Fuel 331 (2023) 125603

Contents lists available at ScienceDirect

Fuel

journal homepage: www.elsevier.com/locate/fuel

Full length article

PTFlash : A vectorized and parallel deep learning framework for two-phase flash calculation

Jingang Qu [a,b,*], Thibault Faney [b], Jean-Charles de Hemptinne [b], Soleiman Yousef [b], Patrick Gallinari [a,c]

[a] Sorbonne Université, CNRS, ISIR, F-75005 Paris, France
[b] IFPEN, France
[c] Criteo AI Lab, Paris, France

ARTICLE INFO

Keywords:
Flash calculation
Two-phase equilibrium
Vectorization
Deep learning

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, CO_2 and N_2, 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 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*.

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. [10] 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

* Corresponding author at: Sorbonne Université, CNRS, ISIR, F-75005 Paris, France.
E-mail address: jingang.qu@sorbonne-universite.fr (J. Qu).

https://doi.org/10.1016/j.fuel.2022.125603
Received 18 May 2022; Received in revised form 25 July 2022; Accepted 10 August 2022
Available online 7 September 2022

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 calculation using PyTorch. In Section 4, we present two neural networks to speed up calculations. In Section 5, we demonstrate the attractive speed-up due to vectorization and the introduction of neural networks. Finally, we summarize our work and suggest future research in Section 6.

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 ($\boldsymbol{z} = (z_1, \dots, z_{N_c})$), 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 θ_V, the composition of the liquid phase $\boldsymbol{x}$ and that of the vapor phase $\boldsymbol{y}$. These properties are constrained by the following mass balance equations:

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

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

In addition, the following equilibrium condition should be satisfied:

$$\frac{\varphi_i^L(P,T,\boldsymbol{x})}{\varphi_i^V(P,T,\boldsymbol{y})} = \frac{y_i}{x_i} \tag{2}$$

where the superscripts L and V refer to the liquid and vapor phases, respectively, and φ_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 [21] and solve it using an iterative approach [25] rather than the analytical solution of the cubic equation, e.g., the 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

Eqs. (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 θ_V, $\boldsymbol{x}$ and $\boldsymbol{y}$ at equilibrium through phase split calculations (Section 2.2.2).

2.2.1. Stability analysis

A mixture of composition $\boldsymbol{z}$ 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]:

$$tpd(\boldsymbol{w}) = \sum_{i=1}^{N_c} w_i(\ln w_i + \ln\varphi_i(\boldsymbol{w}) - \ln z_i - \ln\varphi_i(\boldsymbol{z})) \tag{3}$$

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

$$tm(\boldsymbol{W}) = \sum_{i=1}^{N_c} W_i(\ln W_i + \ln\varphi_i(\boldsymbol{W}) - \ln z_i - \ln\varphi_i(\boldsymbol{z}) - 1) \tag{4}$$

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

$$\ln W_i^{(k+1)} = \ln z_i + \ln\varphi_i(\boldsymbol{z}) - \ln\varphi_i(\boldsymbol{W}^{(k)}) \tag{5}$$

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 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 ω_i is the acentric factor.

Once converging to a stationary point (i.e., $\max(|\partial tm/\partial \boldsymbol{W}|) < 1.0e{-}6$) or a negative tm 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 $tm(\boldsymbol{W})$, 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 $tm^L < tm^V$ 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 Eq. (1) yields the following Rachford–Rice equation [29]:

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

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

To obtain θ_V, $\boldsymbol{x}$ and $\boldsymbol{y}$ at equilibrium, phase split calculations start with the accelerated successive substitution method, as illustrated in Fig. 1, and the corresponding convergence criterion is $\max(|K_i^{(k+1)}/K_i^{(k)} - 1|) < 1.0e{-}8$. 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:

$$G = \sum_{i=1}^{N_c} n_i^L(\ln x_i + \ln\varphi_i^L) + \sum_{i=1}^{N_c} n_i^V(\ln y_i + \ln\varphi_i^V) \tag{8}$$

where $n_i^L = x_i(1-\theta_V)$ and $n_i^V = y_i\theta_V$ are the mole numbers of liquid and vapor phases, respectively. The convergence criterion is $\max(|\partial G/\partial n_i^V|) < 1.0e{-}8$. For more details, see Appendix B.2.

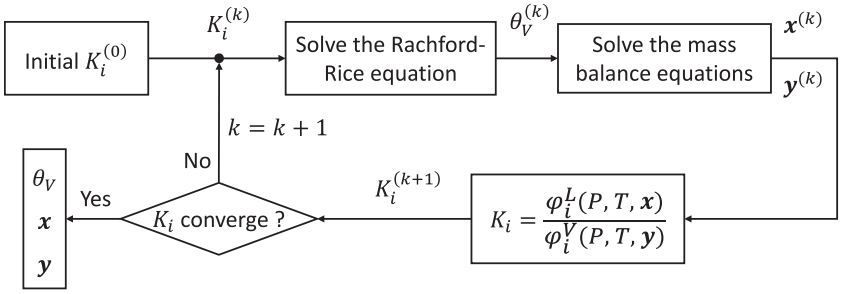


Fig. 1. Successive substitution of phase split calculations.

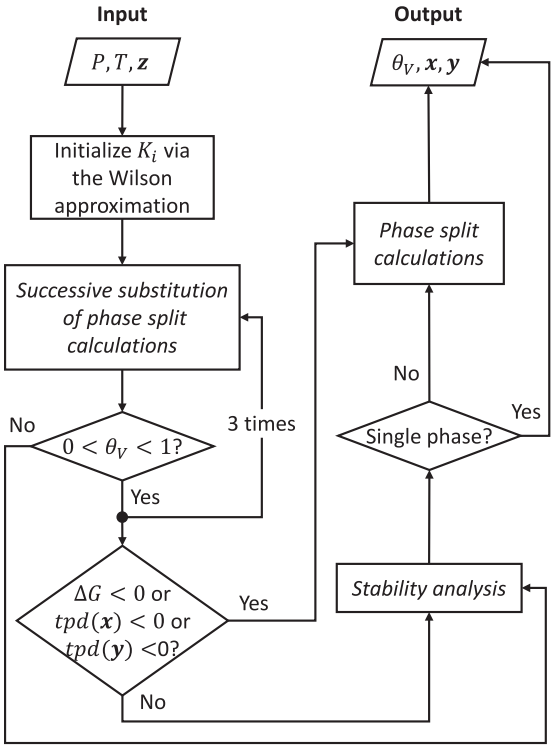


Fig. 2. Flowchart of two-phase flash calculation.

