Journal of Machine Learning Research 27 (2026) 1-50 Submitted 6/25; Revised 7/25; Published 1/26

STDE++: Polynomial-Time Amortization for Linear Differential Operators

Zekun Shi[†,‡] zekun.shi@comp.nus.edu.sg
Zheyuan Hu[†] e0792494@u.nus.edu
Min Lin[‡] linmin@sea.com
Kenji Kawaguchi[†] kenji@nus.edu.sg

[†] *School of Computing, National University of Singapore, 13 Computing Drive, Singapore 117417*
[‡] *Sea AI Lab, 1 Fusionopolis Place, Singapore 138522*

Editor: Kilian Weinberger

Abstract

Optimizing neural networks with losses that contain high-dimensional and high-order differential operators is expensive to evaluate with backpropagation due to $\mathcal{O}(d^k)$ scaling of the derivative tensor size and the $\mathcal{O}(2^{k-1}L)$ scaling in the computation graph, where d is the domain dimension, L is the number of ops in the forward computation graph and k is the derivative order. Previous works addressed the polynomial scaling in d by amortizing the computation over the optimization process via randomization. Separately, the exponential scaling in k for univariate functions ($d = 1$) was addressed with high-order auto-differentiation (AD). In this work, we show how to efficiently perform arbitrary contractions of the derivative tensor of arbitrary order for multivariate functions by properly constructing the input tangents to univariate high-order AD, which can be used to randomize any differential operator efficiently. When applied to Physics-Informed Neural Networks (PINNs) and compared against the original PyTorch implementation of SDGD, our method yields about 1.34×10^3 average speedup and $31.8\times$ average memory reduction across the three inseparable 100K-dimensional PDEs in our benchmark; the best case is 1.59×10^3 speedup and $33.8\times$ memory reduction on Allen-Cahn. We can now solve *1-million-dimensional PDEs in 8 minutes on a single NVIDIA A100 GPU.*[1] Furthermore, we proposed new methods for computing mixed partial derivatives using Taylor mode AD, which scales polynomially with the derivative order. This work opens the possibility of using high-order differential operators in large-scale problems.

Keywords: automatic differentiation, randomized numerical linear algebra, partial differential equations, high-order derivatives, physics-informed neural networks

1. Introduction

In many problems, especially in Physics-informed machine learning (Karniadakis et al., 2021; Raissi et al., 2019), one needs to solve optimization problems where the loss contains

1. Our code is available at https://github.com/sail-sg/stde
. The conference version of this paper received the Best Paper Award at NeurIPS 2024.

©2026 Zekun Shi, Zheyuan Hu, Min Lin, and Kenji Kawaguchi.

License: CC-BY 4.0, see https://creativecommons.org/licenses/by/4.0/. Attribution requirements are provided at http://jmlr.org/papers/v27/25-1474.html.

differential operators:

$$\arg\min_{\theta} f(\mathbf{x}, u_\theta(\mathbf{x}), \mathcal{D}^{\boldsymbol{\alpha}^{(1)}} u_\theta(\mathbf{x}), \dots, \mathcal{D}^{\boldsymbol{\alpha}^{(n)}} u_\theta(\mathbf{x})), \quad u_\theta : \mathbb{R}^d \to \mathbb{R}^{d'}. \tag{1}$$

In the above, $\mathcal{D}^{\boldsymbol{\alpha}} = \frac{\partial^{|\boldsymbol{\alpha}|}}{\partial x_1^{\alpha_1} \dots \partial x_d^{\alpha_d}}$, $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \dots, \alpha_d)$ is a multi-index, u_θ is some neural network parameterized by θ, and f is some cost function. When either the differentiation order k or the dimensionality d is high, the objective function above is expensive to evaluate with back-propagation (backward mode AD) in both memory and computation: the size of the derivative tensor has scaling $\mathcal{O}(d^k)$, and the size of the computation graph has scaling $\mathcal{O}(2^{k-1}L)$ where L is the number of ops in the forward computation graph.

There have been several efforts to tackle this curse of dimensionality. One line of work uses randomization to amortize the cost of computing differential operators with AD over the optimization process so that the d in the above scaling becomes a constant for the case of $k = 2$. Stochastic Dimension Gradient Descent (SDGD) (Hu et al., 2024b) randomizes over the input dimensions, where in each iteration, the partial derivatives are only calculated for a minibatch of sampled dimensions with back-propagation. In (Hu et al., 2024a; Lai et al., 2022; Hu et al., 2024c), the classical technique of Hutchinson Trace Estimator (HTE) (Hutchinson, 1989) is used to estimate the trace of the Hessian or Jacobian of inputs. Others choose to bypass AD completely to reduce the complexity of computation. In (Pang et al., 2020), the finite difference method is used for estimating the Hessian trace. Randomized smoothing (He et al., 2023; Hu et al., 2023) uses the expectation over Gaussian random variables as an ansatz, so that its derivatives can be expressed as another expectation of a Gaussian random variable via Stein's identity (Stein, 1981). However, compared to AD, the accuracy of these methods is highly dependent on the choice of discretization.

In this work, we address the scaling issue in both d and k for the optimization problem in Eq. 1 at the same time, by proposing an amortization scheme that can be efficiently evaluated via high-order AD, which we call *Stochastic Taylor Derivative Estimator (STDE)*. Our **main contributions** are:

- We demonstrate how Taylor mode AD (Bettencourt et al., 2019), a high-order AD method, can be used to amortize the optimization problem in Eq. 1. Specifically, we show that, with properly constructed input tangents, the univariate Taylor mode can be used to contract the multivariate function's derivative tensor of arbitrary order;

- We provide a comprehensive procedure for randomizing arbitrary differential operators with STDE, while previous works mainly focus on the Laplacian operator, and we provide abundant examples of STDE constructed for operators in common PDEs;

- One key tool used in STDE is to compute arbitrary mixed partial derivatives with Taylor mode AD. We proposed an algorithm that scales polynomially in derivative order, which improves upon the previous best approach (Griewank et al., 2000), which has exponential complexity;

- STDE encompass and generalizes previous methods like SDGD (Hu et al., 2024b) and HTE (Hutchinson, 1989; Hu et al., 2024a). We also prove that the HTE-type estimator cannot be generalized beyond second-order differential operators.

- We determine the efficacy of the STDE experimentally. When applied to PINN, our method provides a significant speed-up compared to the baseline method SDGD (Hu et al., 2024b) and the backward-free method like random smoothing (He et al., 2023). Due to STDE's low memory requirements and reduced computation complexity, PINNs with STDE can **solve 1-million-dimensional PDEs on a single NVIDIA A100 40GB GPU within 8 minutes**, which shows that PINNs have the potential to solve complex real-world problems that can be modeled as high-dimensional PDEs. We also provide a detailed ablation study on the source of performance gain of our method.

2. Related Works

High-Order and Forward-Mode AD The idea of generalizing forward mode AD to high-order derivatives has existed in the AD community for a long time (Bendtsen and Stauning, 1997; Karczmarczuk, 1998; Wang, 2017; Laurel et al., 2022). However, an accessible implementation for machine learning was not available until the recent implementation in JAX (Bettencourt et al., 2019; Bradbury et al., 2018), which implemented the Taylor mode AD for accelerating ODE solvers. There are also efforts in creating the forward rule for a specific operator like the Laplacian (Li et al., 2023, 2024). Randomization over the linearized part of the AD computation graph was considered in (Oktay et al., 2021). Forward mode AD can also be used to compute neural network parameter gradient as shown in (Baydin et al., 2022).

Randomized Gradient Estimation Randomization (Martinsson and Tropp, 2021; Murray et al., 2023; Ghojogh et al., 2021) is a common technique for tackling the curse of dimensionality for numerical linear algebra computation, which can be applied naturally in amortized optimization (Amos, 2023). Hutchinson trace estimator (Hutchinson, 1989) is a well-known technique, which has been applied to diffusion model (Song et al., 2019) and PINNs (Hu et al., 2024a). Another case that requires gradient estimation is when the analytical form of the target function is not available (black box), which means AD cannot be applied. The method of zeroth-order optimization (Liu et al., 2020) can be used in this case, as it only requires evaluating the function at an arbitrary input. It is also useful when the function is very complicated, like in the case of a large language model (Malladi et al., 2024).

3. Preliminaries and Discussions

In this section, we will go over the inefficiency of repeated first-order AD and previous works on randomizing differential operators.

3.1 Inefficiency of First-Order AD for High-Order Input Derivatives

High-order input derivatives $\frac{\partial^k u_\theta}{\partial \mathbf{x}^k}$ for scalar $u_\theta : \mathbb{R}^d \to \mathbb{R}$ can be implemented as repeated applications of first-order AD, e.g. `jax.grad(jax.grad(u))` in JAX. However, this approach will exhibit fundamental inefficiency that cannot be remedied by randomization. In the following analysis, we will assume a linear computation graph of length L where all primitives have input and output dimensions of h. For a brief recap on AD, see Appendix B.

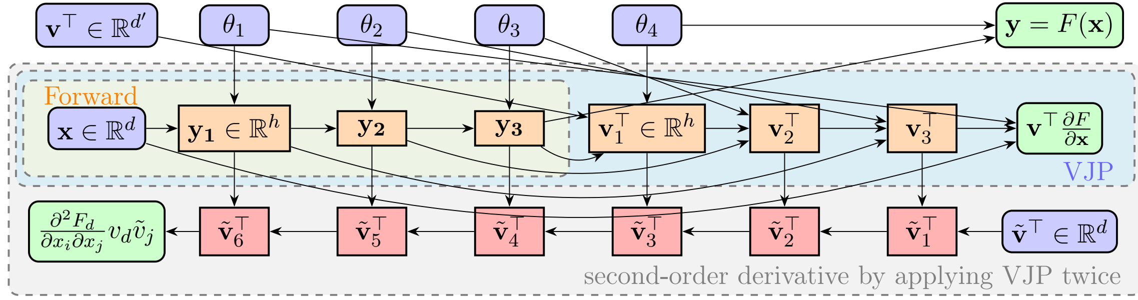

Figure 1: The computation graph of computing second order gradient by repeated application of backward mode AD, for a function $F(\cdot)$ with 4 primitives ($L = 4$), which computes the Hessian-vector-product. Red nodes represent the cotangent nodes in the second backward pass. With each repeated application of VJP the length of sequential computation doubles.

Asymptotics of First-Order AD Each Jacobian-vector-product (JVP) call requires $\mathcal{O}(\max(d, h))$ memory as only the current activation $\mathbf{y}_i$ and tangent $\mathbf{v}_i$ are needed to carry out the computation. Both the forward and the linearized graph have a computational complexity of $\mathcal{O}\big(dh + (L-1)h^2\big)$. For vector-Jacobian-product (VJP), the memory requirement becomes $\mathcal{O}(d + (L-1)h)$ as we need to store the entire evaluation trace.

Repeating Backward-Mode AD With each repeated application of backward mode AD, the new evaluation trace will include the cotangents from the previous application of backward AD, so the length of sequential computation **doubles**. Furthermore, the size of the cotangent also grows by d times. Therefore applying backward mode AD has additional memory cost of $\mathcal{O}(d + (L-1)h)$ and additional computation cost of $\mathcal{O}\big(2dh + 2(L-1)h^2\big)$, which is clear from Fig. 1. In general, with k repeated applications of backward mode AD will incur $\mathcal{O}\big(2^{k-1}(d + (L-1)h)\big)$ memory cost and $\mathcal{O}\big(2^k(dh + (L-1)h^2)\big)$ computation cost. And $\mathcal{O}\big(d^{k-1}\big)$ calls are needed to evaluate the entire derivative tensor. So both memory and compute scale **exponentially** in derivative order k

Repeating Forward-Mode AD Consider $u_\theta : \mathbb{R}^d \to \mathbb{R}$. The input tangent dimension is d on the first application of forward mode AD, but on the second application, it will become $d \times d$ since we are now computing the forward mode AD for $\nabla u_\theta : \mathbb{R}^d \to \mathbb{R}^d$. So the size of the input tangent with k repeated application is $\mathcal{O}\big(d^k\big)$, so it grows **exponentially**. This is also inefficient.

Mixed-Mode AD Schemes Are Also Likely Inefficient See Appendix C.

3.2 Stochastic Dimension Gradient Descent

SDGD (Hu et al., 2024b) amortizes high-dimensional differential operators by computing only a minibatch of derivatives in each iteration. It replaces a differential operator $\mathcal{D}$ with a randomly sampled subset of additive terms, where each term only depends on a few input

dimensions

$$\mathcal{D} := \sum_{j=1}^{N_{\mathcal{D}}} \mathcal{D}_j \approx \frac{N_{\mathcal{D}}}{|J|} \sum_{j \in J} \mathcal{D}_j := \tilde{\mathcal{D}}_J, \tag{2}$$

where $\tilde{\mathcal{D}}_J$ denotes the SDGD operator that approximates the true operator $\mathcal{D}$, J is the sampled index set, and $|J|$ is the batch size. For example, in d-dimensional Poisson equation, $N_{\mathcal{D}} = d$, $\mathcal{D} = \sum_{j=1}^{d} \frac{\partial^2}{\partial x_j^2}$, and the additive terms are $\mathcal{D}_j = \frac{\partial^2}{\partial x_j^2}$. $\tilde{\mathcal{D}}_J$ are cheaper to compute than $\mathcal{D}$ due to reduced dimensionality: for each sampled index, by treating all other inputs as constant, we get a function with scalar input and output. For a given index set J, the memory requirements are reduced from $\mathcal{O}\big(2^{k-1}(d + (L-1)h)\big)$ to $\mathcal{O}\big(|J|(2^{k-1}(1 + (L-1)h))\big)$, and the computation complexity reduces to $\mathcal{O}\big(|J|2^k(h + (L-1)h^2)\big)$. This reduction is significant when $d \gg h$ as in the experimental setting of SDGD (Hu et al., 2024b), but the exponential scaling in k persists.

4. STDE I: Generalizing the Hutchinson Trace Estimator

From the previous discussion, it is clear that the exponential scaling in k for the problem described in Eq. 1 cannot be mitigated by amortization alone. In the following sections, we describe a method that addresses the scaling issue in k and d simultaneously when amortizing Eq. 1 by seeing univariate Taylor mode AD as contractions of multivariate derivative tensors, which we call **Stochastic Taylor Derivative Estimator (STDE)**. Throughout this paper, we will mostly consider differential operators applied to scalar functions $u : \mathbb{R}^d \to \mathbb{R}$, but all the construction generalizes directly to vector-valued functions.

4.1 Regularity Assumptions

Throughout this paper, if the highest derivative order is k, we assume that the relevant differential operators act on functions u_θ that are classically C^k with respect to the input. In neural-network parameterizations, this means using smooth activations such as `tanh`, `softplus`, or `SiLU`, rather than non-smooth activations such as ReLU or LeakyReLU. Although JAX can propagate derivatives through non-smooth primitives by applying implementation-defined AD rules at kink points, we do not interpret those rules as weak derivatives or generalized gradients. Under the C^k assumption, the mixed partial derivatives used throughout the paper commute by Clairaut's theorem.

4.2 Estimating an Arbitrary Linear Differential Operator

Recall the well-known method for trace estimation, the Hutchinson trace estimator (HTE) (Hutchinson, 1989): given an isotropic $p(\mathbf{s})$, i.e. $\mathbb{E}_{\mathbf{s}\sim p(\mathbf{s})}[\mathbf{s}\mathbf{s}^\top] = I$, the trace of a matrix $\mathbf{A}$ can be approximated as follows

$$\operatorname{tr}(\mathbf{A}) \approx \frac{1}{M} \sum_{\mathbf{s}\sim p} \mathbf{s}^\top \mathbf{A}\mathbf{s}, \quad \mathbf{s} \in \mathbb{R}^d, \tag{3}$$

where M is the Monte Carlo sample size, since

$$\operatorname{tr}(\mathbf{A}) = \mathbf{A} \cdot \mathbf{I} = \mathbf{A} \cdot \mathbb{E}_{\mathbf{s}\sim p(\mathbf{s})}\left[\mathbf{s}\mathbf{s}^\top\right] = \mathbb{E}_{\mathbf{s}\sim p(\mathbf{s})}\left[\mathbf{s}^\top \mathbf{A}\mathbf{s}\right], \tag{4}$$

From the above short proof we see that the construction of HTE can be understood as (1) convert the trace function to a tensor contraction $\mathbf{A} \cdot \mathbf{I}$; (2) construct a low-rank random estimator $\mathbf{s}\mathbf{s}^\top$ with the expectation of $\mathbf{I}$.

