C(\cdots)$
      
      /* $C$ returns a Boolean vector and True means convergence. */

   8. **Saving**
      9. indices $\leftarrow i[mask]$
      10. $\bar{X}[\text{indices}] \leftarrow X^{(k+1)}[\text{mask}]$

   9. **Removing**
      10. $i \leftarrow i[\sim \text{mask}]$
      11. $O \leftarrow O[\sim \text{mask}]$
      12. /* Apply this operation to every element in $O$ */
      13. $X^{(k+1)} \leftarrow X^{(k+1)}[\sim \text{mask}]$
      14. $k \leftarrow k + 1$

15. if $\text{len}(i) \neq 0$ then
   16. $\bar{X}[i] \leftarrow X$
      
      /* Also save unconverged results for further utilization. */

**Output:** Converged results $\bar{X}$ and unconverged indices $i$
Algorithm 2: PyTorch pseudo-code of vectorized stability analysis

**Input:** Pressure $P$, temperature $T$, feed composition $z$, component properties ($P_c$, $T_c$, $\omega$, BIPs), initial estimate $W^{(0)}$, convergence criteria $C_{ss}$ and $C_{tr}$, maximum numbers of iterations $K_{ss} = 9$ and $K_{tr} = 20$

1 **Initialization**

   Instantiate $pteos = PTEOS(P_c, T_c, \omega, \text{BIPs})$

   /* PTEOS is a PyTorch-based class to efficiently calculate the
      fugacity coefficients and their partial derivatives. */

3 **Successive substitution**

   Iterated function $f_{ss}$ specified by Equation 5

   Other inputs $\mathcal{O}_{ss} \leftarrow \{P, T, z\}$

   $W, i_{ss} \leftarrow \text{synchronizer}(f_{ss}, W^{(0)}, \mathcal{O}_{ss}, C_{ss}, K_{ss})$

5 Trust-region method

   Iterated function $f_{tr}$ specified by Equation B.1

   $W_{tr}^{(0)} \leftarrow W[i_{ss}]$

   Other inputs $\mathcal{O}_{tr} \leftarrow \{P[i_{ss}], T[i_{ss}], z[i_{ss}]\}$

   $W_{tr}, i_{tr} \leftarrow \text{synchronizer}(f_{tr}, W_{tr}^{(0)}, \mathcal{O}_{tr}, C_{tr}, K_{tr})$

   $W[i_{ss}] \leftarrow W_{tr}$ and $i \leftarrow i_{ss}[i_{tr}]$

**Output:** Converged results $W$ and unconverged indices $i$
References


[27] O. Orbach, C. Crowe, Convergence promotion in the simulation of chemical processes with recycle-the dominant eigenvalue method 49 (4) 509–513, publisher: Wiley Online Library.


[41] L. N. Smith, No more pesky learning rate guessing games 5.


[50] M. S. Wertheim, Fluids with highly directional attractive forces. II. thermodynamic perturbation theory and integral equations 35 (1) 35–47, publisher: Springer.

[51] M. Wertheim, Fluids with highly directional attractive forces. i. statistical thermodynamics 35 (1) 19–34, publisher: Springer.

[52] M. Wertheim, Fluids with highly directional attractive forces. IV. equilibrium polymerization 42 (3) 477–492, publisher: Springer.

[53] M. Wertheim, Fluids with highly directional attractive forces. III. multiple attraction sites 42 (3) 459–476, publisher: Springer.