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 Fig. 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) θ_V is out of bounds (0, 1) during iterations. (2) None of ΔG, $tpd(\boldsymbol{x})$ and $tpd(\boldsymbol{y})$ are negative, where $tpd(\boldsymbol{x})$ and $tpd(\boldsymbol{y})$ are reduced tangent plane distances using current vapor and liquid phases as trial phases, and $\Delta G = \theta_V \times tpd(\boldsymbol{x}) + (1-\theta_V) \times tpd(\boldsymbol{y})$. (3) Any of ΔG, $tpd(\boldsymbol{x})$ and $tpd(\boldsymbol{y})$ 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 θ_V, $\boldsymbol{x}$ and $\boldsymbol{y}$ 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_6H_{14}), and the second one involves four components (CH_4, C_2H_6, C_3H_8 and C_4H_{10}). For these two case studies, the ranges of pressure and temperature are 0.1 MPa–10 MPa and 200 K–500 K, 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 5 MPa–25 MPa and 200 K–600 K, 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 of the common reservoir fluid types, namely wet gas, gas condensate, volatile oil, and black oil. Fig. 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 1
Component properties.

	P_c (MPa)	T_c (K)	*omega*
CH_4	4.6	190.55	0.0111
C_2H_6	4.875	305.43	0.097
C_3H_8	4.268	369.82	0.1536
$n-C_4H_{10}$	3.796	425.16	0.2008
$n-C_5H_{12}$	3.3332	467.15	0.2635
C_6H_{14}	2.9688	507.4	0.296
$C_7H_{16}^+$	2.622	604.5	0.3565
CO_2	7.382	304.19	0.225
N_2	3.3944	126.25	0.039

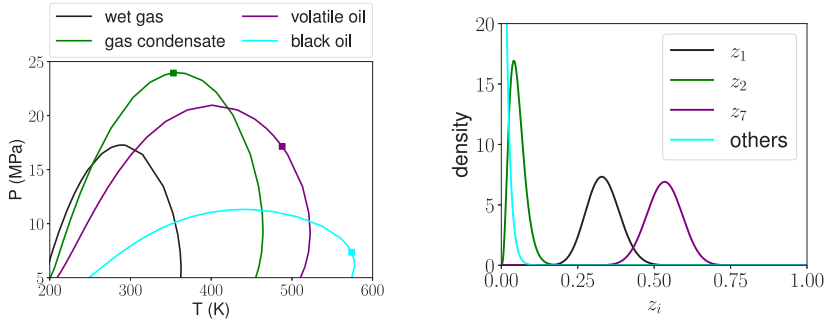
Table 2
Four fluid types characterized by different compositional ranges.

	Wet gas	Gas condensate	Volatile oil	Black oil
CH_4	80%–100%	60%–80%	50%–70%	20%–40%
C_2H_6	2%–7%	5%–10%	6%–10%	3%–6%
C_3H_8	≤3%	≤4%	≤4.5%	≤1.5%
$n-C_4H_{10}$	≤2%	≤3%	≤3%	≤1.5%
$n-C_5H_{12}$	≤2%	≤2%	≤2%	≤1%
C_6H_{14}	≤2%	≤2%	≤2%	≤2%
$C_7H_{16}^+$	≤1%	5%–10%	10%–30%	45%–65%
CO_2	≤2%	≤3.5%	≤2%	≤0.1%
N_2	≤0.5%	≤0.5%	≤0.5%	≤0.5%

2.4. Data generation

To efficiently sample input data including P, T and $\boldsymbol{z}$, we first use Latin Hypercube Sampling (LHS) technique to take space-filling samples [32]. Subsequently, for P and T, we linearly transform the uniform distribution $\mathcal{U}(0,1)$ to the expected ranges. For $\boldsymbol{z}$ subject to $\sum z_i = 1$, we transform a set of $\mathcal{U}(0,1)$ into the Dirichlet distribution $Dir(\boldsymbol{\alpha})$ whose support is a simplex, as follows:

$$x_i \overset{i.i.d.}{\sim} \mathcal{U}(0,1) \text{ using LHS} \tag{9a}$$



(a) Phase diagrams (b) Marginal distribution of z_i for black oil

Fig. 3. In Figure (a), the squares on the phase envelopes represent critical points. In Figure (b), z_1, z_2 and z_7 are the molar fractions of CH_4, C_2H_6 and $C_7H_{16}^+$, respectively.

Table 3
Concentration parameters $\boldsymbol{\alpha}$ for different fluid types in Table 2.

	α_1 for CH_4	α_2 for C_2H_6	α_7 for $C_7H_{16}^+$	α_i for others
Wet gas	100	5	1	1
Gas condensate	40	5	5	1
Volatile oil	55	8	20	1
Black oil	25	4	40	1

$$y_i = \Gamma(\alpha_i, 1).ppf(x_i) \tag{9b}$$

$$z_i = \frac{y_i}{\sum_{i=1}^{N_c} y_i} \tag{9c}$$

where $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_{N_c})$ 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 $\boldsymbol{z} = (z_1, \dots, z_{N_c}) \sim Dir(\boldsymbol{\alpha})$.

For the first two case studies, the concentration parameters are $\boldsymbol{\alpha} = \mathbf{1}$, i.e., all-ones vector. For the third case study, we adjust $\boldsymbol{\alpha}$ for different fluid types to make the probability of each compositional range as large as possible, as shown in Table 3. Fig. 3(b) presents the marginal distribution of z_i for black oil. In summary, we sample $\boldsymbol{z}$ using Eq. (9) with different $\boldsymbol{\alpha}$ 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.