We show that the same idea can be generalized for deriving stochastic estimators for an arbitrary linear differential operator. A linear differential operator $\mathcal{L}$ can be written as a linear combination of derivatives: $\mathcal{L} = \sum_{\boldsymbol{\alpha} \in \mathcal{I}(\mathcal{L})} C_{\boldsymbol{\alpha}} \mathcal{D}^{\boldsymbol{\alpha}}$, where $\mathcal{I}(\mathcal{L})$ is the set of multi-indices representing terms included in the operator $\mathcal{L}$, and the action of the derivative $\mathcal{D}^{\boldsymbol{\alpha}}$ on u at a point $\mathbf{a}$ is an element of the derivative tensor $D_u^{|\boldsymbol{\alpha}|}(\mathbf{a})_{\boldsymbol{\alpha}} \in \mathbb{R}^{d^k}$. For simplicity we only consider homogeneous differential operator, i.e. $|\boldsymbol{\alpha}| = k \in \mathbb{N}$ for all $\boldsymbol{\alpha}$. Then the coefficients $C_{\boldsymbol{\alpha}}$ forms a rank-k symmetric tensor , since a multi-index $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$ can be mapped to a tensor index $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_k)$ as follows

$$C_{\boldsymbol{\gamma}} = \begin{cases} C_{\boldsymbol{\alpha}} / \binom{k}{\boldsymbol{\alpha}}, & \alpha_i = \sum_{j=1}^{k} \delta_{i,\gamma_j}, \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

where the multinomial coefficient $\binom{k}{\boldsymbol{\alpha}} = \binom{k}{\alpha_1, \ldots, \alpha_k} = \frac{k!}{\alpha_1! \ldots \alpha_k!}$ removes double counting. We call $C_{\boldsymbol{\gamma}}$ the coefficient tensor of $\mathcal{L}$. Now $\mathcal{L}$ can be written as the following tensor inner product

$$\mathcal{L}u(\mathbf{a}) = D_u^k(\mathbf{a}) \cdot \mathbf{C}. \tag{6}$$

For example, the coefficient tensor of the Laplacian ∇^2 is the d-dimensional identity matrix $\mathbf{I}$, and the coefficient tensor of $\frac{\partial^2}{\partial x_1 \partial x_2}$ is $(\delta_{ij,12} + \delta_{ij,21})/2$. Now, suppose we can construct a distribution $(\mathbf{s}_1, \ldots, \mathbf{s}_k) \sim p, \mathbf{s}_i \in \mathbb{R}^d$ that satisfies

$$\mathbb{E}_p \left[\mathbf{s}_1 \otimes \cdots \otimes \mathbf{s}_k \right] = \mathbf{C}, \tag{7}$$

then we can construct an unbiased stochastic estimator of $\mathcal{L}u(\mathbf{a})$ for arbitrary $\mathcal{L}$:

$$\mathcal{L}u(\mathbf{a}) = D_u^k(\mathbf{a}) \cdot \mathbb{E}_p \left[\mathbf{s}_1 \otimes \cdots \otimes \mathbf{s}_k \right] \approx \frac{1}{M} \sum_{(\mathbf{s}_1, \ldots, \mathbf{s}_k) \sim p} D_u^k(\mathbf{a}) \cdot \mathbf{s}_1 \ldots \mathbf{s}_k, \tag{8}$$

where each sample computes a tensor inner product between the derivative tensor $D_u^k(\mathbf{a})$ and a low-rank projection tensor $\mathbf{s}_1 \ldots \mathbf{s}_k$. Note that, since the Kronecker product $\otimes$ commutes in the above tensor contraction due to the symmetry of $D_u^k(\mathbf{a})$, dropping the symbol $\otimes$ in such a contraction, as done above, causes no ambiguity.

In the following sections, we will discuss how to construct the distribution p that satisfies Eq. 7.

4.3 Sparse STDE

All coefficient tensors can be additively decomposed into one-hot tensors $\mathbf{e}_{\gamma_1} \otimes \cdots \otimes \mathbf{e}_{\gamma_k}$ where $\mathbf{e}_i$ is the ith standard basis:

$$\mathbf{C} = \sum_{\boldsymbol{\gamma}} S_{\boldsymbol{\gamma}} |C_{\boldsymbol{\gamma}}| \mathbf{e}_{\gamma_1} \otimes \cdots \otimes \mathbf{e}_{\gamma_k}, \tag{9}$$

where $S_{\boldsymbol{\gamma}}$ is the sign of $C_{\boldsymbol{\gamma}}$. Therefore, the following sampling scheme with discrete distribution p supported on the index set $\mathcal{I}(\mathcal{L})$ forms an unbiased estimator of $\mathcal{L}$:

$$\mathcal{L}u(\mathbf{a}) = Z \mathbb{E}_p [D_u^k(\mathbf{a})_{\boldsymbol{\gamma}} S_{\boldsymbol{\gamma}}] \approx \frac{Z}{M} \sum_{\boldsymbol{\gamma} \sim p} D_u^k(\mathbf{a})_{\boldsymbol{\gamma}} S_{\boldsymbol{\gamma}}, \quad p(\boldsymbol{\gamma}) = \frac{|C_{\boldsymbol{\gamma}}|}{Z}, \quad Z = \sum_{\boldsymbol{\gamma} \in \mathcal{I}(\mathcal{L})} |C_{\boldsymbol{\gamma}}|. \tag{10}$$

For clarity, we write the kth order derivative tensor $D_u^k(\mathbf{a})$ as simply D in this section from now on. Denoting the sign of $D_{\boldsymbol{\gamma}} S_{\boldsymbol{\gamma}}$ as $s_{\boldsymbol{\gamma}}$, its mean as $\overline{s} = \mathbb{E}_p[s_{\boldsymbol{\gamma}}]$, and the covariance between $|D_{\boldsymbol{\gamma}}|$ and $s_{\boldsymbol{\gamma}}$ as c, we have

$$\mathbb{E}_p[D_{\boldsymbol{\gamma}} S_{\boldsymbol{\gamma}}] = \mathbb{E}_p\left[|D_{\boldsymbol{\gamma}}| s_{\boldsymbol{\gamma}}\right] = \overline{s}\mathbb{E}_p\left[|D_{\boldsymbol{\gamma}}|\right] + c. \tag{11}$$

Thus, the variance of the estimator is given by

$$\begin{aligned} \frac{Z^2}{M}\mathbb{V}_p[D_{\boldsymbol{\gamma}} S_{\boldsymbol{\gamma}}] =& \frac{Z^2}{M}\left[\mathbb{E}_p\left[|D_{\boldsymbol{\gamma}}|^2\right] - \left(\overline{s}\mathbb{E}_p\left[|D_{\boldsymbol{\gamma}}|\right] - c\right)^2\right] \\ =& \frac{(\mathcal{L}u(\mathbf{a}))^2}{M}\left[\frac{\mathbb{E}_p\left[|D_{\boldsymbol{\gamma}}|^2\right]}{\left(\overline{s}\mathbb{E}_p\left[|D_{\boldsymbol{\gamma}}|\right] - c\right)^2} - 1\right] \propto (M\overline{s})^{-2}. \end{aligned} \tag{12}$$

We see that, besides the structure of D, the average sign $\overline{s}$ determines the variance of the sparse STDE estimator. As $\overline{s} \to 0$, the variance blows up, same as the sign problem in quantum Monte Carlo (Troyer and Wiese, 2004). However, note that in our case the variance comes purely from the structure of the operator $\mathcal{L}$, since the average sign does not scale with the size of the system, so we do not have a scaling issue in terms of dimensionality.

4.4 Dense STDE

It is also possible to construct p with continuous support for arbitrary second-order differential operators. Suppose $\mathcal{L}$ is a second-order differential operator with coefficient tensor $\mathbf{C}$. If $\mathbf{C}$ is not positive definite, we add a constant diagonal $\lambda\mathbf{I}$ where $-\lambda$ is smaller than the smallest eigenvalue of $\mathbf{C}$. The matrix $\mathbf{C}' = \mathbf{C} + \lambda\mathbf{I}$ then has the eigendecomposition $\mathbf{U}\boldsymbol{\Sigma}\mathbf{U}^\top$ where $\boldsymbol{\Sigma}$ is diagonal and all positive. Now we have

$$\mathbb{E}_{\mathbf{s}\sim\mathcal{N}(\mathbf{0},\boldsymbol{\Sigma})}[\mathbf{U}\mathbf{s}(\mathbf{U}\mathbf{s})^\top] - \lambda\mathbb{E}_{\mathbf{s}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[\mathbf{s}\mathbf{s}^\top] = \mathbf{U}\boldsymbol{\Sigma}\mathbf{U}^\top - \lambda\mathbf{I} = \mathbf{C}. \tag{13}$$

The construction above directly generalizes HTE: When $\mathcal{L} = \nabla^2$, $\mathbf{C} = \mathbf{I}$ and $\mathbf{U} = \mathbf{I}$. We call the STDE construct with continuous support "dense" as opposed to sparse STDE.

However, it is not always possible to construct dense STDE beyond the second order, even if we consider p with nondiagonal covariance. We prove this by providing a counterexample: one cannot construct a dense STDE for the fourth-order operator $\sum_{i=1}^d \frac{\partial^4}{\partial x^4}$ (see Appendix K). For specific high-order operators like the Biharmonic operator, it is still possible to construct STDE with dense jets, which we show in Appendix J.

4.5 Discussion

The main differences between the sparse and the dense versions of STDE are:

1. sparse STDE is universally applicable, whereas the dense STDE can only be applied to certain operators;
2. the source of variance is different (see Appendix L).

It is also worth noting that both the sparse and the dense versions of STDE would have similar computation costs if the Monte Carlo sample size were the same. In general, we would suggest using sparse STDE unless it is known a priori that the sparse version would suffer from excess variance, and the dense STDE is applicable.

5. STDE II: Derivative Tensor Contraction via Taylor-Mode AD

In the last section, we showed that STDE is general, as it can be applied to arbitrary linear differential operators of any order. However, the efficiency of STDE depends on whether we can compute the tensor contraction $D_u^k(\mathbf{a}) \cdot \mathbf{s}_1 \dots \mathbf{s}_k$ in Eq. 8 efficiently. This section shows that such contraction naturally occurs in high-order directional derivatives, which can be computed efficiently using Taylor-mode AD.

5.1 Multiset

First, we will set up the notation that we will use to denote various properties of a multiset, which is used throughout this paper. A multiset $\mathbf{p}$ is a set with repeated elements, e.g., $\{2, 2, 3\}$ is a multiset. When its universe is ordered (e.g. $\mathbb{N}$), it is also a multi-index $\mathbf{p} = (p_1, \dots, p_k)$ where p_i is the multiplicity of the i-th element in the universe. Note that in this paper, we only consider multisets over the universe $\{0, \dots, k\}$, so they are also multi-indices. We define its support as the set of elements with nonzero multiplicity: $\operatorname{supp}(\mathbf{p}) = \{i : p_i \neq 0\}$. We further define two norms: $|\mathbf{p}| = \sum_{i=1}^k p_i$ denotes the sum of multiplicities of $\mathbf{p}$, and $|\mathbf{p}|_{\text{mul}} = \sum_{a \in \mathbf{p}} a = \sum_{i=1}^k i p_i$ is the sum of all elements in $\mathbf{p}$, which we call the multiset norm of $\mathbf{p}$. We also call $\mathbf{p}$ a multiset partition of $|\mathbf{p}|$. A simple multiset $\mathbf{p}$ is an ordinary set, i.e. $\mathbf{p}$ where $p_i \in \{0, 1\}$.

5.2 High-Order Directional Derivatives Contract the Derivative Tensor

Directional derivative ∂u computes the rate of change of a function u along the line $\mathbf{x}(t) = \mathbf{x}^{(0)} + \mathbf{x}^{(1)} t$, where $\mathbf{x}^{(0)} \in \mathbb{R}^d$ is the primal and $\mathbf{x}^{(1)} \in \mathbb{R}^d$ is the tangent:

$$\partial u(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}) = \frac{\mathrm{d}}{\mathrm{d}t}[u \circ \mathbf{x}]\bigg|_{t=0} = \frac{\partial u}{\partial \mathbf{x}}\bigg|_{\mathbf{x}=\mathbf{x}^{(0)}} \mathbf{x}^{(1)}, \tag{14}$$

This is the JVP of u, which the first-order forward-mode AD computes. It can be generalized to higher order by considering the kth order rate of change, along a smooth 1D curve $\mathbf{x}(t)$ which has the Taylor expansion $\mathbf{x}(t) = \mathbf{x}^{(0)} + \mathbf{x}^{(1)} t + \frac{1}{2!}\mathbf{x}^{(2)} \dots + \frac{1}{k!}\mathbf{x}^{(k)} t^k + \mathcal{O}(t^{k+1})$:

$$\partial^k u(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(k)}) = \frac{\mathrm{d}^k}{\mathrm{d}t^k} u(\mathbf{x}(t))\bigg|_{t=0} = \sum_{|\mathbf{p}|_{\text{mul}}=k} \binom{k}{\mathbf{p}} D_u^{|\mathbf{p}|}(\mathbf{x}^{(0)}) \cdot \prod_{j=1}^k \left(\frac{1}{j!}\mathbf{x}^{(j)}\right)^{p_j} \tag{15}$$

where $\mathbf{p}$ is a multiset of $\{0, \dots, k\}$. The above explicit formula is known as the **Faa di Bruno's formula**, which is a high-order generalization of the chain rule (see proof sketch at Appendix D). From the formula, we see that high-order directional derivatives are a sum of derivative tensor inner product with the low-rank tensor formed by the Kronecker product of input tangents $\mathbf{x}^{(i)}$. For example, for $k = 2$ the multiset $\mathbf{p}$ that satisfies $|\mathbf{p}|_{\text{mul}} = 2$ are $(2, 0)$ and $(0, 1)$, and we have

$$\partial^2 u(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \frac{\partial^2}{\partial t^2}[u \circ \mathbf{x}](t) = D_u(\mathbf{x}^{(0)}) \cdot \mathbf{x}^{(2)} + D_u^2(\mathbf{x}^{(0)}) \cdot (\mathbf{x}^{(1)})^2. \tag{16}$$

Setting $\mathbf{x}^{(0)} = \mathbf{a}, \mathbf{x}^{(1)} = \mathbf{s}, \mathbf{x}^{(2)} = \mathbf{0}$, one can compute via $\partial^2 u$ the quadratic form of Hessian $D_u^2(\mathbf{x}) \cdot (\mathbf{s})^2$. We will show later that, for any multiset $\mathbf{I}$ of $\mathbb{N}$ where $\sum_{i=1}^k I_i = k$, one can

always find large enough $l \geq k$ and an assignment of $\mathbf{x}^{(i)}$ such that

$$\partial^l u(\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(l)}) \propto D_u^k(\mathbf{a}) \cdot \mathbf{s}_1^{I_i} \ldots \mathbf{s}_k^{I_k}. \tag{17}$$

5.3 Taylor-Mode AD

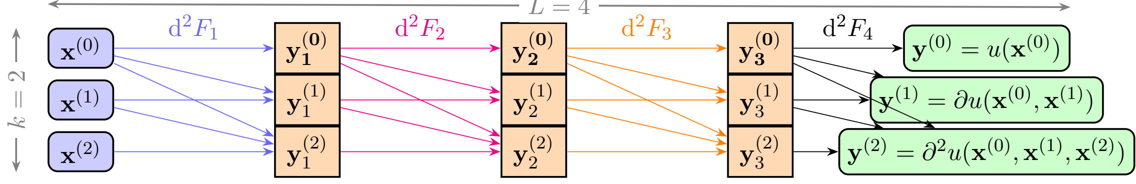

Figure 2: The computation graph of $\mathrm{d}^2 u$ for $u = F_4 \circ F_3 \circ F_2 \circ F_1$. Parameters θ_i of F_i are omitted. The first column from the left represents the input 2-jet $(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)})$, and $\mathrm{d}^2 F_1$ pushes it forward to the 2-jet $(\mathbf{y}_1^{(0)}, \mathbf{y}_1^{(1)}, \mathbf{y}_1^{(2)})$ which is the subsequent column. No evaluation trace needs to be cached, and the number of sequential computations stays linear in k.