Eventually, the samples of P, T and $\boldsymbol{z}$ 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 $\boldsymbol{P} = (P_1, \dots, P_n)$, $\boldsymbol{T} = (T_1, \dots, T_n)$ and $\mathbf{z} = (\boldsymbol{z}_1, \dots, \boldsymbol{z}_n)$, where $\boldsymbol{P}$ and $\boldsymbol{T}$ are vectors, $\mathbf{z}$ 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 [33], 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 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:

$$\boldsymbol{X}^{(k+1)} \leftarrow f(\boldsymbol{X}^{(k)}) \tag{10a}$$

$$\text{Save} \quad \boldsymbol{X}^{(k+1)}[\text{mask}] \quad \text{to} \quad \widetilde{\boldsymbol{X}} \tag{10b}$$

$$\boldsymbol{X}^{(k+1)} \leftarrow \boldsymbol{X}^{(k+1)}[\sim \text{mask}] \tag{10c}$$

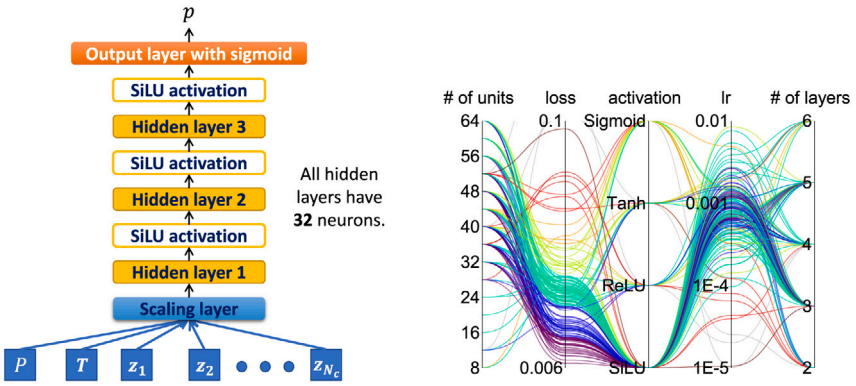
$$k \leftarrow k + 1 \tag{10d}$$

where k is the number of iterations, $f(\boldsymbol{X})$ is a vectorized iterated function taking as input $\boldsymbol{X} \in \mathbb{R}^{n \times m}$ (n is the batch dimension, i.e., number of samples, and m is the dimension of X), $\widetilde{\boldsymbol{X}}$ 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 converged samples are still waiting for unconverged 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 $\boldsymbol{X}^{(0)}$ 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 Fig. 2, which can avoid any subroutine connected to it proceeding further until all others terminate and arrive at this barrier.

[1] At the time of writing, PyTorch team released a fledgling library, *functorch*, which takes inspiration from JAX and supports Automatic Vectorization.



(a) The architecture of classifier for the 9-component mixture

(b) Tuning hyper-parameters of classifier

Fig. 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. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

We also optimized the code using TorchScript [22], allowing for more efficient execution through algebraic peephole optimizations and fusion of some operations, 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 = classifier$(P, T, \boldsymbol{z})$, 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., $\ln K_i$ = initializer$(P, T, \boldsymbol{z})$, which relates to a regression problem. It can reduce the number of iterations required to reach convergence and alleviate the asynchronous 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 Fig. 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 $\boldsymbol{x}$ consists of P, T and $\boldsymbol{z}$, and the output is the probability p that a given mixture is stable. The scaling layer standardizes the inputs as $(\boldsymbol{x} - \boldsymbol{u})/\boldsymbol{s}$, where $\boldsymbol{u}$ and $\boldsymbol{s}$ are the mean and standard deviation of $\boldsymbol{x}$ 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:

$$\text{bce}(y, p) = y \ln p + (1 - y) \ln(1 - p) \tag{11}$$

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

The architecture of the classifier is obtained by tuning hyper-parameters using Tree-Structured Parzen Estimator optimization algorithm [38] with Asynchronous 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 Fig. 4(b). We can see that SiLU largely outperforms other activation functions.

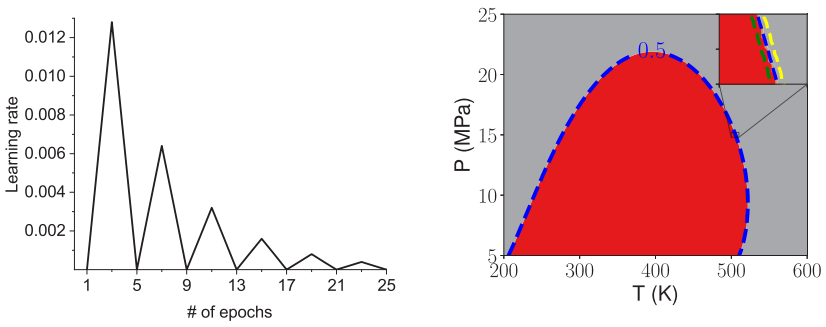
4.1.2. Training

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 s. 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 Fig. 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 min 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, Fig. 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.

4.2. Initializer

4.2.1. Architecture

The input of the initializer includes P, T and $\boldsymbol{z}$, and its output is $\ln K_i$. The initializer has 1 hidden layer and 3 residual blocks, as shown in Fig. 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



(a) Cyclic learning rate of the classifier (b) Prediction of the classifier for volatile oil

Fig. 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. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

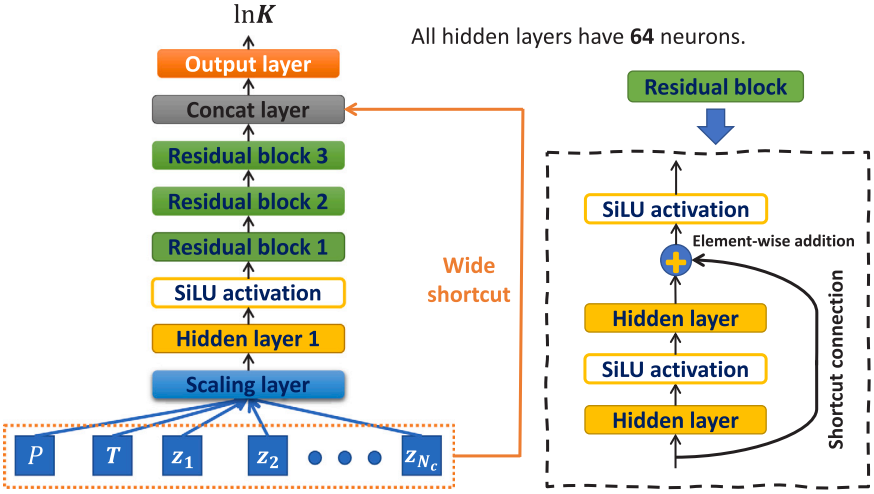


Fig. 6. The architecture of initializer for the 9-component mixture.

in different orders of magnitude, which hampers the training of the initializer, whereas $\ln K_i$ does not.

We found that the convergence of phase split calculations is robuster if K_i predicted by the initializer can lead to more accurate values of the vapor fraction θ_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 θ_V, as follows:

$$\text{mae}(\ln \boldsymbol{K}, \ln \hat{\boldsymbol{K}}) = \sum_{i=1}^{N_c} |\ln K_i - \ln \hat{K}_i| \tag{12a}$$

$$\text{mae}(\theta_V, \hat{\theta}_V) = |\theta_V - \hat{\theta}_V| \tag{12b}$$

where $\ln \boldsymbol{K}$ is the ground truth, $\ln \hat{\boldsymbol{K}}$ is the prediction of the initializer, θ_V is the vapor fraction at equilibrium, and $\hat{\theta}_V$ is obtained by solving the Rachford–Rice equation given $\boldsymbol{z}$ and the prediction $\hat{\boldsymbol{K}}$.

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 (15%) and test (15%) sets. The training of the initializer is carried out in two stages. First, we train it to minimize $\text{mae}(\ln \boldsymbol{K}, \ln \hat{\boldsymbol{K}})$, using Adam with CLR and setting the batch size to 512. Second, after the above training, we further train it to minimize $\text{mae}(\ln \boldsymbol{K}, \ln \hat{\boldsymbol{K}}) + \text{mae}(\theta_V, \hat{\theta}_V)$, using Adam with a small learning rate $1.0e{-}5$. Here, $\partial \hat{\theta}_V / \partial \hat{\boldsymbol{K}}$ 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 $\partial \hat{\theta}_V / \partial \hat{\boldsymbol{K}}$ by using the derivative information at the solution point of the Rachford–Rice equation, as follows:

$$\partial \hat{\theta}_V / \partial \hat{\boldsymbol{K}} = -[\partial_{\theta_V} f_{RR}(\hat{\theta}_V, \hat{\boldsymbol{K}})]^{-1} \partial_{\boldsymbol{K}} f_{RR}(\hat{\theta}_V, \hat{\boldsymbol{K}}) \tag{13}$$

This way is obviously more efficient and avoids differentiation through iterations. We give more details about the derivation of Eq. (13) in Appendix C.2.

Eventually, 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 Fig. 7, given P, T and $\boldsymbol{z}$, we first use the classifier to predict p. Next, based on two predefined thresholds, p_l and p_r, satisfying $p_l \le p_r$, the given mixture is thought of as unstable if $p \le p_l$ or stable if $p \ge p_r$. 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 $p \ge p_c$ or unstable otherwise), and no extra stability

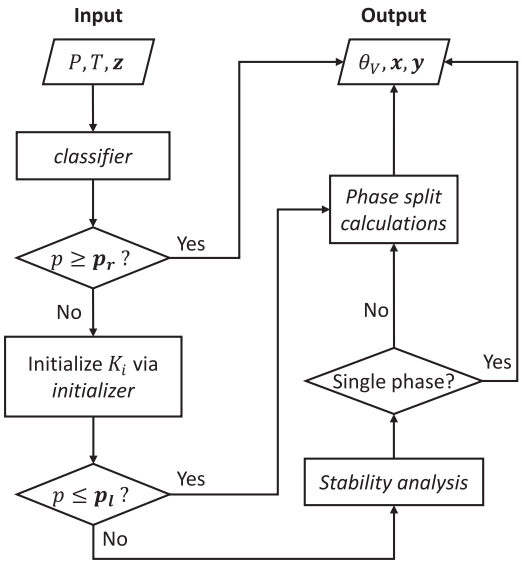


Fig. 7. Acceleration of flash calculation using neural networks.

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 Fig. 7. If only the initializer is available, one may use it to initialize K_i in Fig. 2.

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 Fig. 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 4.5 GHz, 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 Figs. 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' Ref. [48], which claimed that the total computation time of flash calculations is 10 s 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, Figs. 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:

$$d_c = \sqrt{\sum_{i=1}^{N_c} \ln K_i^2} \tag{14}$$

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

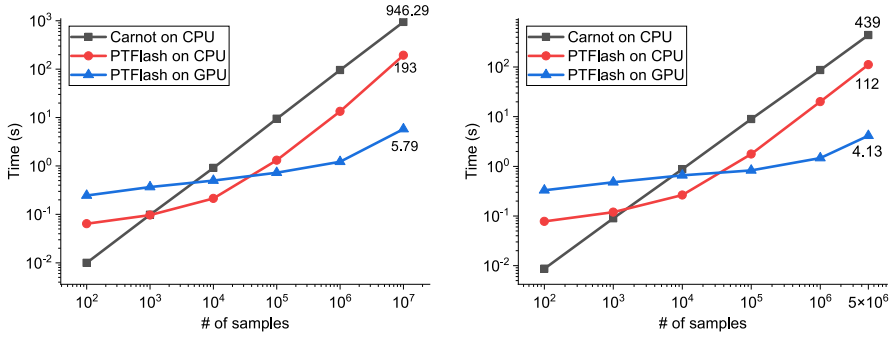
The observations of Figs. 9 are summarized as follows: (1) In Fig. 9(a), the slope of time with respect to the number of iterations is decreasing because the workload is reduced due to incremental convergence. (2) In Fig. 9(b), for the samples that do not converge after successive substitution, the majority of them (92.67%) converge after 3 iterations of the trust-region method. (3) In Fig. 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.