High-order directional derivatives ∂^k can be computed efficiently using Taylor-mode AD. Any forward computation

$$u : \mathbb{R}^d \to \mathbb{R}^{d'}, \quad \mathbf{y}^{(0)} = u(\mathbf{x}^{(0)}) \tag{18}$$

can be extended to operate on 1D smooth curve $\mathbf{x}(t) : \mathbb{R} \to \mathbb{R}^{d'}$, since applying u on the whole curve $\mathbf{x}(t)$ is equivalent to perform function composition:

$$\mathbf{x}(0) = \mathbf{x}^{(0)}, \quad \mathbf{y}(0) = \mathbf{y}^{(0)}, \quad \mathbf{y}(t) = [u \circ \mathbf{x}](t). \tag{19}$$

The kth derivatives of $\mathbf{y}(t)$ is by definition (eq. 15), a high-order directional derivative with $\mathbf{x}^{(i)}$ as input tangents

$$\mathbf{y}^{(k)} = \frac{\mathrm{d}^k}{\mathrm{d}t^k} u(\mathbf{x}(t)) \bigg|_{t=0} := \partial^k u(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(k)}). \tag{20}$$

If we represent both input and output curves as kth-order truncated Taylor series, then the curves can be represented compactly as a tuple of the first k Taylor coefficients, also called k-**jet**:

$$J_{\mathbf{x}}^k := (\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(k)}), \quad J_{\mathbf{y}}^k := (\mathbf{y}^{(0)}, \ldots, \mathbf{y}^{(k)}), \quad J_{\mathbf{y}}^k = \mathrm{d}^k u(J_{\mathbf{x}}^k). \tag{21}$$

In the above, d^k represents the extension operator that extends a function to operate on jets. Geometrically, d^k is a high-order pushforward. d^k is an homomorphism of the group $(\{F_i\}, \circ)$ to the group $(\{\mathrm{d}^k F_i, \circ\})$:

$$\mathrm{d}^k[F_2 \circ F_1](J_{\mathbf{x}}^k) = J_{F_2 \circ F_1 \circ \mathbf{x}}^k = \mathrm{d}^k F_2(J_{F_1 \circ \mathbf{x}}^k) = [\mathrm{d}^k F_2 \circ \mathrm{d}^k F_1](J_{\mathbf{x}}^k). \tag{22}$$

That is, the d^k of an arbitrary composition of primitives F_i can be computed viaPolynomial the d^k of primitives, which is assumed to be known analytically. This forms a high-order

forward-mode AD scheme, since one can now compute the extension $\mathrm{d}^k F$ of arbitrary composition F by composing the extended primitives $\mathrm{d}^k F_i$, and the case of $k = 1$ corresponds to first-order forward mode AD. From the computation graph (Fig. 2), we see that the length of the sequential computation does not grow exponentially for computing higher-order derivatives.

The jet notation described above is what Taylor-mode AD in JAX (Bettencourt et al., 2019) uses. The `jet(f, primals, tangents)` function from `jax.experimental.jet` calls the function `f` extended by d^k with primals $\mathbf{x}^{(0)}$ and tangents $(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(k)})$.

6. STDE III: Complexity of Derivative Tensor Contractions

In the last section, we claim that any contraction of $D_u(k)$ with rank-1 tensors $\mathbf{s}_i$ can be formulated as a high-order directional derivative (Eq. 17), and hence can be computed efficiently via Taylor-mode AD. In this section, we provide new algorithms for constructing the appropriate input tangents $\mathbf{x}^{(i)}$, whose complexity scales **polynomially** in k in the worst case.

We call a differential operator *diagonal* if it is a linear combination of diagonal elements from the derivative tensor: $\mathcal{L} = \sum_{j=1}^{d} \frac{\partial^k}{\partial x_j^k}$, i.e. $\mathbf{I} = (k, 0, \dots, 0)$. In this section, we will focus on the hardest case where the contraction is maximally nondiagonal, i.e. $\mathbf{I} = (1, \dots, 1)$, which is sufficient to bound the worst-case complexity and to demonstrate the generality of our method.

6.1 Multiple Linear Jets Scheme

This algorithm is due to (Griewank et al., 2000). The idea is to use linear jets $(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{0}, \dots)$ which evaluates the diagonal contraction of D_u^k under pushforward ∂^k

$$\partial^k u(\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{0}, \dots) = D_u(\mathbf{x}^{(0)}) \cdot (\mathbf{x}^{(1)})^k. \tag{23}$$

with multinomial tangent $\mathbf{x}^{(1)} = \mathbf{s}_1 + \dots + \mathbf{s}_k$ to get the mixed term. For example, for the case of $k = 2$, we can contract the Hessian along two different directions, i.e. $D_u^2(\mathbf{a}) \cdot \mathbf{s}_1 \mathbf{s}_2$, by computing three ∂^2 with carefully chosen tangent:

$$\begin{aligned} D_u^2(\mathbf{a}) \cdot \mathbf{s}_1 \mathbf{s}_2 =& D_u^2(\mathbf{a}) \cdot \frac{1}{2}[(\mathbf{s}_1 + \mathbf{s}_2)^2 - \mathbf{s}_1^2 - \mathbf{s}_2^2] \\ =& \frac{1}{2}[\partial^2 u(\mathbf{a}, \mathbf{s}_1 + \mathbf{s}_2, \mathbf{0}) - \partial^2 u(\mathbf{a}, \mathbf{s}_1, \mathbf{0}) - \partial^2 u(\mathbf{a}, \mathbf{s}_2, \mathbf{0})]. \end{aligned} \tag{24}$$

In the above, the tangent $\mathbf{s}_1 + \mathbf{s}_2$ is used to generate the cross term $\mathbf{s}_1 \otimes \mathbf{s}_2$, and the square term $\mathbf{s}_i \otimes \mathbf{s}_i$ can be canceled out. To evaluate the maximally nondiagonal contraction, one needs to perform an exponential number of jet pushforwards to cancel out unwanted terms:

$$\mathbf{I} = (1, \dots, 1) \quad \mapsto \quad D_u^k(\mathbf{a}) \cdot \prod_{j=1}^{k} \mathbf{s}_j = \frac{1}{k!} \sum_{\mathbf{i} \leq \mathbf{I}} (-1)^{k - |\mathbf{i}|} \partial^k u(\mathbf{a}, \sum_{i_j \neq 0} \mathbf{s}_j, \mathbf{0}, \dots). \tag{25}$$

This construction can be proved using combinatorial arguments similar to the inclusion-exclusion principle. Starts with jets with multinomial tangent $\mathbf{x}^{(1)} = \sum_{j=1}^{k} \mathbf{s}_j$. The output

contains all kth-order monomials consisting of $\{\mathbf{s}_1, \dots, \mathbf{s}_k\}$, including the desired cross term $\mathbf{s}_1 \dots \mathbf{s}_k$. Subtracting the pushforward with the multinomial tangent associated with subsets $\mathbf{i}$ of $\mathbf{I}$ with cardinality $|\mathbf{i}| = k-1$ removes all unwanted terms. However, all pairwise intersections between subsets $\mathbf{i}$ are subtracted twice. The intersections are covered by subsets $\mathbf{i}$ with cardinality $|\mathbf{i}| = k-2$, so we add them back. Inductively applying this logic though $|\mathbf{i}| = k-1$ to $|\mathbf{i}| = 1$ yields the above formula, where the prefactor $\frac{1}{k!}$ comes from Faa di Bruno's formula (Eq. 15): $\binom{k}{(1,\dots,1)} = k!$.

Since the summation $\sum_{\mathbf{i}\leq\mathbf{I}}$ goes through all subsets of $\{1, \dots, k\}$, in total we need to perform $\mathcal{O}(2^k)$ pushforwards of k-jets. Since each k-jet pushforward has complexity $\mathcal{O}(k^2)$, the overall complexity is $\mathcal{O}(k^2 2^k)$, which is **exponential** in k.

6.2 Single-Jet Scheme

Here we present a new algorithm for evaluating maximally non-diagonal mixed partials that uses only one jet pushforward and scales **polynomially** in k.

6.2.1 Main Idea

One can remove all but one term in the summation of Faa di Bruno's formula (Eq. 15), by using a set of carefully constructed input tangents.

Recall that $\partial^l u$ is a sum of terms proportional to $D_u^{|\mathbf{p}|} \cdot \prod_{j=1}^{l} (\mathbf{x}^{(j)})^{p_j}$ in all $\mathbf{p}$ with $|\mathbf{p}|_{\mathrm{mul}} = l$. These $\mathbf{p}$ form a partially ordered set $\mathcal{P}_l = (\{\mathbf{p} : |\mathbf{p}|_{\mathrm{mul}} = l\}, \preceq)$ where the partial order $\preceq$ is defined based on the inclusion of the support: for $\mathbf{p}, \mathbf{p}' \in \mathcal{P}_l$,

$$\mathbf{p} \preceq \mathbf{p}' \quad \Longleftrightarrow \quad \operatorname{supp}(\mathbf{p}) \subseteq \operatorname{supp}(\mathbf{p}'). \tag{26}$$

For a given $\mathbf{p}$, we define the **mask function** m that masks out tangents from a jet based on the support of $\mathbf{p}$:

$$m(\mathbf{p}, J_{\mathbf{x}}^l) = (\mathbf{x}^{(0)}, \mathbf{x}^{(1)} \mathbf{1}_{p_1 \neq 0}, \dots, \mathbf{x}^{(l)} \mathbf{1}_{p_l \neq 0}). \tag{27}$$

For any $\mathbf{p}' \in \mathcal{P}_l$, applying the mask function leaves only terms with $\mathbf{p} \preceq \mathbf{p}'$, as for $\mathbf{p} \not\preceq \mathbf{p}'$ there will be a $\mathbf{x}^{(j)} = \mathbf{0}$ in the term associated with $\mathbf{p}$:

$$\partial^l u(m(\mathbf{p}', J_{\mathbf{x}}^l)) = \sum_{\mathbf{p}, \mathbf{p} \preceq \mathbf{p}'} C(l, \mathbf{p}) D_u^{|\mathbf{p}|}(\mathbf{x}^{(0)}) \cdot \prod_{j=1}^{l} (\mathbf{x}^{(j)})^{p_j}, \quad C(l, \mathbf{p}) = \binom{l}{\mathbf{p}} \prod_{j=1}^{l} \left(\frac{1}{j!}\right)^{p_j}. \tag{28}$$

Since $\mathcal{P}_l$ is finite, there is always at least one minimal element $\mathbf{p}^*$, i.e. $\{\mathbf{p}' : \mathbf{p}' \prec \mathbf{p}^*\} = \emptyset$. Suppose a minimal $\mathbf{p}^*$ is unique, i.e., $\mathbf{p}^*$ is the only element in its equivalence class. Applying the mask function with $\mathbf{p}^*$ leaves only one term in the summation. Here we give a concrete example: $\mathbf{p}^* = \{2, 3\}$ is an unique minimal in $\mathcal{P}_5$, and

$$\partial^5 u(\mathbf{x}^{(0)}, \mathbf{0}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{0}, \mathbf{0}) = 10 D_u^2(\mathbf{x}^{(0)}) \cdot \mathbf{s}_1 \mathbf{s}_2. \tag{29}$$

For $l \geq k$, $\partial^l u$ will contain some contraction of D_u^k when $|\mathbf{p}| = k$. Suppose one has an algorithm $\mathcal{A}$ that can find an unique minimal simple multiset partition of k, i.e. $\mathcal{A}(k) = \mathbf{p}^*$ where $|\mathbf{p}^*| = k$, $\mathbf{p}^*$ is minimal in $\mathcal{P}_{|\mathbf{p}^*|_{\mathrm{mul}}}$, then we are able to compute arbitrary kth order mixed partial with just a single jet. In the following sections, we propose concrete constructions of $\mathcal{A}$. Before that, we formalize the above discussion into the following lemma:

Lemma 1 (Single-Jet Contraction Under Unique Minimality) *Let $u : \mathbb{R}^d \to \mathbb{R}$ be C^ℓ in a neighborhood of $x^{(0)}$. Fix $\ell \in \mathbb{N}$ and suppose $\mathbf{p}^\star \in \mathcal{P}_l$ is a unique minimal element. Then for any l-jet $J^l_{\mathbf{x}} = (\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(l)})$ satisfying $\mathbf{x}^{(j)} = \mathbf{0}$ whenever $p^\star_j = 0$, we have*

$$\partial^l u\Big(m(p^\star, J^l_{\mathbf{x}})\Big) = C(l, \mathbf{p}^\star)\ D^{|\mathbf{p}^\star|} u(\mathbf{x}^{(0)}) \cdot \prod_{j=1}^{l} (\mathbf{x}^{(j)})^{p^\star_j}, \tag{30}$$

where $C(l, \mathbf{p}) = \binom{l}{\mathbf{p}} \prod_{j=1}^{l} \left(\frac{1}{j!}\right)^{p_j}$. In particular, if $\mathbf{p}^\star$ is simple with $\mathrm{supp}(\mathbf{p}^\star) = \{a_1, \dots, a_k\}$ *and we set*

$$\mathbf{x}^{(a_i)} = \mathbf{s}_i \in \mathbb{R}^d \quad (i = 1, \dots, k), \qquad \mathbf{x}^{(j)} = \mathbf{0}\ (j \notin \{a_i\}),$$

then $l = \sum_{i=1}^{k} a_i$ and

$$\partial^l u\Big(m(\mathbf{p}^\star, J^l_{\mathbf{x}})\Big) = \frac{l!}{\prod_{i=1}^{k} a_i!}\ D^k u(x^{(0)}) \cdot \mathbf{s}_1 \cdots \mathbf{s}_k. \tag{31}$$

6.2.2 Characterizing the Unique Minimal Multiset Partition of k

For a given $\mathbf{p} \in \mathcal{P}_l$ where $|\mathbf{p}| = k$, to determine whether there exists $\mathbf{q} \prec \mathbf{p}$, we need to solve the following linear Diophantine equation with variables x_i:

$$\sum_{i=1}^{|\mathrm{supp}(\mathbf{p})|} a_i x_i = |\mathbf{p}|_{\mathrm{mul}}, \quad \forall i, x_i \in \mathbb{N}, a_i \in \mathrm{supp}(\mathbf{p}). \tag{32}$$

$\mathbf{p}$ is minimal if for all solution $\mathbf{x}$, $\mathrm{supp}(\mathbf{x}) = \mathrm{supp}(\mathbf{p})$. It is a unique minimal if the only solution is $x_i = p_i$. This problem is equivalent to the well-known unbounded knapsack problem, which is **NP-hard**, and there is no known polynomial time algorithm that works for all a_i. However, since $\mathbf{p}$ is chosen by us, we can ask the question: how to choose a $\mathbf{p}$ such that the above question is easy to solve? In the next section, we show that there exists a special $\mathbf{p}$ which indeed makes the Diophantine equation easy to solve.

6.2.3 Exponential Algorithm

The following is a **deterministic** algorithm for constructing a unique minimal $\mathbf{p}$.

Theorem 2 *Given k, the following algorithm $\mathcal{A}$ gives a minimal multiset partition of k:*

$$\mathcal{A}_{exp}(k) = \{2^{k-1} + ... + 2^{k-i-1} = -2^{k-i-1}(1 - 2^{i+1}) = 2^k - 2^{k-i-1} : i = 0, \dots, k-1\} \tag{33}$$

where

$$l = |\mathbf{p}|_{mul} = \sum_{i=0}^{k-1} (2^k - 2^{k-i-1}) = k2^k - \frac{1 - 2^k}{1 - 2} = (k-1)2^k + 1. \tag{34}$$

Proof Let $b_j = 2^j$. The corresponding Diophantine equation is given by

$$\begin{aligned} \sum_{i=0}^{k-1} (2^k - 2^{k-i-1}) x_i = \left(\sum_{i=0}^{k-1} x_i\right) 2^k - \sum_{i=0}^{k-1} 2^{(k-i-1)} x_i &= (k-1)2^k + 1 \\ \sum_{j=0}^{k-1} b_j x_{k-j-1} &= \underbrace{\left(\sum_{i=0}^{k-1} x_i - (k-1)\right)}_{=:N} 2^k - 1. \end{aligned} \tag{35}$$

In binary representation, $b_j = 00\dots\underbrace{1}_{(j-1)\text{th bit}}\dots 00_k$, and $2^k - 1 = 11\dots 11_k$. If $N = 0$, there are no solutions; if $N = 1$, the only solution is $x_i = 1$ for all i; if $N > 1$, again there are no solutions. ■

$\mathcal{A}_{\exp}(k)$ has complexity of $\mathcal{O}(l^2) = \mathcal{O}(k^2 2^{2k})$, which is worse than the $\mathcal{O}(k^2 2^k)$ rate of the multiple linear jets scheme described in section 6.1. In the next section, we will give an algorithm with polynomial complexity in k, using a stochastic precomputation.

6.2.4 Polynomial Algorithm

The idea is to use the unstructuredness of primes. We first need to find an interval that contains at least k primes. By the prime number theorem, $\pi(k) \sim \frac{k}{\log k}$ where $\pi(k)$ is a function that counts the number of primes less than or equal to k. Therefore, $\pi(k^2) = \Theta\left(\frac{k^2}{\log k}\right)$ and there are more than k primes in the range $[0, Ck^2]$ for sufficiently large k and some C.