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.

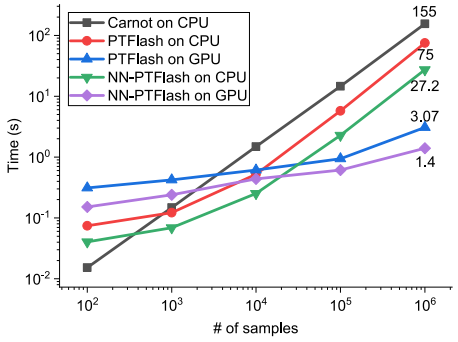
5.2. Deep-learning-powered vectorized flash calculation

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 Fig. 8(c), we can see that *NN-PTFlash* outpaces *PTFlash* on both CPU (2.7× speed-up) and GPU (2.2× 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

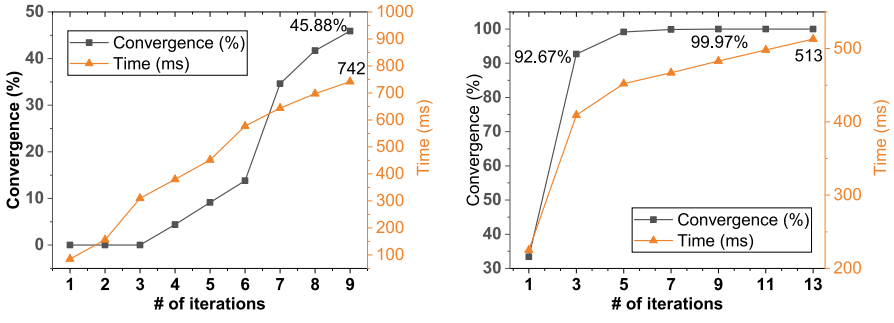


(a) Mixture of CH_4 and C_6H_{14} (b) Mixture of 4 components

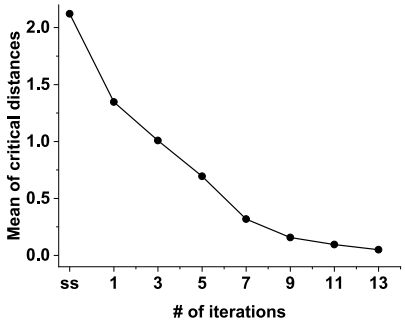


(c) Mixture of 9 components

Fig. 8. Comparison between *PTFlash* and *Carnot* in terms of speed. *NN-PTFlash* is *PTFlash* accelerated by neural networks, as presented in Section 4.



(a) Successive substitution (b) Trust-region method



(c) Closeness to critical points

Fig. 9. Figures (a) and (b) show the convergence percentage and the elapsed time up to each iteration of phase split calculations of *PTFlash* on GPU. In Figure (c), on the x-axis, ss corresponds to the end of successive substitution and other integers are the number of iterations of the trust-region method.

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

	ss of phase split calculations	Stability analysis				Phase split calculations	
		vapor-like estimate		liquid-like estimate			
		ss	tr	ss	tr	ss	tr
# of samples	10^6	625 645	130 715	625 645	90 179	413 442	223 741
Convergence	37.44%[a]	79.11%	100%	85.59%	100%	45.88%	100%
Max number of iterations	3	9	18	9	16	9	13
Total time	0.4565 s	0.4136 s	0.3417 s	0.4044 s	0.2706 s	0.7412 s	0.5132 s
		1.3237 s[b]				1.2544 s	
ss: successive substitution				tr: trust-region method			

[a]37.44% is the percentage of samples for which any of ΔG, tpd_x and tpd_y is negative after 3 attempts of successive substitution, as described in Section 2.2.3.
[b]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.

Table 5
Performance profiler of *NN-PTFlash* on GPU (Fig. 7) for the mixture of 9 components at $n = 10^6$ in Fig. 8(c).

	Classifier	Stability analysis				Phase split calculations	
		vapor-like estimate		liquid-like estimate			
		ss	tr	ss	tr	ss	tr
# of samples	10^6	5818	1073	5818	1704	413 442	134 786
Convergence	99.42%[a]	81.56%	100%	70.71%	100%	67.40%	100%
Max number of iterations		9	13	9	12	9	13
Total time	0.0005 s	0.1365 s	0.128 s	0.0514 s	0.12 s	0.7043 s	0.3388 s
		0.34 s				1.0431 s	
ss: successive substitution				tr: trust-region method			

[a]99.42% includes 58.38% predicted as stable (i.e., $p > p_r$) and 41.04% predicted as unstable (i.e., $p < p_l$).

overall time is also greatly reduced, which is contributed to better initial K_i provided by the initializer.

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 1.88 s. However, the attempt to use the initializer alone fails because we found its outputs may reach unreasonably large values (e.g., $1.0e15$) 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 problem, 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 because remaining samples located in the single-phase region are fairly close to the boundary after filtering through the classifier, as shown in Fig. 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 computation 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 Figs. 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 flexibility. First, we can seamlessly incorporate neural networks into *PTFlash*. Second, any subroutine of *PTFlash* is fully differentiable through automatic differentiation, and we can also leverage the implicit function theorem for efficient differentiation, 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 distribution 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.

CRediT authorship contribution statement

Jingang Qu: Conceptualization, Methodology, Formal analysis, Investigation, Software, Visualization, Writing – original draft, Writing – review & editing. **Thibault Faney:** Methodology, Validation, Investigation, Writing – original draft, Writing – review & editing. **Jean-Charles de Hemptinne:** Software, Investigation, Resources, Writing – review & editing. **Soleiman Yousef:** Investigation, Writing – review & editing. **Patrick Gallinari:** Investigation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

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]:

$$P = \frac{RT}{V-b} - \frac{a\alpha}{V(V+b)} \tag{A.1}$$

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:

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

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

$$b = \sum_{i=1}^{N_c} c_i b_i \tag{A.2c}$$

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

$$b_i = \frac{0.08664 \cdot R\,T_{c,i}}{P_{c,i}} \tag{A.2e}$$

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

$$m_i = 0.480 + 1.574\,\omega_i - 0.176\,\omega_i^2 \tag{A.2g}$$

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 ω_i is the acentric factor. We reformulate Eq. (A.1) as a cubic equation in terms of the compressibility factor Z:

$$f_{srk}(Z) = Z^3 - Z^2 + \rho_1 Z - \rho_0 = 0 \tag{A.3}$$

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 φ_i are calculated as follows:

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

where $\boldsymbol{c}$ 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].

B.1. Trust-region method for stability analysis

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

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

The minimization is accomplished by iterating the following equations:

$$\boldsymbol{\beta}^{(k)} = 2\sqrt{\boldsymbol{W}^{(k)}} \tag{B.1a}$$

$$(\boldsymbol{H}^{(k)} + \eta^{(k)}\boldsymbol{I}) \cdot \Delta\boldsymbol{\beta} + \boldsymbol{g}^{(k)} = \boldsymbol{0} \quad s.t. \quad \|\Delta\boldsymbol{\beta}\| \le \Delta_{max}^{(k)} \tag{B.1b}$$

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} + \Delta\boldsymbol{\beta} \tag{B.1c}$$

$$\boldsymbol{W}^{(k+1)} = \left(\frac{\boldsymbol{\beta}^{(k+1)}}{2}\right)^2 \tag{B.1d}$$

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

$$g_i = \sqrt{W_i}(\ln W_i + \ln\varphi_i(\boldsymbol{W}) - \ln z_i - \ln\varphi_i(\boldsymbol{z})) \tag{B.2a}$$

$$H_{ij} = \sqrt{W_i W_j}\frac{\partial \ln\varphi_i}{\partial W_i} + \sigma_{ij}\left(1 + \frac{g_i}{\beta_i}\right) \quad \text{where } \sigma_{ij} = 1 \Leftrightarrow i = j \tag{B.2b}$$

In addition, η is the trust-region size used to guarantee the positive definiteness of $\boldsymbol{H} + \eta\boldsymbol{I}$ and to tailor the step size to meet $\|\Delta\boldsymbol{\beta}\| \le \Delta_{max}$, where Δ_{max} is adjusted during iterations depending on the match between the actual reduction $\delta_{tm} = tm^{(k+1)} - tm^{(k)}$ and the predicted reduction based on the quadratic approximation $\hat{\delta}_{tm} = \Delta\boldsymbol{\beta}^T\boldsymbol{g} + \frac{1}{2}\Delta\boldsymbol{\beta}^T\boldsymbol{H}\Delta\boldsymbol{\beta}$, using the following heuristic rules:

$$\Delta_{max}^{(k+1)} = \begin{cases} \dfrac{\Delta_{max}^{(k)}}{2}, & \text{if } \left|\delta_{tm}/\hat{\delta}_{tm}\right| \le 0.25 \\ 2\Delta_{max}^{(k)}, & \text{if } \left|\delta_{tm}/\hat{\delta}_{tm}\right| \ge 0.75 \\ \Delta_{max}^{(k)}, & \text{otherwise} \end{cases} \tag{B.3}$$

The convergence criterion of Eq. (B.1) is $\max(|\boldsymbol{g}|) < 1.0e{-}6$.

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

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

$$G = \sum_{i=1}^{N_c} n_i^L(\ln x_i + \ln\varphi_i^L) + \sum_{i=1}^{N_c} n_i^V(\ln y_i + \ln\varphi_i^V)$$

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

$$\left(\tilde{\boldsymbol{H}}^{(k)} + \tilde{\eta}^{(k)} \cdot \boldsymbol{D}\left(\frac{\boldsymbol{z}}{\boldsymbol{xy}}\right)\right) \cdot \Delta\boldsymbol{n}^V + \tilde{\boldsymbol{g}}^{(k)} = \boldsymbol{0} \quad s.t. \quad \|\Delta\boldsymbol{n}^V\| \le \tilde{\Delta}_{max}^{(k)} \tag{B.4a}$$

$$\boldsymbol{n}^{V,k+1} = \boldsymbol{n}^{V,k} + \Delta\boldsymbol{n}^V \tag{B.4b}$$

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

$$\tilde{g}_i = \ln y_i + \ln\varphi_i^V - \ln x_i - \ln\varphi_i^L \tag{B.5a}$$

$$\tilde{H}_{ij} = \frac{1}{\theta_V(1-\theta_V)}\left(\frac{z_i}{x_i y_i}\sigma_{ij} - 1 + \theta_V\frac{\partial \ln\varphi_i^L}{\partial n_j^L} + (1-\theta_V)\frac{\partial \ln\varphi_i^V}{\partial n_j^V}\right) \tag{B.5b}$$

In addition, $\boldsymbol{D}(\cdot)$ is a diagonal matrix with diagonal entries in parentheses. The above iteration stops if $\max(|\tilde{\boldsymbol{g}}|) < 1.0e{-}8$. Here, the trust-region method is implemented in the same way as in stability analysis.

Appendix C. The Rachford-Rice equation

C.1. Solution of the Rachford-Rice equation

The Rachford–Rice equation is as follows:

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

Given $\boldsymbol{K}$, 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:

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

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 θ_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.

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

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

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

We rearrange the above equation and get Eq. (13), as follows:

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

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

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

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

See Table D.6

Table D.6
Some typical reservoir fluid compositions.

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

Appendix E. Vectorized algorithms

E.1. Synchronizer

See Algorithm 1.

E.2. Vectorized stability analysis

See Algorithm 2.