Lemma 3 *Given $k \geq 2$, let $\mathfrak{p}_k$ be the set of primes in the range within the range $[0, Ck^2]$, and let $\mathbf{p}$ be randomly uniformly drawn from the collection of all subsets of $\mathfrak{p}_k$ with cardinality k. Then for arbitrary $q \in \mathfrak{p}_k$, the r.v. $l := |\mathbf{p}|_{mul}$ modulus q has approximately uniform residual:*

$$\mathbb{P}_l[l \equiv r \mod q] = \frac{1}{q} + \mathcal{O}\left(k^{-\frac{3}{2}}\sqrt{\log k}\right). \tag{36}$$

This proof is technical, so we leave it in Appendix E.

Theorem 4 *Under the same setup as in Lemma 3, for sufficiently large k, the probability that $\mathbf{p}$ is not a unique minimal is $e^{-\Theta(k \log k)}$.*

Proof Fix a multiplicity vector $\mathbf{q} \neq (1, \dots, 1)$ with $q_i \geq 0$ and $\text{supp}(\mathbf{q}) \subset \text{supp}(\mathbf{p})$. The condition $|\mathbf{q}|_{\text{mul}} = l$ requires $\sum_i (q_i - 1) p_i = 0$, which implies $\sum_{i \neq j} (q_i - 1) p_i \equiv 0 \pmod{p_j}$ for every j. By a similar argument as in Lemma 3, the residue of $\sum_{i \neq j} (q_i - 1) p_i$ modulo p_j is approximately uniform for sufficiently large k, giving

$$\mathbb{P}\Big[\textstyle\sum_{i \neq j} (q_i - 1) p_i \equiv 0 \pmod{p_j}\Big] \leq \frac{2}{p_j}. \tag{37}$$

Since $p_1, \dots, p_k$ are pairwise coprime, the Chinese Remainder Theorem implies that these k divisibility conditions are asymptotically independent for large k, yielding

$$\mathbb{P}[|\mathbf{q}|_{\text{mul}} = l] \leq \mathbb{P}\Big[\textstyle\sum_{i \neq j} (q_i - 1) p_i \equiv 0 \pmod{p_j},\ \forall j\Big] \lesssim \displaystyle\prod_{j=1}^{k} \frac{2}{p_j} =: e^{-c_1 k}, \tag{38}$$

where $c_1 k = \sum_j \log(p_j / 2)$. Since each $q_i \leq l / \min(\mathbf{p}) \leq 2k$ for primes in $[k^2, Ck^2]$, the total number of candidate multiplicity vectors is at most $(2k+1)^k$. The union bound gives

$$\mathbb{P}[\exists \mathbf{q} \prec \mathbf{p}] \leq (2k{+}1)^k e^{-c_1 k} = \exp\big(k \log(2k{+}1) - c_1 k\big). \tag{39}$$

Since $c_1 \geq 2 \log k - \mathcal{O}(1)$ (as $p_j \geq k^2$ for most selected primes), the exponent is $-k(\log k - \mathcal{O}(1))$, giving decay $e^{-\Theta(k \log k)}$ for sufficiently large k. ■

k	exp sum	prime sum	primes
3	17	23	5, 7, 11
4	49	80	13, 17, 19, 31
5	129	199	29, 31, 37, 41, 61
6	321	560	71, 73, 79, 83, 103, 151
7	769	1357	127, 137, 157, 193, 197, 269, 277

Table 1: Output from the random prime algorithm. Uniqueness is verified by dynamic programming.

The above theorem implies that the probability of a random draw from $\mathfrak{p}_k$ produces a non-minimal $\mathbf{p}$ decrease exponentially. Although the step of finding a unique minimal has exponential complexity, this step can be precomputed and tabulated, and the resulting $\mathbf{p}$ can be stored, and the result is problem independent. Thus, we do not count the complexity for the sampling process.

We performed rejection sampling to obtain a unique minimal $\mathbf{p}$ for small k (see Table 1). Since $l \leq 2k^3$, the above construction has complexity of $\mathcal{O}(l^2) = \mathcal{O}(k^6)$. However, the prefactor of this scheme is large, and for the practical case of $k \leq 7$, the exponential algorithm described in section 6.2.3 yields a smaller l. The crossover point is expected to occur at a much larger k. We used the `primerange` function from `sympy` to generate the list of primes, and we used the Diophantine solver `diop_linear` from `sympy` to determine uniqueness and minimality. The corresponding code is in our GitHub repository.

7. STDE Applied to Some Common PDEs

In this section, we show some concrete examples for constructing sparse STDEs.

7.1 Differential Operators with Easy-to-Remove Mixed Partial Derivatives

Laplacian From Eq. 16 we know that the quadratic form of Hessian can be computed through ∂^2 by setting $\mathbf{x}^{(2)} = \mathbf{0}$ and $\mathbf{x}^{(1)} = \mathbf{e}_j$. Therefore, the STDE of the Laplacian operator is given by

$$\tilde{\nabla}^2{}_J u_\theta(\mathbf{a}) = \frac{d}{|J|} \sum_{j \in J} \frac{\partial^2}{\partial x_j^2} u_\theta(\mathbf{a}) = \frac{d}{|J|} \sum_{j \in J} \partial^2 u_\theta(\mathbf{a}, \mathbf{e}_j, \mathbf{0}) \tag{40}$$

where J is the sampled index set. See example implementation in JAX in Appendix A.4.

High-Order *Diagonal* Differential Operators From Eq. 15 we see that setting the first-order tangent $\mathbf{x}^{(1)}$ to $\mathbf{e}_j$ and all other tangents $\mathbf{x}^{(i)}$ to the zero vector gives the desired high-order diagonal element:

$$\tilde{\mathcal{L}}_J u_\theta(\mathbf{a}) = \frac{d}{|J|} \sum_{j \in J} \frac{\partial^k}{\partial \mathbf{x}_j^k} u_\theta(\mathbf{a}) = \frac{d}{|J|} \sum_{j \in J} \partial^k u_\theta(\mathbf{a}, \mathbf{e}_j, \mathbf{0}, \dots). \tag{41}$$

Second-Order Parabolic PDEs Second-order parabolic PDEs are a large class of PDEs. It includes the Fokker-Planck equation in statistical mechanics to describe the evolution of the

state variables in stochastic differential equations (SDEs), which can be used for generative modeling (Song et al., 2021). It also includes the Black-Scholes equation in mathematical finance for option pricing, the Hamilton-Jacobi-Bellman equation in optimal control, and the Schrödinger equation in quantum physics and chemistry. Its form is given by

$$\frac{\partial}{\partial t}u(\mathbf{x},t)+\frac{1}{2}\operatorname{tr}\left(\sigma\sigma^\top(\mathbf{x},t)\frac{\partial^2}{\partial\mathbf{x}^2}u(\mathbf{x},t)\right)+\nabla u(\mathbf{x},t)\cdot\mu(\mathbf{x},t)+f(t,\mathbf{x},u(\mathbf{x},t),\sigma^\top(\mathbf{x},t)\nabla u(\mathbf{x},t))=0. \tag{42}$$

We have a second-order derivative term $\frac{1}{2}\operatorname{tr}\left(\sigma(\mathbf{x},t)\sigma(\mathbf{x},t)^\top\frac{\partial^2}{\partial\mathbf{x}^2}u(\mathbf{x},t)\right)$ with the *off-diagonal* term. The off-diagonals can be easily removed via a change of variable:

$$\frac{1}{2}\operatorname{tr}\left(\sigma(\mathbf{x},t)\sigma(\mathbf{x},t)^\top\frac{\partial^2}{\partial\mathbf{x}^2}u(\mathbf{x},t)\right)=\frac{1}{2}\sum_{i=1}^{d}\partial^2 u((\mathbf{x},t),\sigma(\mathbf{x},t)\mathbf{e}_i,\mathbf{0}). \tag{43}$$

See the derivation in the appendix F.

7.2 Differential Operators with Arbitrary Mixed Partial Derivatives

It is not always possible to remove mixed partial derivatives. But as discussed in Section 6, for an arbitrary kth order derivative tensor element $D_u^k(\mathbf{a})_{n_1,\dots,n_k}$, we can find an appropriate l-jet J_x^l such that $\partial^l u(J_x^l)=D_u^k(\mathbf{a})_{n_1,\dots,n_k}$ using the single jet scheme. Here we show a concrete example.

2D Korteweg-de Vries (KdV) Equation Consider the following 2D KdV equation

$$u_{ty}+u_{xxxy}+3(u_y u_x)_x-u_{xx}+2u_{yy}=0. \tag{44}$$

All the derivative terms can be found in the pushforward of the following jet:

$$\begin{aligned}J_{\mathbf{y}}^{13}=\mathrm{d}^{13}u(J_{\mathbf{x}}^{13}),\ \mathbf{x}^{(3)}=\mathbf{e}_x,\mathbf{x}^{(4)}=\mathbf{e}_y,\mathbf{x}^{(7)}=\mathbf{e}_t,\mathbf{x}^{(i)}=\mathbf{0},\forall i\notin\{3,4,7\},\\ u_x=\mathbf{y}^{(1)},\ u_y=\mathbf{y}^{(2)},\ u_{xx}=\mathbf{y}^{(4)},\ u_{xy}=\mathbf{y}^{(5)}/35,\\ u_{yy}=\mathbf{y}^{(6)}/35,\ u_{ty}=\mathbf{y}^{(9)}/330,\ u_{xxxy}=\mathbf{y}^{(11)}/200200.\end{aligned} \tag{45}$$

where the prefactors are determined through Faa di Bruno's formula (Eq. 15). In this case, no randomization is needed since all the terms can be computed with just one pushforward. When the input dimension d is high, randomization via STDE will provide a significant speedup. We also tested a few more high-order PDEs with irremovable mixed partial derivatives (see Appendix I.4).

8. Experiments

We applied STDE to amortize the training of PINNs on a set of real-world PDEs. For the case of $k=2$ and large d, we tested two types of PDEs: inseparable and effectively high-dimensional PDEs (Appendix I.1) and semilinear parabolic PDEs (Appendix I.2). We also tested high-order PDEs (Appendix I.4) that cover the case of $k=3,4$, which includes PDEs describing 1D and 2D nonlinear dynamics, and high-dimensional PDEs with gradient regularization (Yu et al., 2022). Furthermore, we tested a weight-sharing technique (Appendix

G), which further reduces memory requirements (Appendix I.3). In all our experiments, STDE drastically reduces computation and memory costs in training PINNs, compared to the baseline method of SDGD with stacked backward-mode AD. Due to the page limit, the most important results are reported here, and the full details, including the experiment setup and hyperparameters (Appendix H), can be found in the Appendix.

8.1 Physics-Informed Neural Networks

PINN (Raissi et al., 2019) is a class of neural PDE solver where the ansatz $u_\theta(\mathbf{x})$ is parameterized by a neural network with parameter θ. It is a prototypical case of the optimization problem in Eq. 1. We consider PDEs defined on a domain $\Omega \subset \mathbb{R}^d$ and boundary/initial $\partial\Omega$ as follows

$$\mathcal{L}u(\mathbf{x}) = f(\mathbf{x}), \quad \mathbf{x} \in \Omega, \quad \mathcal{B}u(\mathbf{x}) = g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega, \tag{46}$$

where $\mathcal{L}$ and $\mathcal{B}$ are known operators, $f(\mathbf{x})$ and $g(\mathbf{x})$ are known functions for the residual and boundary/initial conditions, and $u : \mathbb{R}^d \to \mathbb{R}$ is a scalar-valued function, which is the unknown solution to the PDE. The approximated solution $u_\theta(\mathbf{x}) \approx u(\mathbf{x})$ is obtained by minimizing the mean squared error (MSE) of the PDE residual $R(\mathbf{x};\theta) = \mathcal{L}u_\theta(\mathbf{x}) - f(\mathbf{x})$:

$$\ell_{\text{residual}}(\theta; \{\mathbf{x}^{(i)}\}_{i=1}^{N_r}) = \frac{1}{N_r}\sum_{i=1}^{N_r}\left|\mathcal{L}u_\theta(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i)})\right|^2 \tag{47}$$

where the residual points $\{\mathbf{x}^{(i)}\}_{i=1}^{N_r}$ are sampled from the domain Ω. We use the technique from (Lu et al., 2021) that reparameterizes u_θ such that the boundary/initial condition $\mathcal{B}u(\mathbf{x}) = g(\mathbf{x})$ are satisfied exactly for all $\mathbf{x} \in \partial\Omega$, so boundary loss is not needed.

Amortized PINNs PINN training can be amortized by replacing the differential part of the operator $\mathcal{L}$ with a stochastic estimator like SDGD and STDE. For example, for the Allen-Cahn equation, $\mathcal{L}u = \nabla^2 u + u - u^3$, the differential part of $\mathcal{L}$ is the Laplacian ∇^2. With amortization, we minimize the following loss

$$\tilde{\ell}_{\text{residual}}(\theta; \{\mathbf{x}^{(i)}\}_{i=1}^{N_r}, J, K) = \frac{1}{N_r}\sum_{i=1}^{N_r}\left[\tilde{\mathcal{L}}_J u_\theta(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i)})\right]\cdot\left[\tilde{\mathcal{L}}_K u_\theta(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i)})\right], \tag{48}$$

which is a modification of Eq. 47. Its gradient $\frac{\partial\tilde{\ell}_{\text{residual}}}{\partial\theta}$ is then an unbiased estimator to the gradient of the original PINN residual loss, i.e. $\mathbb{E}[\frac{\partial\tilde{\ell}_{\text{residual}}}{\partial\theta}] = \frac{\partial\ell_{\text{residual}}}{\partial\theta}$.

8.2 Ablation Study on the Performance Gain

To ascertain the source performance gain of our method, we conduct a detailed ablation study on the inseparable Allen-Cahn equation with a two-body exact solution described in Appendix I.1. The results are in Table 2 and 3, where the best results for each dimensionality are marked in bold. Figure 3 visualizes the scaling trends. All methods were implemented using JAX unless stated. OOM indicates that the memory requirement exceeds 40 GB. Since the only change is how the derivatives are computed, the relative L2 error is expected to be of the same order among different randomization methods, as seen in Table 4 in the Appendix. We have included the Forward Laplacian, which is an exact method. It is expected to perform

better in terms of L2 error. However, as we can see in Table 4, the L2 error is of the same order, at least in the case where the dimension is more than 1000.

Speed (it/s) ↑	100 D	1K D	10K D	100K D	1M D
Backward mode SDGD (PyTorch) (Hu et al., 2024b)	55.56	3.70	1.85	0.23	OOM
Backward mode SDGD	40.63	37.04	29.85	OOM	OOM
Parallelized backward mode SDGD	1376.84	845.21	216.83	29.24	OOM
Forward-over-Backward SDGD	778.18	560.91	193.91	27.18	OOM
Forward Laplacian (Li et al., 2023)	**1974.50**	373.73	32.15	OOM	OOM
STDE	1035.09	**1054.39**	**454.16**	**156.90**	**13.61**

Table 2: Speed ablation for the two-body Allen-Cahn equation.

Memory (MB) ↓	100 D	1K D	10K D	100K D	1M D
Backward mode SDGD (PyTorch) (Hu et al., 2024b)	1328	1788	4527	32777	OOM
Backward mode SDGD	553	565	1217	OOM	OOM
Parallelized backward mode SDGD	539	579	1177	4931	OOM
Forward-over-Backward SDGD	537	579	1519	4929	OOM
Forward Laplacian (Li et al., 2023)	**507**	913	5505	OOM	OOM
STDE	543	**537**	**795**	**1073**	**6235**

Table 3: Memory ablation for the two-body Allen-Cahn equation.

JAX vs. PyTorch The original SDGD with stacked backward mode AD was implemented in PyTorch. We reimplement it in JAX (see Appendix A.1). From Table 2 and 3, JAX provides $\sim 15\times$ speed-up and up to $\sim 4\times$ memory reduction.

Parallelization The original SDGD implementation uses a for-loop to iterate through the sampled dimension (Appendix A.1). This can be parallelized (denoted as "Parallelized SDGD via HVP", details in Appendix A.2). Parallelization provides $\sim 15\times$ speed up and reduction in peak memory for the JIT compilation phase. We also tested mixed-mode AD (dubbed as "Forward-over-Backward SDGD"), which gives roughly the same performance as parallelized stacked backward mode, which is expected as explained in Appendix C.