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

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

Initialization
  Set the number of iterations $k \leftarrow 1$
  Generate a vector $\boldsymbol{i}$ containing indices from 0 to $n-1$
  `/* n is the number of samples and indexing starts from 0. */`
  Create a placeholder matrix $\widetilde{\boldsymbol{X}}$ of the same shape as $\boldsymbol{X}^{(0)}$
while $k \leq K$ **do**
  $X^{(k+1)} \leftarrow f(X^{(k)}, \mathbb{O})$
  mask $\leftarrow C(\cdots)$
  `/* C returns a Boolean vector and True means convergence. */`
  Saving
    indices $\leftarrow \boldsymbol{i}$[mask]
    $\widetilde{\boldsymbol{X}}$[indices] $\leftarrow \boldsymbol{X}^{(k+1)}$[mask]
  Removing
    $\boldsymbol{i} \leftarrow \boldsymbol{i}$[$\sim$ mask]
    $\mathbb{O} \leftarrow \mathbb{O}$[$\sim$ mask]
    `/* Apply this operation to every element in 𝕆 */`
    $\boldsymbol{X}^{(k+1)} \leftarrow \boldsymbol{X}^{(k+1)}$[$\sim$ mask]
  $k \leftarrow k+1$
if len($\boldsymbol{i}$) $\neq 0$ **then**
  $\widetilde{\boldsymbol{X}}[\boldsymbol{i}] \leftarrow \boldsymbol{X}$
  `/* Also save unconverged results for further utilization. */`

Output: Converged results $\widetilde{\boldsymbol{X}}$ and unconverged indices $\boldsymbol{i}$

Algorithm 2: PyTorch pseudo-code of vectorized stability analysis

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

Initialization
  Instantiate *pteos* = *PTEOS*($\boldsymbol{P_c}$, $\boldsymbol{T_c}$, $\boldsymbol{\omega}$, BIPs)
  `/* PTEOS is a PyTorch-based class to efficiently calculate the fugacity coefficients and their partial derivatives. */`
Successive substitution
  Iterated function f_{ss} specified by Equation (5)
  Other inputs $\mathbb{O}_{ss} \leftarrow \{\boldsymbol{P}, \boldsymbol{T}, \mathbf{z}\}$
  $\boldsymbol{W}, \boldsymbol{i}_{ss} \leftarrow synchronizer(f_{ss}, \boldsymbol{W}^{(0)}, \mathbb{O}_{ss}, \mathbb{C}_{ss}, K_{ss})$
Trust-region method
  Iterated function f_{tr} specified by Equation (B.1)
  $\boldsymbol{W}_{tr}^{(0)} \leftarrow \boldsymbol{W}[\boldsymbol{i}_{ss}]$
  Other inputs $\mathbb{O}_{tr} \leftarrow \{\boldsymbol{P}[\boldsymbol{i}_{ss}], \boldsymbol{T}[\boldsymbol{i}_{ss}], \mathbf{z}[\boldsymbol{i}_{ss}]\}$
  $\boldsymbol{W}_{tr}, \boldsymbol{i}_{tr} \leftarrow synchronizer(f_{tr}, \boldsymbol{W}_{tr}^{(0)}, \mathbb{O}_{tr}, \mathbb{C}_{tr}, K_{tr})$
$\boldsymbol{W}[\boldsymbol{i}_{ss}] \leftarrow \boldsymbol{W}_{tr}$ and $\boldsymbol{i} \leftarrow \boldsymbol{i}_{ss}[\boldsymbol{i}_{tr}]$

Output: Converged results $\boldsymbol{W}$ and unconverged indices $\boldsymbol{i}$

References

[1] Michelsen ML. The isothermal flash problem. Part II. Phase-split calculation, Vol. 9. Elsevier; 1982, p. 21–40, (1).
[2] Michelsen ML. The isothermal flash problem. Part I. Stability, Vol. 9. Elsevier; 1982, p. 1–19, (1).
[3] Wang P, Stenby EH. Non-iterative flash calculation algorithm in compositional reservoir simulation, Vol. 95. Elsevier; 1994, p. 93–108.
[4] Belkadi A, Yan W, Michelsen ML, Stenby EH. Comparison of two methods for speeding up flash calculations in compositional simulations. In: SPE reservoir simulation symposium. OnePetro; 2011.
[5] Dogru AH, Fung LSK, Middya U, Al-Shaalan T, Pita JA. A next-generation parallel reservoir simulator for giant reservoirs. In: SPE reservoir simulation symposium. OnePetro; 2009.
[6] Michelsen ML. Simplified flash calculations for cubic equations of state, Vol. 25. ACS Publications; 1986, p. 184–8, (1).
[7] Hendriks EM. Reduction theorem for phase equilibrium problems, Vol. 27. ACS Publications; 1998, p. 1728–32, (9).
[8] Hendriks EM, Van Bergen A. Application of a reduction method to phase equilibria calculations, Vol. 74. Elsevier; 1992, p. 17–34.
[9] Rasmussen CP, Krejbjerg K, Michelsen ML, Bjurstrm KE. Increasing the computational speed of flash calculations with applications for compositional, transient simulations, Vol. 9. OnePetro; 2006, p. 32–8, (1).
[10] Voskov D, Tchelepi HA. Compositional space parameterization for flow simulation. In: SPE reservoir simulation symposium. OnePetro; 2007.
[11] Gaganis V, Varotsis N. Machine learning methods to speed up compositional reservoir simulation. In: SPE Europec/EAGE annual conference. OnePetro; 2012.
[12] Gaganis V, Varotsis N. An integrated approach for rapid phase behavior calculations in compositional modeling, Vol. 118. Elsevier; 2014, p. 74–87.
[13] Gaganis V. Rapid phase stability calculations in fluid flow simulation using simple discriminating functions. Comput Chem Eng 2018;108:112–27.
[14] Kashinath A, Szulczewski M, Dogru A. A fast algorithm for calculating isothermal phase behavior using machine learning, 465, 2018, 73–82. http://dx.doi.org/10.1016/j.fluid.2018.02.004.
[15] Wang S, Sobecki N, Ding D, Zhu L, Wu Y-S. Accelerating and stabilizing the vapor-liquid equilibrium (VLE) calculation in compositional simulation of unconventional reservoirs using deep learning based flash calculation, 253, 2019, p. 209–19. http://dx.doi.org/10.1016/j.fuel.2019.05.023.
[16] Zhang T, Li Y, Li Y, Sun S, Gao X. A self-adaptive deep learning algorithm for accelerating multi-component flash calculation. Comput Methods Appl Mech Engrg 2020;369:113207.