Forward Laplacian Forward Laplacian (Li et al., 2023) provides a constant-level optimization for the calculation of the Laplacian operator by removing the redundancy in the AD pipeline, and we can see from Table 2 and 3 that it is the best method in both speed and memory when the dimension is 100. But since it is not a randomized method, the scaling is much worse. Its computation complexity is $\mathcal{O}(d)$, whereas a randomized estimator like STDE has a computation complexity of $\mathcal{O}(|J|)$. Naturally, with a high enough input dimension d, the difference in the constant prefactor is trumped by scaling. When the dimension is larger than 1000, it becomes worse than even parallelized stacked backward mode SDGD.

STDE Compared to the best realization of the baseline method, SDGD, the parallelized stacked backward mode AD, STDE provides up to $10\times$ speed up and memory reduction of at least $4\times$.

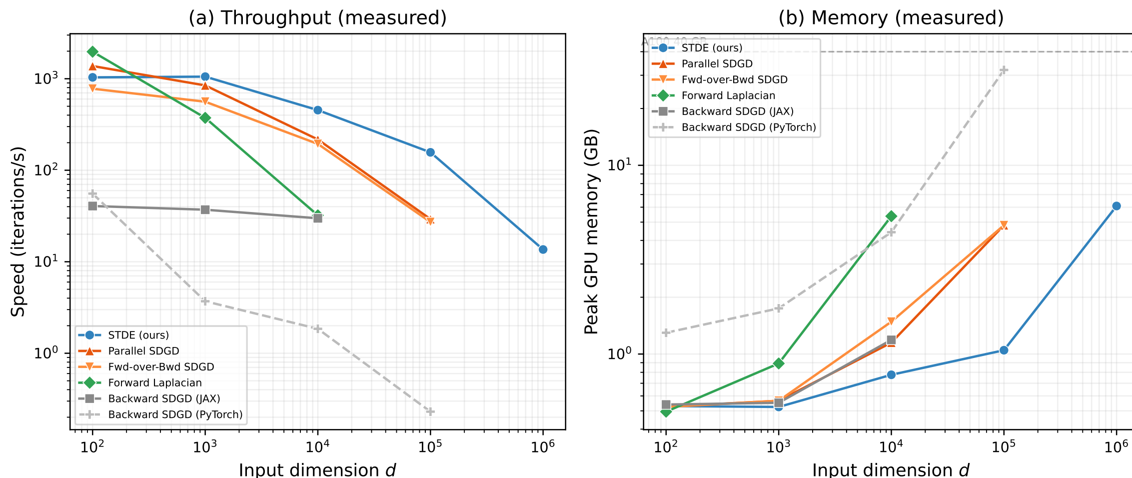

Figure 3: Performance analysis on A100 40 GB GPU for the two-body Allen-Cahn equation with randomization batch size $|J| = 100$. (a) Measured throughput. STDE is the only method that scales to $d = 10^6$. (b) Measured peak GPU memory. STDE maintains the lowest memory footprint across all dimensions tested.

9. Conclusion

We introduce STDE, a general method for constructing stochastic estimators for arbitrary differential operators that can be evaluated efficiently via Taylor mode AD. We evaluated STDE on PINNs, an instance of the optimization problem where the loss contains differential operators. Amortization with STDE outperforms the baseline methods, and STDE also applies to a wider class of problems, as it can be applied to arbitrary differential operators.

Applicability Besides PINNs, STDE can be applied to arbitrarily high-order and high-dimensional AD-based PDE solvers. This makes STDE more general than a branch of related methods. STDE is also more applicable than the deep Ritz method (Weinan and Yu, 2017), weak adversarial network (WAN) (Zang et al., 2020), backward SDE-based solvers (Beck et al., 2021; Raissi, 2018; Han et al., 2018), deep Galerkin method (Sirignano and Spiliopoulos, 2018), and the recently proposed forward Laplacian (Li et al., 2023), which are all restricted to specific forms of second-order PDEs. STDE applies naturally to differential operators in PDEs, but it can also be applied to other problems that require input gradients. For example, adversarial attacks, feature attribution, and meta-learning, to name a few.

Limitations Being a general method, STDE forgoes the optimization possibilities that apply to specific operators. Furthermore, we did not consider variance reduction techniques that could be applied, which can be explored in future work. Also, we observed that lowering the randomization batch size improves both speed and memory profile, but the trade-off between cheaper computation and larger variance needs further analysis. Furthermore, the

method is not suited for computing the high-order derivative of neural network parameters as explained in Section 3.

Future Work The key insight of the STDE construction is that the univariate Taylor-mode AD contains arbitrary contraction of the derivative tensor and that differential operators are derivative tensor contractions. This shows the connection between the fields of AD and randomized numerical linear algebra and indicates that further work in the intersection of these two fields might bring significant progress in large-scale scientific modeling with neural networks. One example would be the many-body Schrödinger equations, where one needs to compute a high-dimensional Laplacian. Another example is the high-dimensional Black-Scholes equation, which has numerous uses in mathematical finance.

Acknowledgments

This material is based upon work supported by the Air Force Office of Scientific Research under award number FA2386-24-1-4011, and this research is partially supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (Award No: T1 251 RES2207).

Appendix A. Example Implementations

A.1 PyTorch Implementation of SDGD-PINN Using Backward-Mode AD

The original implementation of SDGD-PINN (Hu et al., 2024b) computes the SDGD estimation of derivatives using a for-loop that iterates over the sampled PDE term/dimension. For example, given a function f representing the MLP PINN, the computation of SDGD for the Laplacian operator can be implemented in PyTorch as follows:

```
f_x = torch.autograd.grad(f.sum(), x, create_graph=True)[0]
idx_set = np.random.choice(dim, sdgd_batch_size, replace=False)
hess_diag_val = 0.
for i in idx_set:
    hess_diag_i = torch.autograd.grad(
        f_x[:, i].sum(), x, create_graph=True)[0][:, i]
    hess_diag_val += hess_diag_i.detach() * dim / sdgd_batch_size
```

After computing the PDE differential operator, it is plugged into the residual loss, and then backward-mode AD is employed to produce the gradient for optimization concerning θ.

A.2 JAX Implementation of SDGD Parallelization via HVP

```
def hvp(f, x, v):
  """stacked backward-mode Hessian-vector product"""
  return jax.grad(lambda x: jnp.vdot(jax.grad(f)(x), v))(x)

f_hess_diag_fn = lambda i: hvp(f_partial, x_i, jnp.eye(dim)[i])[i]
idx_set = jax.random.choice(
  key, dim, shape=(sdgd_batch_size,), replace=False
)
hess_diag_val = jax.vmap(f_hess_diag_fn)(idx_set)
```

A.3 JAX Implementation of Forward-over-Backward AD

The forward-over-backward AD in JAX mentioned in Appendix C can be implemented as follows:

```
f_grad_fn = jax.grad(f)
f_x, f_hess_fn = jax.linearize(f_grad_fn, x_i)  # jvp over vjp
f_hess_diag_fn = lambda i: f_hess_fn(jnp.eye(dim)[i])[i]
hess_diag_val = jax.vmap(f_hess_diag_fn)(idx_set)
```

A.4 JAX Implementation of STDE for the Laplacian Operator

```
idx_set = jax.random.choice(
  key, dim, shape=(batch_size,), replace=False
)
rand_jet = jax.vmap(lambda i: jnp.eye(dim)[i])(idx_set)
pushfwd_2_fn = lambda v: jet.jet(
```

```
  fun=fn, primals=(x,), series=((v, jnp.zeros(dim)),)
) # pushforward of the 2-jet (x, v, 0), i.e. \dd^2 f(x, v, 0)
f_vals, (_, vhv) = jax.vmap(pushfwd_2_fn)(rand_jet)
hess_diag_val = dim / batch_size * vhv
```

The `jet.jet` function from JAX implements the high-order directional derivatives d^n of jets in Eq. 21. It decomposes the input function into primitives, which have analytical derivatives derived up to arbitrary order, and uses the generalized chain rule (see section 22) to compose the primitives into the directional derivative of the given function. Note that in the API of `jet.jet`, all the high-order tangents of the input jet are specified via the `series` argument.

Appendix B. First-Order Auto-Differentiation

AD is a technique for evaluating the gradients of the composition of known functions, commonly called primitives. It is based on the chain rule: The Jacobian of the composition $F = F_L \circ F_{L-1} \circ \cdots \circ F_1 : \mathbb{R}^m \to \mathbb{R}^n$ at a point $\mathbf{a} \in \mathbb{R}^m$ is the product of the Jacobian of the primitives evaluated at the evaluation trace $\mathbf{y}_i = [F_{i-1} \circ \cdots \circ F_1](\mathbf{a})$:

$$\left.\frac{\partial F}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{a}} = \left.\frac{\partial F_L}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{y}_{L-1}} \left.\frac{\partial F_{L-1}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{y}_{L-2}} \cdots \left.\frac{\partial F_1}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{a}} \in \mathbb{R}^{m\times n}. \tag{49}$$

Therefore, given a set of functions with known analytic Jacobians, called the primitives, one can evaluate the Jacobian of an arbitrary composition of the primitives.

Forward-Mode AD Instantiating the full Jacobians of the primitives is memory-intensive. Instead, one can compute the Jacobian column-wise with the Jacobian-vector-product (JVP) map $(\mathbf{a}, \mathbf{v}) \mapsto \left.\frac{\partial F}{\partial \mathbf{x}}\right|_{\mathbf{a}} \mathbf{v}$, since one can compute ith column of the Jacobian by setting the tangent $\mathbf{v}$ to the ith standard basis. JVP can be computed using the JVP of the primitives, which can be seen by right multiplying $\mathbf{v}$ to Equation 49. Now, one only needs to store the intermediate JVP $\mathbf{v}_i = \left.\frac{\partial F_{L-1}}{\partial \mathbf{x}}\right|_{\mathbf{y}_{L-2}} \cdots \left.\frac{\partial F_1}{\partial \mathbf{x}}\right|_{\mathbf{a}} \mathbf{v} \in \mathbb{R}^m$ in addition to $\mathbf{y}_i$ instead of the full Jacobian. Furthermore, many primitives have efficient JVP implementations due to sparsity. For example, element-wise application of a scalar function (e.g., activation in neural networks) has a diagonal Jacobian, and its JVP can be efficiently implemented as a Hadamard product. Another prominent example is discrete convolution, whose JVP has an efficient implementation via FFT.

Backward-Mode AD When the output of the compute graph is scalar, e.g., when optimizing scalar cost functions $\ell(\theta) : \mathbb{R}^n \to \mathbb{R}$, it is desirable to compute the full Jacobian row-wise since there is only one row. This corresponds to the vector-Jacobian-product (VJP) $(\mathbf{a}, \mathbf{v}^\top) \mapsto \mathbf{v}^\top \left.\frac{\partial F}{\partial \mathbf{x}}\right|_{\mathbf{a}}$, which is the transpose of JVP. For the scalar output case, the cotangent $\mathbf{v}^\top$ is set to 1.

Similarly, the VJP of the computation graph can be computed via the VJP of the primitives. However, due to the reversed dependency to the evaluation trace, one need to first perform a forward pass to obtain and store the full evaluation trace $\{\mathbf{y}_i\}_{i=1}^{L-1}$ before computing the backward pass, as can be seen in Figure 4.

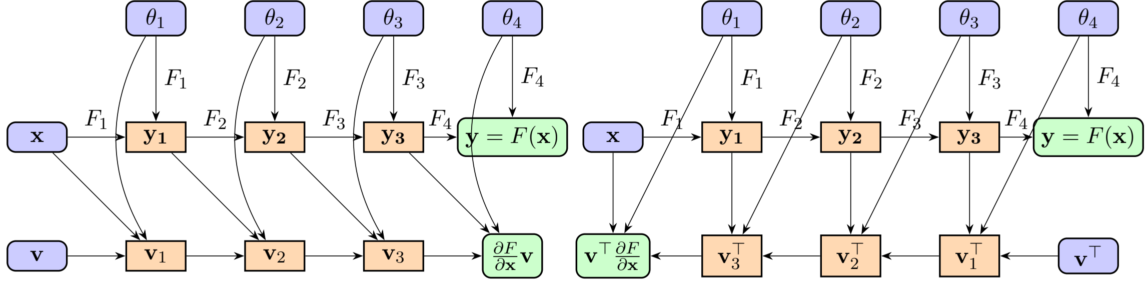

Figure 4: The computation graph of forward mode AD (left) and backward mode AD (right) of a function $F(\cdot)$ with 4 primitives F_i each parameterized by θ_i. Nodes represent (intermediate) values, and arrows represent computation. Input nodes are colored blue; output nodes are colored green, and intermediate nodes are colored yellow.

Appendix C. Why Mixed-Mode AD Schemes Like Forward-over-Backward Might Not Be Better Than Stacked Backward-Mode AD for PINNs

In AD literature (Griewank and Walther, 2008), the second-order derivative is recommended to be computed via forward-over-backward AD, i.e., first do a backward mode AD to get the first-order derivative, then apply forward mode AD to the first-order derivative to obtain the second-order derivative. Usually, we will expect that forward-over-backward AD gives better performance in memory usage over stacked backward AD since the outer differential operator has to differentiate a larger computation graph than the inner one, and forward AD has less overhead as explained in section B. Essentially, forward-over-backward reverses the arrows in the third row in Fig. 1, therefore reducing the number of sequential computations and also the size of the evaluation trace. However, in the case of PINN, yet another differentiation of the network parameters θ needs to be taken. So, computing the second-order differential operator here with forward-over-backward AD might not yield any advantage.

Appendix D. Proof Sketch for Faa di Bruno's Formula

We will prove Faa di Bruno's formula (Eq. 15) using induction. For $k = 1$, it is just the first-order chain rule:

$$\frac{\mathrm{d}}{\mathrm{d}t} F(x(t))\bigg|_{t=0} = D_F(\mathbf{x}^{(0)}) \cdot \frac{1}{j!}\mathbf{x}^{(1)}. \tag{50}$$

Consider now applying $\frac{\mathrm{d}}{\mathrm{d}t}$ to the case of $k-1$. Consider one term on the right hand side with $|\mathbf{p}|_{\text{mul}} = k-1$. $\frac{\mathrm{d}}{\mathrm{d}t}$ can either act on $D_F^{|\mathbf{p}|}$ or one of $\mathbf{x}^{(i)}$. If it acts on $D_F^{|\mathbf{p}|}$, we will have $D_F^{|\mathbf{p}|+1} \cdot \mathbf{x}^{(1)}$ by the first order chain rule. Let the resulting multiset partition be $\mathbf{p}'$, we have

$$p_1' = p_1 + 1, \quad |\mathbf{p}'| = |\mathbf{p}| + 1, \quad |\mathbf{p}'|_{\text{mul}} = |\mathbf{p}|_{\text{mul}} + 1 = k, \tag{51}$$

which implies $|\mathbf{p}'|_{\text{mul}} = k$. If $\frac{\mathrm{d}}{\mathrm{d}t}$ acts on one of $\mathbf{x}^{(i)}$, we have

$$p_i' = p_i - 1, \quad p_{i+1}' = p_{i+1} + 1, \quad |\mathbf{p}'| = |\mathbf{p}|, \quad |\mathbf{p}'|_{\text{mul}} = |\mathbf{p}|_{\text{mul}} + 1 = k, \tag{52}$$

which also implies $|\mathbf{p}'|_{\mathrm{mul}} = k$. It is not hard to see that the right-hand side must include all $\mathbf{p}$ with $|\mathbf{p}|_{\mathrm{mul}} = k$

Appendix E. Proof of Lemma 3

Here we give a self-contained but slightly verbose proof.

E.1 Characterizing Collision Probability with the Additive Character of $(\mathbb{Z}/q\mathbb{Z}, +)$

Denote the qth root-of-unity as $\omega = e^{\mathrm{i}2\pi/q}$, then $\psi_t(r) = \omega^{rt}$ is an additive character for the group $(\mathbb{Z}/q\mathbb{Z}, +)$, since it has period q and $\psi_t(r + r') = \psi_t(r)\psi_t(r')$. $\{\psi_t\}_{t=0}^{q-1}$ form a complete orthogonal basis and therefore satisfy the completeness relation

$$\frac{1}{q}\sum_{t=0}^{q-1}\psi_t(u)\overline{\psi_t(v)} = \frac{1}{q}\sum_{t=0}^{q-1}\omega^{(u-v)t} = \delta_{uv} = \begin{cases} 1, & u \equiv v \mod q \\ 0, & u \not\equiv v \mod q \end{cases} \tag{53}$$

Short proof on the orthogonality: When $u \equiv v \mod q$, each summand is 1, so the average is 1. When $u \not\equiv v \mod q$, $\omega^{t(u-v)}$ is a geometric series and the sum is given by $\frac{1-(\omega^{u-v})^q}{1-\omega^{u-v}} = 0$. From the completeness relation, we immediately have

$$\mathbb{P}_l[l \equiv r \mod q] = \mathbb{E}_l\left[\frac{1}{q}\sum_{t=0}^{q-1}\psi_t(l)\overline{\psi_t(r)}\right] = \frac{1}{q}\sum_{t=0}^{q-1}\omega^{-rt}\mathbb{E}_l\left[\omega^{lt}\right]. \tag{54}$$

Since the $t = 0$ term contributes $\frac{1}{q}$, we just need to show that when $t \neq 0$, $\left|\mathbb{E}\left[\omega^{lt}\right]\right|$ can be upperbounded.