[17] Cortes C, Vapnik V. Support-vector networks, Vol. 20. Springer; 1995, p. 273–97, (3).
[18] Tipping ME. Sparse Bayesian learning and the relevance vector machine, 1, 2001, 211–44.
[19] Goodfellow I, Bengio Y, Courville A. Deep learning. MIT Press; 2016.
[20] Chen Z, Liu H, Yu S, Hsieh B, Shao L. GPU-based parallel reservoir simulators. In: Domain decomposition methods in science and engineering XXI. Springer; 2014, p. 199–206.
[21] Soave G. Equilibrium constants from a modified Redlich-Kwong equation of state, Vol. 27. Elsevier; 1972, p. 1197–203, (6).
[22] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, et al. Pytorch: An imperative style, high-performance deep learning library, 32, 2019, 8026–37.
[23] Lomont C. Introduction to intel advanced vector extensions, Vol. 23. 2011.
[24] Sanders J, Kandrot E. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional; 2010.
[25] Deiters UK, Macías-Salinas R. Calculation of densities from cubic equations of state: revisited. Ind Eng Chem Res 2014;53(6):2529–36.
[26] Zhi Y, Lee H. Fallibility of analytic roots of cubic equations of state in low temperature region, Vol. 201. Elsevier; 2002, p. 287–94, (2).
[27] Orbach O, Crowe C. Convergence promotion in the simulation of chemical processes with recycle-the dominant eigenvalue method, Vol. 49. Wiley Online Library; 1971, p. 509–13, (4).
[28] Hebden M. An algorithm for minimization using exact second derivatives. Citeseer; 1973.
[29] Rachford HH, Rice J. Procedure for use of electronic digital computers in calculating flash vaporization hydrocarbon equilibrium. Journal of Petroleum Technology 1952;4(10):19–3.
[30] Leibovici C, Neoschil J. A new look at the Rachford-Rice equation, 74, 1992, 303–308. http://dx.doi.org/10.1016/0378-3812(92)85069-K.
[31] Michelsen ML, Mollerup J. Thermodynamic modelling: fundamentals and computational aspects. Tie-Line Publications; 2004.
[32] McKay MD, Beckman RJ, Conover WJ. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code, Vol. 42. Taylor & Francis; 2000, p. 55–61, (1).
[33] Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q. Jax: composable transformations of python+numpy programs. 2018, URL http://github.com/google/jax.
[34] Van Rossum G, Drake FL. The python language reference manual. Network Theory Ltd.; 2011.
[35] Hendrycks D, Gimpel K. Gaussian error linear units (gelus). 2016.
[36] Elfwing S, Uchibe E, Doya K. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, Vol. 107. Elsevier; 2018, p. 3–11.
[37] Ramachandran P, Zoph B, Le QV. Swish: a self-gated activation function, Vol. 7. Technical report, 2017, p. 1.
[38] Bergstra J, Bardenet R, Bengio Y, Kégl B. Algorithms for hyper-parameter optimization, Vol. 24. 2011.
[39] Li L, Jamieson K, Rostamizadeh A, Gonina E, Ben-Tzur J, Hardt M, Recht B, Talwalkar A. A system for massively parallel hyperparameter tuning, 2, 2020, 230–46.
[40] Kingma DP, Ba J. Adam: a method for stochastic optimization. 2014.
[41] Smith LN. No more pesky learning rate guessing games, Vol. 5. 2015.
[42] Smith LN. Cyclical learning rates for training neural networks. In: 2017 IEEE winter conference on applications of computer vision (WACV). IEEE; 2017, p. 464–72.
[43] Smith LN, Topin N. Super-convergence: Very fast training of neural networks using large learning rates. In: Artificial intelligence and machine learning for multi-domain operations applications, Vol. 11006. International Society for Optics and Photonics; 2019, 1100612.
[44] Prechelt L. Early stopping-but when? In: Neural networks: tricks of the trade. Springer; 1998, p. 55–69.
[45] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern Recognition, 2016, p. 770–8.
[46] Cheng H-T, Koc L, Harmsen J, Shaked T, Chandra T, Aradhye H, Anderson G, Corrado G, Chai W, Ispir M, et al. Wide & deep learning for recommender systems. In: Proceedings of the 1st workshop on deep learning for recommender systems, 2016, p. 7–10.
[47] Krantz SG, Parks HR. The implicit function theorem: history, theory, and applications. Springer Science & Business Media; 2002.
[48] Michelsen ML, Yan W, Stenby EH. A comparative study of reduced-variables-based flash and conventional flash. SPE J 2013;18(05):952–9.
[49] Kontogeorgis GM, Folas GK. Thermodynamic models for industrial applications: from classical and advanced mixing rules to association theories. John Wiley & Sons; 2009.
[50] Wertheim MS. Fluids with highly directional attractive forces. II. Thermodynamic perturbation theory and integral equations, Vol. 35. Springer; 1984, p. 35–47, (1).
[51] Wertheim M. Fluids with highly directional attractive forces. I. Statistical thermodynamics, Vol. 35. Springer; 1984, p. 19–34, (1).
[52] Wertheim M. Fluids with highly directional attractive forces. IV. Equilibrium polymerization, Vol. 42. Springer; 1986, p. 477–92, (3).
[53] Wertheim M. Fluids with highly directional attractive forces. III. Multiple attraction sites, Vol. 42. Springer; 1986, p. 459–76, (3).
[54] Chapman WG, Gubbins KE, Jackson G, Radosz M. New reference equation of state for associating liquids, Vol. 29. ACS Publications; 1990, p. 1709–21, (8).
[55] Huang SH, Radosz M. Equation of state for small, large, polydisperse, and associating molecules, Vol. 29. ACS Publications; 1990, p. 2284–94, (11).
[56] Kontogeorgis GM, Voutsas EC, Yakoumis IV, Tassios DP. An equation of state for associating fluids, Vol. 35. ACS Publications; 1996, p. 4310–8, (11).
[57] Kontogeorgis GM, Yakoumis IV, Meijer H, Hendriks E, Moorwood T. Multicomponent phase equilibrium calculations for water–methanol–alkane mixtures, Vol. 158. Elsevier; 1999, p. 201–9.