E.2 Residuals of p_i Follow a Hypergeometric Distribution

Let $r_i = p_i \mod q$, and define the count of elements with residual r in $\mathbf{p}$ as $C_r = |\{p_i \equiv r \mod q | p_i \in \mathbf{p}\}|$, and in $\mathfrak{p}_k$ as $N_r = |\{p \equiv r \mod q | p \in \mathfrak{p}_k\}|$. Then the distribution of the vector $\mathbf{C} = (C_0, \ldots, C_{q-1})$ follows multivariate hypergeometric distribution:

$$\mathbb{P}[\mathbf{C}] = \prod_{r=0}^{q-1}\binom{N_r}{C_r}\Big/\binom{|\mathfrak{p}_k|}{k} \tag{55}$$

for admissible $\mathbf{C}$, i.e. $C_r \leq N_r, \sum_r C_r = k$. Since all elements in $\mathfrak{p}_k$ is in $[k^2, 2k^2]$, no two p_i can have the same residual, we have $N_r \in \{0, 1\}$, and the distribution simplifies to $\mathbb{P}[\mathbf{C}] = 1/\binom{|\mathfrak{p}_k|}{k}$ for admissible $\mathbf{C}$. Furthermore, all r_i are distinct, which means $\mathbf{r} = \{r_1, \ldots, r_k\} \subset \mathbf{R} = \{r \in \{0, \ldots, q-1\} | N_r = 1\}$ where $\mathbf{R}$ is the set of residuals in $\mathfrak{p}_k$. Note that $|\mathbf{R}| = |\mathfrak{p}_k|$. Now for any t, we have

$$\mathbb{E}[\omega^{lt}] = \mathbb{E}\left[\prod_{j=1}^{k}\omega^{r_j t}\right] = \mathbb{E}_{\mathbf{C}}\left[\prod_{r=0}^{q-1}\omega^{C_r rt}\right] = \frac{1}{\binom{|\mathfrak{p}_k|}{k}}\sum_{\substack{C_r \leq N_r,\\ \sum_r C_r = k}}\prod_{r=0}^{q-1}\omega^{C_r rt} = \frac{1}{\binom{|\mathfrak{p}_k|}{k}}\sum_{\substack{\mathbf{r}\subset\mathbf{R},\\ |\mathbf{r}|=k}}\omega^{\sum_{i=1}^{k} r_i t}. \tag{56}$$

E.3 Simulating the Hypergeometric Distribution with Bernoulli Trials

Now let's try to bound $\left|\mathbb{E}[\omega^{lt}]\right|$. Note that the hypergeometric distributed C_r can be simulated via a two-step sampling process:

1. For each $r \in \mathbf{R}$, draw a i.i.d. random indicator I_r from the Bernoulli distribution with $p = \frac{k}{|\mathbf{R}|}$. Then any r, we have $\mathbb{E}[\omega^{I_r rt}] = (1-p) + p\omega^{rt}$.

2. If $|\mathbf{I}| - k > 0$, sample $||\mathbf{I}| - k|$ items without replacement from the set $\{r : I_r = 1\}$, and set I_r to 0. Similarly, set I_r to 1 if $|\mathbf{I}| - k < 0$. Denote the adjusted indicator as J_r, then J_r has the same distribution as C_r.

Let $E(\mathbf{I}, t) = \mathbb{E}_{\mathbf{I}}\left[\prod_{r\in\mathbf{R}} \omega^{I_r rt}\right]$, then by triangular inequality:

$$\left|\mathbb{E}[\omega^{lt}]\right| = |E(\mathbf{J}, t)| \leq |E(\mathbf{J}, t) - E(\mathbf{I}, t)| + |E(\mathbf{I}, t)|. \tag{57}$$

We will process to bound the two terms on the right-hand side.

E.4 Bounding $|E(\mathbf{I}, t)|$

E.4.1 Dirichlet Characters and Gauss Sums

A Dirichlet characters is a function $\chi : (\mathbb{Z}/q\mathbb{Z}, \times) \to \mathbb{C}^{\times}$ that satisfies $\chi(nm) = \chi(n)\chi(m)$ for nm coprime to q, and $\chi(o) = 0$ for o not coprime to q. Dirichlet characters are also an orthogonal basis:

$$\frac{1}{q-1} \sum_{\chi \mod q} \chi(u)\overline{\chi(v)} = \delta_{uv}. \tag{58}$$

where the summation sums over all Dirichlet characters of $(\mathbb{Z}/q\mathbb{Z}, \times)$.

For each Dirichlet characters χ, denote its inner product with ψ_t as $\tau_t(\chi) = \sum_{a=1}^{q} \psi_t(a)\overline{\chi(a)}$, which is also known as the Gauss sum. From the completeness of ψ_t we have that $\tau_t(\chi_0) = 0$ for $t \not\equiv 0 \mod q$ where χ_0 is the principle character. For non-principle χ, we have $|\tau_t(\chi)| = \sqrt{q}$:

$$\tau_t(\chi)\overline{\tau_t(\chi)} = \sum_{c=0}^{q-1}\sum_{a=1}^{q} \chi(a)\overline{\chi(a+c)}\psi_t(c) = \sum_{c=0}^{q-1} \delta_{c0}\psi_t(c) = q. \tag{59}$$

Naturally, we can represent ψ_t with χ with a change of basis

$$\begin{aligned}
\frac{1}{q-1} \sum_{\chi \mod q} \chi(n)\tau_t(\chi) =& \frac{1}{q-1} \sum_{\chi \mod q} \chi(n) \left(\sum_{a=1}^{q} \overline{\chi(a)}\psi_t(a)\right) \\
=& \frac{1}{q-1} \sum_{a=1}^{q} \psi_t(a) \sum_{\chi \mod q} \overline{\chi(a)}\chi(n) \\
=& \frac{1}{q-1} \sum_{a=1}^{q} \psi_t(a)\delta_{na} = \psi_t(n).
\end{aligned} \tag{60}$$

E.4.2 The Bound

We first need the following result: for any $z = e^{\mathrm{i}\theta}$ and $p = \frac{k}{|\mathbf{R}|}$,

$$|(1-p)+pz| \leq \exp\left[-\frac{p}{2}(1-\cos\theta)\right]. \tag{61}$$

Let's prove the above claim. Firstly

$$|(1-p)+pz|^2 = [(1-p)+p\cos\theta]^2 + (p\sin\theta)^2 = 1-2(1-p)p(1-\cos\theta). \tag{62}$$

Since $1-x \leq e^{-x}$ for $x<1$, we have

$$|(1-p)+pz| \leq \exp\left[-(1-p)p(1-\cos\theta)\right] \tag{63}$$

Now since by PNT, $|\mathbf{R}| = |\mathfrak{p}_k| = \Theta(\frac{k^2}{\log k})$, $p = \frac{k}{|\mathbf{R}|} = \Theta(\frac{\log k}{k})$, so $p < \frac{1}{2}$ for sufficiently large k. Thus

$$|(1-p)+pz| \leq \exp\left[-\frac{p}{2}(1-\cos\theta)\right]. \tag{64}$$

Now

$$\begin{aligned}\left|\mathbb{E}[\omega^{\sum_{r\in\mathbf{R}} I_r r t}]\right| = \prod_{r\in\mathbf{R}} \left|(1-p)+p\omega^{rt}\right| &\leq \exp\left[\sum_{r\in\mathbf{R}} -\frac{p}{2}(1-\cos(2\pi rt/q))\right] \\ &= \exp\left[-\frac{p}{2}\left(|\mathbf{R}| - \operatorname{Re}\sum_{r\in\mathbf{R}}\psi_t(r)\right)\right] \\ &\leq \exp\left[-\frac{p}{2}\left(|\mathbf{R}| - \left|\sum_{r\in\mathbf{R}}\psi_t(r)\right|\right)\right].\end{aligned} \tag{65}$$

Now, using the Dirichlet representation derived in the previous section, applying the Cauchy–Schwarz inequality, we have

$$\begin{aligned}\left|\sum_{r\in\mathbf{R}}\psi_t(r)\right| &= \left|\frac{1}{q-1}\sum_{r\in\mathbf{R}}\sum_{\chi \mod q}\chi(n)\tau_t(\chi)\right| \\ &\leq \frac{1}{q-1}\left(\sum_{\chi \mod q}|\tau_t(\chi)|^2\right)^{\frac{1}{2}}\left(\sum_{\chi \mod q}\left|P(\chi;k^2,2k^2)\right|^2\right)^{\frac{1}{2}}.\end{aligned} \tag{66}$$

where $P(\chi;x,y) := \sum_{\substack{x\leq p\leq y,\\ p \text{ prime}}}\chi(p)$. Let $S(x) = \sum_{n=1}^{x}\chi(n)$ be the partial sum, by Pólya-Vinogradov inequality, for any Dirichlet character χ over $(\mathbb{Z}/q\mathbb{Z},\times)$, we have

$$|S(x)| = \left|\sum_{n=1}^{x}\chi(n)\right| \leq 2\sqrt{q}\log q. \tag{67}$$

Now we just need to convert the summation over $\mathbf{R}$ to a partial sum $\sum_{n=1}^{x}$. Now let

$$\theta(x,\chi) = \sum_{n\leq x}\Lambda(n)\chi(n), \quad \Lambda(n) = \begin{cases}\log p, & n = p^m \text{ for prime } p \\ 0, & \text{otherwise}\end{cases} \tag{68}$$

where $\Lambda(n)$ is the von Mangoldt function. Let $a(n) = \Lambda(n)\chi(n), A(t) = \sum_{0\le n\le t} a(n), \phi(n) = \frac{1}{\log n}$, then by Abel summation we have

$$\begin{aligned}\sum_{x\le n\le y} a(n)\phi(n) =& A(y)\phi(y) - A(x)\phi(x) - \int_x^y A(u)\phi'(u)\,\mathrm{d}u \\ P(\chi;x,y)+\epsilon =& \frac{\theta(y,\chi)}{\log y} - \frac{\theta(x,\chi)}{\log x} + \int_x^y \frac{\theta(u,\chi)}{u(\log u)^2}\,\mathrm{d}u\,.\end{aligned} \tag{69}$$

First, we quantify the error term ϵ, which includes all prime powers p^m with $m \ge 2$. For $m = 2$, $p^2 \in [k^2, 2k^2]$ implies $p \in [k, \sqrt{2}k]$, and again by PNT, the number of such p is $\Theta(\frac{k}{\log k})$, which contributes to $\mathcal{O}(k)$ to ϵ. The $m \ge 3$ terms are dominated by the $m = 2$ term, so $\epsilon = \mathcal{O}(k)$. Next, we bound $\theta(x,\chi)$ with Pólya-Vinogradov. For $x = k^2, y = 2k^2$, we have

$$\begin{aligned}\theta(x,\chi) =& \int_1^x \log u\,\mathrm{d}S(u) \\ =& \log x S(x) - \int_1^x \frac{1}{u} S(u)\,\mathrm{d}u \\ \Rightarrow |\theta(x,\chi)| \le& 2\sqrt{q}\log q\left(\log x + \int_1^x \frac{\mathrm{d}u}{u}\right) \le Ck\log^2 k.\end{aligned} \tag{70}$$

Then

$$\begin{aligned}|P| = |P(\chi;x,y)| \le& |P(\chi;x,y)+\epsilon| + |\epsilon| \\ \le& C_1 k\log^2 k\left(\frac{1}{\log k} + \int_x^y \frac{1}{u(\log u)^2}\,\mathrm{d}u\right) + C_2 k \le Ck\log k.\end{aligned} \tag{71}$$

Now substitute the above into Eq. 66, and use the fact that $\tau_t(\chi) = \sqrt{q}$, we get

$$\left|\sum_{r\in\mathbf{R}} \psi_t(r)\right| \le Ck\log k \quad \Rightarrow \quad \frac{k}{|\mathbf{R}|}\left|\sum_{r\in\mathbf{R}} \psi_t(r)\right| = \mathcal{O}\big(\log^2 k\big). \tag{72}$$

Finally, substitute the above into Eq. 65 we get

$$\left|\mathbb{E}[\omega^{\sum_{r\in\mathbf{R}} I_r r t}]\right| \le \exp\left[\frac{k}{|\mathbf{R}|}\left|\sum_{r\in\mathbf{R}} \psi_t(r)\right| - k\right] = e^{-\Theta(k)}. \tag{73}$$

E.5 Bounding $|E(\mathbf{J},t) - E(\mathbf{I},t)|$

Let $K_r = |I_r - J_r|$ be the indicators of the flipping going from I to J, then $D := |\mathbf{K}| = \pm(|\mathbf{I}| - k)$. Let $u_{r,t} = \omega^{\pm K_r r t}$, then $|u_{r,t}| = 1$ and

$$|E(\mathbf{J},t) - E(\mathbf{I},t)| = \left|\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t}]\right| \tag{74}$$

Since $|\mathbf{I}|$ has binomial distribution, we will first calculate the expectation under the condition D then average, i.e. $\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t}] = \mathbb{E}_D[\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t}|D]]$. The conditional expectation is

given by

$$\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t} \mid D] = \frac{1}{\binom{|\mathbf{R}|}{D}} \sum_{\substack{\mathbf{S}\subset\mathbf{R},\\ |\mathbf{S}|=D}} \prod_{r\in\mathbf{S}} u_{r,t} = \frac{1}{\binom{|\mathbf{R}|}{D}} e_D(\{u_{r,t}\}) \tag{75}$$

where e_D denotes the Dth order elementary symmetric polynomial. Let $P_1 = \sum_{r\in\mathbf{R}} u_{r,t}$, then $P_1^D = D! e_D(\{u_{r,t}\}) + T_1$ where T_1 includes all terms with repeated indicies r. Since $|u_{r,t}| = 1$, $|T_1|$ is bounded by the number of terms in T_1, i.e. $|T_1| \leq |\mathbf{R}|^D - \binom{|\mathbf{R}|}{D}$. Now by the reverse triangular inequality $|x - y| \geq ||x| - |y||$:

$$\begin{aligned} \left|P_1^D\right| \geq & D!|e_D(\{u_{r,t}\})| - |T_1| \geq D!|e_D(\{u_{r,t}\})| - \left[|\mathbf{R}|^D - \binom{|\mathbf{R}|}{D}\right] \\ \Rightarrow (1+o(1))\left|P_1^D\right| \geq & D!|e_D(\{u_{r,t}\})| \end{aligned} \tag{76}$$

Since $u_{r,t}$ is a qth root of unity which distributes uniformly on the unit circle, we can treat them like i.i.d. samples that are drawn from the uniform distribution on the unit circle. Hence $\mathbb{E}[|u_{r,t}|^2] = 1$ and $\mathbb{E}[|S_1|^2] = |\mathbf{R}|$. So

$$|P_1| \leq C_0\sqrt{|\mathbf{R}|}. \tag{77}$$

Putting these two together, we get

$$|e_D(\{u_{r,t}\})| \leq \frac{1}{D!}(1+o(1))(C_0\sqrt{|\mathbf{R}|})^D \tag{78}$$

and thus

$$\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t} \mid D] \leq \frac{1}{\binom{|\mathbf{R}|}{D}} \frac{1}{D!}(1+o(1))(C_0\sqrt{|\mathbf{R}|})^D \leq \quad (1+o(1))\left(\frac{2C_0}{\sqrt{|\mathbf{R}|}}\right)^D. \tag{79}$$

where we used the fact that $|\mathbf{R}| \gg D$, and in particular $|\mathbf{R}| - D > \frac{|\mathbf{R}|}{2}$.

Next, we compute the expectation over D. Let $\alpha = C_0|\mathbf{R}|^{-\frac{1}{2}} = \mathcal{O}\big(k^{-1}\sqrt{\log k}\big) < 1$, then

$$\begin{aligned} \mathbb{E}_D[\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t} \mid D]] \leq & \mathbb{E}_D[\alpha^D] = \sum_{d=1}^{\infty} \alpha^d \mathbb{P}[D=d] \\ & = \underbrace{\sum_{d=1}^{\lceil\sqrt{k}\rceil} \alpha^d \mathbb{P}[D=d]}_{S_1} + \underbrace{\sum_{d=\lceil\sqrt{k}\rceil+1}^{\infty} \alpha^d \mathbb{P}[D=d]}_{S_2}. \end{aligned} \tag{80}$$

Note that the $D = 0$ term is excluded since all $K_r = 0$ in that case.

E.5.1 Bounding Tail Sum S_2

To bound the $d > \lceil\sqrt{k}\rceil$ summation S_2, we will use a tail bound for $\mathbb{P}[D \geq d]$. Since $|\mathbf{I}|$ is the sum of i.i.d. r.v. $I_r \in \{0, 1\}$, where $\mathbb{E}[I_r] = \frac{k}{|\mathbf{R}|}$, we have $\mathbb{E}[|\mathbf{I}|] = k$ and $\sigma^2 := \mathbb{V}[|\mathbf{I}|] = |\mathbf{R}|p(1-p) = k(1-p) \leq k$. By Bernstein's inequality and the fact that $d > 0$:

$$\mathbb{P}[D \geq d] = \mathbb{P}[||\mathbf{I}| - k| \geq d] \leq \exp\left(-\frac{d^2}{2\sigma^2 + \frac{2}{3}d}\right) \leq \exp\left(-\frac{d^2}{2k + \frac{2}{3}d}\right) \leq \exp\left(-\frac{d^2}{2k}\right). \tag{81}$$

Now

$$\begin{aligned} S_2 &\leq \sum_{d=\lceil\sqrt{k}\rceil+1}^{\infty} \alpha^d \mathbb{P}[D \geq d] \leq \sum_{d=\lceil\sqrt{k}\rceil+1}^{\infty} \alpha^d \exp\left(-\frac{d}{2\sqrt{k}}\right) = \sum_{d=\lceil\sqrt{k}\rceil+1}^{\infty} \left[\alpha e^{-\frac{1}{2\sqrt{k}}}\right]^d \\ &\leq \sum_{d=\lceil\sqrt{k}\rceil+1}^{\infty} 2^{-d} = \mathcal{O}\left(2^{-\sqrt{k}}\right) = \mathcal{O}\left(k^{-3}\right). \end{aligned} \tag{82}$$

In the last inequality, we used the fact that $\alpha < 1$.

E.5.2 Bounding Small-d Summation S_1

For the $d \leq \lceil\sqrt{k}\rceil$ summation S_1, we need a stronger bound on D. First note that $\mathbb{P}[D = d]$ is given by the binomial PMF

$$\mathbb{P}[D = d] = \mathbb{P}[|\mathbf{I}| = k \pm d] = \binom{|\mathbf{R}|}{k \pm d} p^{k\pm d}(1-p)^{|\mathbf{R}|-(k\pm d)}. \tag{83}$$

By De Moivre-Laplace theorem, the binomial PMF converges locally to a normal distribution:

$$\mathbb{P}[D = d] = \frac{1}{\sqrt{2\pi k(1-p)}} \exp\left[-\frac{d^2}{2k(1-p)}\right](1 + o(1)) = \mathcal{O}\left(k^{-\frac{1}{2}}\right) \tag{84}$$

Now

$$S_1 = \sum_{d=1}^{\lceil\sqrt{k}\rceil} \alpha^d \mathbb{P}[D = d] \leq Ck^{-\frac{1}{2}} \sum_{d=1}^{\lceil\sqrt{k}\rceil} \left[\alpha e^{-\frac{d}{2k}}\right]^d \leq C' k^{-\frac{1}{2}} \alpha = \mathcal{O}\left(k^{-\frac{3}{2}} \sqrt{\log k}\right) \tag{85}$$

Putting everything together, we get

$$\mathbb{E}_D[\mathbb{E}[\prod_{r\in\mathbf{R}} u_{r,t} \mid D]] \leq \mathcal{O}\left(k^{-\frac{3}{2}} \sqrt{\log k}\right). \tag{86}$$

E.6 Bounding the Collision Probability $\mathbb{P}[l \equiv 0 \mod q]$

Now we are ready to put everything together. Firstly, substitute Eq. 80 and Eq. 65 into Eq. 57 we get

$$\left|\mathbb{E}[\omega^{lt}]\right| \leq |E(\mathbf{J}, t) - E(\mathbf{I}, t)| + |E(\mathbf{I}, t)| \leq \mathcal{O}\left(k^{-\frac{3}{2}} \sqrt{\log k}\right). \tag{87}$$

Finally, substitute the above into Eq. 54 and use the fact that $\left|\omega^{rt}\right| = 1$, we get

$$\mathbb{P}[l \equiv r \mod q] = \frac{1}{q} + \frac{1}{q} \sum_{t=1}^{q-1} \omega^{-rt} \mathbb{E}\left[\omega^{lt}\right] \leq \frac{1}{q} + \mathcal{O}\left(k^{-\frac{3}{2}} \sqrt{\log k}\right). \tag{88}$$

Appendix F. Removing the Mixed Partial Derivatives Term from a Second-Order Semilinear Parabolic PDE

$$\begin{aligned}\frac{1}{2}\operatorname{tr}\left(\sigma(\mathbf{x},t)\sigma(\mathbf{x},t)^\top(\mathrm{Hess}_{\mathbf{x}}u)(\mathbf{x},t)\right) =&\frac{1}{2}\operatorname{tr}\left(\sigma(\mathbf{x},t)^\top(\mathrm{Hess}_{\mathbf{x}}u)(\mathbf{x},t)\sigma(\mathbf{x},t)\right)\\ =&\frac{1}{2}\sum_{i=0}^{d}\left[\sigma(\mathbf{x},t)^\top(\mathrm{Hess}_{\mathbf{x}}u)(\mathbf{x},t)\sigma(\mathbf{x},t)\right]_{i,i}\\ =&\frac{1}{2}\sum_{i=0}^{d}\mathbf{e}_i^\top\sigma(\mathbf{x},t)^\top(\mathrm{Hess}_{\mathbf{x}}u)(\mathbf{x},t)\sigma(\mathbf{x},t)\mathbf{e}_i\\ =&\frac{1}{2}\sum_{i=0}^{d}\partial^2u((\mathbf{x},t),\sigma(\mathbf{x},t)\mathbf{e}_i,\mathbf{0}^\top).\end{aligned} \tag{89}$$

Appendix G. Further Memory Reduction via Weight Sharing in the First Layer

When dealing with high-dimensional data, the parameters of the model's first layer in a conventional fully connected network would grow proportionally with the input dimension, resulting in a significant increase in memory requirements and forming a memory bottleneck due to massive model parameters. To address this issue, convolutional networks are often employed in deep learning for images to reduce the number of model parameters. Here, we adopt a similar approach to mitigate the memory cost of model parameters in high-dimensional PDEs, called weight sharing in the first layer.

Denote the input dimension as d, which is potentially excessively high, and the hidden dimension of the MLP as h, and assume that $d \gg h$. The first layer weight is an $d \times h$ dimensional matrix, whereas all subsequent layers have a weight matrix with a size of only $h \times h$.

By introducing a weight-sharing scheme, one can reduce the redundancy in the parameters in the first layer. Specifically, we perform an additional 1D convolution on the input vectors $\mathbf{x}_i$ before passing the input into the MLP PINN, as in Fig. 5. The 1D convolution has a filter size B that divides D and a stride size B, so the convolution output is non-overlapping, and the number of channels is set to 1.

This weight-sharing scheme reduces the parameters by approximately $\frac{1}{B}$. The number of parameters in the filters is $B \times 1$, and the subsequent fully connected layer will have a weight matrix of size $\frac{d}{B} \times H$. Therefore, the total number of the first layer is reduced from $d \times h$ to only $\frac{d}{B} \times h + B$, and we can see that with a larger block size B, we will have fewer parameters, and the reduction factor is approximately $\frac{1}{B}$. More concretely, suppose $d = 10^6, h = 100$ where one million (10^6) dimensional problems are also tested experimentally, so the number of parameters in the first layer is $d \times h = 100 \times 10^6$. If we use a block size of $B = 100$, we will reduce the number of parameters to $\frac{d}{B} \times h + B = 10^6 + 100$. If the block size is $B = 10$, the number of parameters will be $\frac{d}{B} \times h + B = 10 \times 10^6 + 10$. In other words, with a larger block size of B, we significantly reduce the number of model parameters.

We will demonstrate the memory efficiency and acceleration thanks to weight-sharing in the experimental section.

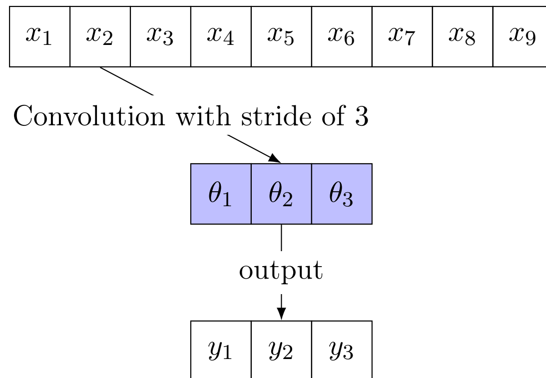

Figure 5: Convolutional weight sharing in the first layer, with input dimension 9 and filter size 3.

Appendix H. Experiment Setup

Each experiment is run with five different random seeds, and the average and the standard deviations of these runs are reported.

To get an accurate reading of memory usage, we use a separate run where GPU memory pre-allocation for JAX is disabled, and the test data set is stored on the CPU memory. The pre-allocation is disabled by setting the environment variable `XLA_PYTHON_CLIENT_ALLOCATOR=platform`. The GPU memory usage was obtained via `NVIDIA-smi` and peak memory was reported.

All the experiments were done on a single NVIDIA A100 GPU with 40GB of memory and CUDA 12.2. with driver 535.129.03 and JAX version 0.4.23.

Network Architecture and Training Hyperparameters For the semilinear parabolic PDEs tested in Appendix I.2 We follow the network architecture of the original SDGD (Hu et al., 2024b):

- The network is a 4-layer multi-layer perceptron (MLP) with 128 hidden units activated by Tanh.
- The network is trained with Adam (Kingma and Ba, 2015) for 10K steps, with an initial learning rate of 1e-3 that linearly decays to 0 in 10K steps, where at each step we calculate the model parameters' gradient with 100 uniformly sampled random residual points.
- The model is evaluated using 20K uniformly sampled random points fixed throughout the training.
- The zero boundary condition is satisfied via the following parameterization

$$u_\theta(\mathbf{x}) = (1 - \|\mathbf{x}\|_2^2) u_\theta^{\mathrm{MLP}}(\mathbf{x}) \tag{90}$$

  where u_θ^{MLP} is the MLP network, and u_θ is the PDE ansatz, as described in (Lu et al., 2021).

For the semilinear parabolic PDEs tested in Appendix I.2, we made the following modifications:

- Instead of using re-parameterization, the boundary/initial condition is satisfied by adding a regularization loss to the residual loss:

$$\ell_{\text{boundary}}(\theta; \{\mathbf{x}_{b,i}\}_{i=1}^{N_b}) = \frac{1}{N_b}\sum_{i=1}^{N_b} |u_\theta(\mathbf{x}_{b,i}, 0) - g(\mathbf{x}_{b,i})|^2 + C_g \cdot \frac{1}{N_b}\sum_{i=1}^{N_b} |\nabla u_\theta(\mathbf{x}_{b,i}, 0) - \nabla g(\mathbf{x}_{b,i})|^2 \tag{91}$$

  where $g(\cdot)$ is the initial data, N_b is the batch size for boundary points, u_θ is the PDE ansatz, C_g is the coefficient for the first-order derivative boundary loss term, which we set to 0.05. The total loss is

$$\ell_{\text{residual}}(\theta; \{\mathbf{x}_{r,i}\}_{i=1}^{N_r}) + 20\ell_{\text{boundary}}(\theta; \{\mathbf{x}_{b,i}\}_{i=1}^{N_b}). \tag{92}$$

- Instead of discretizing the time and sample residual points using the underlying stochastic process, we uniformly sample the time steps between the initial and the terminal time, i.e. $t \sim \text{uniform}[0, T]$, and then sample $\mathbf{x}$ directly from the distribution of $\mathbf{X}_t$, i.e. $\mathbf{x} \sim \mathcal{N}(0, (T-t) \cdot \mathbf{I}_{d\times d})$. To match the original training setting of 100 SDE trajectories with 0.015 step size for time discretization, we use a batch size of 2000 for residual points and 100 for boundary/initial points.

- We use a 4-layer multi-layer perceptron (MLP) with 1024 hidden units activated by Tanh. The network is trained with Adam (Kingma and Ba, 2015) for 10K steps, with an initial learning rate of 1e-3 that exponentially decays with exponent 0.9995.

- To test the quality of the PINN solution, we measure the relative L1 error at the point $(\mathbf{x}_{\text{test}}, T)$ against the reference value computed via multilevel Picard's method (Beck et al., 2021; Becker et al., 2020; Hutzenthaler et al., 2018).

  In all experiments, we use the biased version of Eq. 48:

$$\tilde{\ell}_{\text{residual}}(\theta; \{\mathbf{x}^{(i)}\}_{i=1}^{N_r}, J) = \frac{1}{N_r}\sum_{i=1}^{N_r} \left[\tilde{\mathcal{L}}_J u_\theta(\mathbf{x}^{(i)}) - f(\mathbf{x}^{(i)})\right] \tag{93}$$

As the bias in practice is very small and does not affect convergence.

Appendix I. Experimental Results

I.1 Inseparable and Effectively High-Dimensional PDEs

The first class of PDEs is defined via a nonlinear, inseparable, and effectively high-dimensional exact solution $u_{\text{exact}}(\mathbf{x})$ defined within the d-dimensional unit ball $\mathbb{B}^d$:

$$\begin{aligned} \mathcal{L}u(\mathbf{x}) &= f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{B}^d \\ u(\mathbf{x}) &= 0, \quad \mathbf{x} \in \partial\mathbb{B}^d \end{aligned} \tag{94}$$

where $\mathcal{L}$ is a linear/nonlinear operator and $g(\mathbf{x}) = \mathcal{L}u_{\text{exact}}(\mathbf{x})$. The zero boundary condition ensures that no information about the exact solution is leaked through the boundary condition. We will consider the following operators:

- Poisson equation: $\mathcal{L}u(\mathbf{x}) = \nabla^2 u(\mathbf{x})$.
- Allen-Cahn equation: $\mathcal{L}u(\mathbf{x}) = \nabla^2 u(\mathbf{x}) + u(\mathbf{x}) - u(\mathbf{x})^3$.
- Sine-Gordon equation: $\mathcal{L}u(\mathbf{x}) = \nabla^2 u(\mathbf{x}) + \sin(u(\mathbf{x}))$.

For the exact solution, we consider the following with all $c_i \sim \mathcal{N}(0,1)$:

- two-body interaction: $u_{\text{exact}}(\mathbf{x}) = (1-\|\mathbf{x}\|_2^2)\left(\sum_{i=1}^{d-1} c_i \sin(x_i + \cos(x_{i+1}) + x_{i+1}\cos(x_i))\right)$.
- three-body interaction: $u_{\text{exact}}(\mathbf{x}) = (1-\|\mathbf{x}\|_2^2)\left(\sum_{i=1}^{d-2} c_i \exp(x_i x_{i+1} x_{i+2})\right)$.

We tested the performance of STDE on these equations, and the results are presented in Table 4, 5, 6, 7. For the Allen-Cahn equation, we performed a detailed ablation study (Table 4), and we expect these results to generalize over these second-order PDEs.

I.1.1 Further Details on the Ablation Study

The Gain from Using JAX Instead of PyTorch Since the original SDGD was implemented in PyTorch, we implemented the stacked backward mode without parallelization in SDGD dimensions in JAX for fair comparison (dubbed as "Stacked Backward mode SDGD in JAX" in Table 4). The for-loop over the SDGD dimension is implemented using `jax.lax.scan`. Table 4 shows that, even with the original stacked backward mode AD, the speed of JAX implementation can be more than 10× faster when the dimension is high. The memory profile is similar. The difference could come from the fact that JAX uses XLA to perform Just-in-time (JIT) compilation of the Python code into optimized kernels. However, note that for the case of 100,000 dimensions, the JAX implementation of the stacked backward mode AD encountered an out-of-memory (OOM) error. This is because performing JIT compilation requires extra memory, and the peak memory requirement during JIT compilation is higher than that during training.

Randomization Batch Size We also tested the case where the STDE randomization batch size is reduced to 16. As seen in Table 4, in the case of Allen-Cahn provides ∼2× speed up, without hurting performance. However, theoretically, lowering the randomization batch size leads to higher variance. The trade-off between computational efficiency and stability in convergence warrants further studies.

I.2 Semilinear Parabolic PDEs

The second class of PDEs is the semilinear parabolic PDEs, where the initial condition is specified:

$$\begin{aligned}\frac{\partial}{\partial t}u(\mathbf{x},t) =& \mathcal{L}u(\mathbf{x},t) \quad (\mathbf{x},t)\in\mathbb{R}^d\times[0,T]\\ u(\mathbf{x},t) =& g(\mathbf{x}), \quad (\mathbf{x},t)\in\mathbb{R}^d\times\{0\}\end{aligned} \tag{95}$$

where $g(\mathbf{x})$ is a known, analytical, and time-independent function that specifies the initial condition, and T is the terminal time. We aim to approximate the solution's true value at one test point $\mathbf{x}_{\text{test}} \in \mathbb{R}^d$, at the terminal time $t = T$, i.e., at $(\mathbf{x}_{\text{test}}, T)$.

We will consider the following operators

Method	Metric	100 D	1K D	10K D	100K D	1M D
Backward mode SDGD (PyTorch) (Hu et al., 2024b)	Speed	55.56it/s	3.70it/s	1.85it/s	0.23it/s	OOM
	Memory	1328MB	1788MB	4527MB	32777MB	OOM
	Error	7.187E-03	5.615E-04	1.864E-03	2.178E-03	OOM
Backward mode SDGD (JAX)	Speed	40.63it/s	37.04it/s	29.85it/s	OOM	OOM
	Memory	553MB	565MB	1217MB	OOM	OOM
	Error	3.51E-03 ±8.47E-05	7.29E-04 ±5.45E-06	3.46E-03 ±2.01E-04	OOM	OOM
Parallelized backward mode SDGD	Speed	1376.84it/s	845.21it/s	216.83it/s	29.24it/s	OOM
	Memory	539MB	579MB	1177MB	4931MB	OOM
	Error	6.87E-03 ±6.97E-05	3.12E-03 ±7.04E-04	2.59E-03 ±2.20E-05	1.60E-03 ±1.13E-05	OOM
Forward-over -Backward SDGD	Speed	778.18it/s	560.91it/s	193.91it/s	27.18it/s	OOM
	Memory	537MB	579MB	1519MB	4929MB	OOM
	Error	4.07E-03 ±7.42E-05	2.19E-03 ±2.03E-04	5.47E-04 ±7.48E-05	4.21E-03 ±2.53E-04	OOM
Forward Laplacian (Li et al., 2023)	Speed	**1974.50it/s**	373.73it/s	32.15it/s	OOM	OOM
	Memory	507MB	913MB	5505MB	OOM	OOM
	Error	4.33E-03 ±4.97E-05	5.50E-04 ±4.60E-05	5.58E-03 ±2.73E-04	OOM	OOM
STDE	Speed	1035.09it/s	1054.39it/s	454.16it/s	156.90it/s	13.61it/s
	Memory	543MB	537MB	795MB	1073MB	**6235MB**
	Error	1.03E-02 ±7.69E-05	6.21E-04 ±2.22E-04	3.45E-03 ±1.17E-05	2.59E-03 ±7.93E-06	1.38E-03 ±3.34E-05
STDE (batch size=16)	Speed	1833.78it/s	**1559.36it/s**	**587.60it/s**	**283.33it/s**	**21.34it/s**
	Memory	**457MB**	**481MB**	**741MB**	**1063MB**	6295MB
	Error	1.89E-02 ±2.37E-04	7.07E-04 ±1.02E-05	8.33E-04 ±2.96E-04	1.50E-03 ±1.02E-05	3.99E-03 ±3.41E-05

Table 4: Computational results for the Inseparable Allen-Cahn equation with the two-body exact solution, where the randomization batch size is set to 100 unless stated otherwise.

Method	Metric	100D	1K D	10K D	100K D	1M D
Backward mode SDGD (PyTorch) (Hu et al., 2024b)	Speed	55.56it/s	3.70it/s	1.85it/s	0.23it/s	OOM
	Memory	1328MB	1788MB	4527MB	32777MB	OOM
	Error	7.189E-03	5.611E-04	1.850E-03	2.175E-03	OOM
STDE (batch size=16)	Speed	2020.05it/s	1649.20it/s	584.98it/s	281.78it/s	20.38it/s
	Memory	457MB	481MB	741MB	1063MB	6295MB
	Error	3.50E-03 ±1.44E-04	4.91E-04 ±3.45E-05	4.70E-03 ±2.10E-05	3.49E-03 ±2.14E-05	9.18E-04 ±6.39E-06

Table 5: Computational results for the Inseparable Poisson equation with two-body exact solution.

Method	Metric	100 D	1K D	10K D	100K D	1M D
Backward mode SDGD (PyTorch) (Hu et al., 2024b)	Speed	55.56it/s	3.70it/s	1.85it/s	0.23it/s	OOM
	Memory	1328MB	1788MB	4527MB	32777MB	OOM
	Error	7.192E-03	5.641E-04	1.854E-03	2.177E-03	OOM
STDE (batch size=16)	Speed	1926.33it/s	1467.38it/s	566.26it/s	279.24it/s	19.88it/s
	Memory	457MB	481MB	741MB	1063MB	6295MB
	Error	3.64E-03 ±1.46E-04	5.40E-04 ±7.21E-05	5.32E-03 ±5.12E-04	9.56E-04 ±8.03E-06	9.47E-04 ±8.30E-06

Table 6: Computational results for the Inseparable Sine-Gordon equation with two-body exact solution.

Eq.	Metric	100 D	1K D	10K D	100K D	1M D
Allen-Cahn	Speed	1938.80it/s	1840.21it/s	1291.67it/s	356.76it/s	46.97it/s
	Memory	461MB	481MB	539MB	1055MB	6233MB
	Error	9.97E-03 ±3.89E-04	1.43E-03 ±1.60E-04	6.21E-04 ±8.15E-05	1.56E-05 ±3.28E-07	2.25E-06 ±1.48E-07
Poisson *	Speed	1991.28it/s	1872.31it/s	1276.21it/s	364.04it/s	31.73it/s
	Memory	473MB	481MB	539MB	1055MB	6233MB
	Error	1.00E-02 ±3.27E-04	1.02E-03 ±3.67E-05	1.01E-04 ±2.40E-07	9.26E-02 ±5.36E-04	4.82E-06 ±2.16E-07
Sine-Gordon	Speed	1938.80it/s	1840.21it/s	1291.67it/s	356.76it/s	46.88it/s
	Memory	475MB	479MB	539MB	1063MB	6233MB
	Error	9.97E-03 ±3.89E-04	1.43E-03 ±1.60E-04	6.21E-04 ±8.15E-05	1.56E-05 ±3.28E-07	2.31E-05 ±1.48E-06

Table 7: Computational results for the Inseparable Allen-Cahn, Poisson, and Sine-Gordon equation with the three-body exact solution, computed via STDE with randomization batch size $|J|$ set to 16. *STDE with randomization batch size ($|J|$) of 16 performs poorly on the 1M-dimensional Inseparable Poisson equation with three-body exact solution: the L2 relative error is only 9.05E-02± 6.88E-04. To get better convergence, we increase the randomization batch size to 50 for the 1M case. This incurs no extra memory cost and is only slightly slower than the original setting (speed is 46.80it/s when randomization batch size is 16).

- Semilinear Heat Eq.

$$\mathcal{L}u(\mathbf{x},t) = \nabla^2 u(\mathbf{x},t) + \frac{1-u(\mathbf{x},t)^2}{1+u(\mathbf{x},t)^2}. \tag{96}$$

  with initial condition $g(\mathbf{x}) = 5/(10+2\|\mathbf{x}\|^2)$,

- Allen-Cahn equation

$$\mathcal{L}u(\mathbf{x},t) = \nabla^2 u(\mathbf{x},t) + u(\mathbf{x},t) - u(\mathbf{x},t)^3. \tag{97}$$

  with initial condition $g(\mathbf{x}) = \arctan(\max_i x_i)$,

- Sine-Gordon equation

$$\mathcal{L}u(\mathbf{x},t) = \nabla^2 u(\mathbf{x},t) + \sin(u(\mathbf{x},t)). \tag{98}$$

  with initial condition $g(\mathbf{x}) = 5/(10+2\|\mathbf{x}\|^2)$,

All three equation uses the test point $\mathbf{x}_{\text{test}} = \mathbf{0}$ and terminal time $T = 0.3$.

Method	Metric	10 D	100 D	1K D	10K D
Backward mode SDGD (PyTorch) (Hu et al., 2024b)	Speed	-	-	-	-
	Memory	-	-	-	-
	Error	1.052E-03	5.263E-04	6.910E-03	1.598E-03
BackwardBackward mode SDGD (JAX)	Speed	211.63it/s	207.66it/s	188.31it/s	93.21it/s
	Memory	**619MB**	**621MB**	**655MB**	1371MB
	Error	8.55E-05 ±6.75E-05	4.02E-04 ±2.07E-04	3.81E-04 ±4.43E-04	2.60E-03 ±1.38E-03
STDE	Speed	**660.82it/s**	**635.16it/s**	**599.15it/s**	**361.11it/s**
	Memory	625MB	625MB	657MB	**971MB**
	Error	6.99E-05 ±5.78E-05	3.69E-04 ±2.19E-04	3.38E-04 ±3.30E-04	6.08E-03 ±7.47E-03

Table 8: Computational results for the Time-dependent Semilinear Heat equation, where the number of SDGD sampled dimensions is set to 10.

I.3 Weight Sharing

We tested the weight-sharing technique mentioned in Section G.

In this section, we evaluate the performance of the weight-sharing scheme described in Appendix G. We tested the best-performing method from Table 4 (STDE with a small randomization batch size of 16) with different weight-sharing block sizes, on the inseparable Allen-Cahn equation with the two-body exact solution.

From Table 11, we can see that weight sharing drastically reduces the number of network parameters and memory usage. With $B = 50$, there is a 2.5x reduction in memory, and there is no performance loss in terms of L2 relative error.

Method	Metric	10 D	100 D	1K D	10K D
BackwardBackward mode SDGD (PyTorch) (Hu et al., 2024b)	Speed	-	-	-	-
	Memory	-	-	-	-
	Error	7.815E-04	3.142E-04	7.042E-04	2.477E-04
Backward mode SDGD (JAX)	Speed	211.38it/s	206.42it/s	188.02it/s	93.20it/s
	Memory	619MB	621MB	657MB	1371MB
	Error	6.31E-02 ±3.79E-02	4.38E-03 ±2.48E-03	1.35E-03 ±1.23E-03	3.97E-04 ±3.03E-04
STDE	Speed	**677.51it/s**	**650.98it/s**	**598.33it/s**	**361.31it/s**
	Memory	**533MB**	**535MB**	657MB	**903MB**
	Error	6.37E-02 ±3.77E-02	4.38E-03 ±2.47E-03	1.26E-03 ±1.29E-03	3.79E-04 ±2.75E-04

Table 9: Computational results for the Time-dependent Allen-Cahn equation, where the number of SDGD sampled dimensions is set to 10.

Method	Metric	10 D	100 D	1K D	10K D
BackwardBackward mode SDGD (PyTorch) (Hu et al., 2024b)	Speed	-	-	-	-
	Memory	-	-	-	-
	Error	7.815E-04	3.142E-04	7.042E-04	2.477E-04
BackwardBackward mode SDGD (JAX)	Speed	210.83it/s	207.44it/s	187.98it/s	93.17it/s
	Memory	619MB	621MB	655MB	1371MB
	Error	5.39E-05 ±4.10E-05	9.15E-05 ±6.06E-05	4.19E-04 ±2.18E-04	3.74E-02 ±4.15E-02
STDE	Speed	629.04it/s	608.83it/s	596.12it/s	365.09it/s
	Memory	525MB	539MB	655MB	971MB
	Error	4.15E-05 ±3.21E-05	2.54E-04 ±1.76E-04	4.05E-03 ±1.44E-02	1.66E-02 ±5.95E-03

Table 10: Computational results for the Time-dependent Sine-Gordon equation, where the number of SDGD sampled dimensions is set to 10.

However, from the experiments we can see that, in both the 1M and the 5M case, increasing the block size beyond 50 provides diminishing returns. For the 1M case, increasing B to 1000 affects the convergence quality, as the L2 relative error goes up by 100x. For 5M, the maximum block size one can use before degrading performance is 500, which is expected as the dimensionality of the problem is higher.

From Table 11 we can also see that in the 5M-dimensional case, we will have an out-of-memory (OOM) error without weight sharing. With weight sharing enabled, we can effectively solve the 5M-dimensional PDE with good relative L2 error, in around 30 minutes.

dim		$B=1$	$B=10$	$B=50$	$B=100$	$B=500$	$B=1000$
1M	Speed	21.34it/s	16.67it/s	23.14it/s	23.73it/s	25.47it/s	26.60it/s
	Memory	6295MB	4819MB	2505MB	2461MB	2409MB	2403MB
	#Param.	128,033,281	12,833,292	2,593,332	1,313,382	289,782	162,282
	Error	3.99E-03 ±3.41E-05	1.86E-02 ±3.13E-04	4.76E-03 ±1.27E-04	1.22E-03 ±6.05E-05	2.57E-03 ±1.15E-04	6.06E-01 ±4.17E-04
5M	Speed	OOM	3.16it/s	4.47it/s	4.74it/s	4.82it/s	4.76it/s
	Memory	OOM	25023MB	10595MB	10359MB	10163MB	10143MB
	#Param.	640,033,281	64,033,292	12,833,332	6,433,382	1,313,782	674,282
	Error	OOM	5.11E-01 ±4.01E-04	3.13E-03 ±2.34E-04	3.94E-03 ±2.22E-04	1.98E-03 ±5.20E-05	6.27E-01 ±3.03E-04

Table 11: Effects of different weight sharing block sizes B for the Inseparable Allen-Cahn equation with two-body exact solution solved with STDE with randomization batch size of 16. $B=1$ equals no weight sharing.

I.4 High-Order PDEs

Here we demonstrate how to use STDE to calculate mixed partial derivatives in some actual PDEs. We will consider the 2D Korteweg-de Vries (KdV) equation and the 2D Kadomtsev-Petviashvili equation from (Pu and Chen, 2024), and the regular 1D KdV equation with gPINN (Yu et al., 2022).

We will demonstrate that STDE increases the speed for computing the mixed partial derivatives, as it avoids computing the entire derivative tensor. Since these equations are low-dimensional, we do not need to sample over the space dimension.

In this section, the equations are all time-dependent, and the space is 2D, and we will omit the argument to the solution, i.e., we will write $u(\mathbf{x}, t) = u$. To test the speed improvement, we run the STDE implementation against repeated backward mode AD on an Nvidia A100 GPU with 40 GB of memory. The results are reported in Table 12. From the Table, we see that STDE provides around $\sim 2\times$ speed up compared to repeated application of backward mode AD across different network sizes.

I.4.1 High-Order Low-Dimensional PDEs

I.4.2 Alternative Way to Compute the Terms in the 2D Korteweg-de Vries (KdV) Equation

The terms in the 2D KdV equation

$$u_{ty} + u_{xxxy} + 3(u_y u_x)_x - u_{xx} + 2u_{yy} = 0. \tag{99}$$

can alternatively be computed with the pushforward of the following jets

$$\mathbf{y}_1 = \mathrm{d}^9 u(\mathbf{x}, \mathbf{0}, \mathbf{e}_x, \mathbf{e}_y, \mathbf{0}, \dots), \ \ \mathbf{y}_2 = \mathrm{d}^3 u(\mathbf{x}, \mathbf{0}, \mathbf{e}_y, \mathbf{e}_t), \ \ \mathbf{y}_3 = \mathrm{d}^3 u(\mathbf{x}, \mathbf{0}, \mathbf{e}_y, \mathbf{0}). \tag{100}$$

All the derivative terms can be found in these output jets $\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3\}$:

$$\begin{gathered} u_x = \mathbf{y}_1^{(2)}, \ u_y = \mathbf{y}_1^{(3)}, \ u_{xx} = \mathbf{y}_1^{(4)}/3, \ u_{xy} = \mathbf{y}_1^{(5)}/10, u_{yy} = \mathbf{y}_3^{(2)}, \\ u_{yyy} = \mathbf{y}_3^{(3)}, u_{xxxy} = (\mathbf{y}_1^{(9)} - 280 u_{yyy})/840, \ u_{ty} = (\mathbf{y}_2^{(3)} - u_{yyy})/3, \end{gathered} \tag{101}$$

2D Kadomtsev-Petviashvili (KP) Equation Consider the following equation

$$(u_t + 6uu_x + u_{xxx})_x + 3\sigma^2 u_{yy} = 0. \tag{102}$$

which can be expanded as

$$u_{tx} + 6u_x u_x + 6uu_{xx} + u_{xxxx} + 3\sigma^2 u_{yy} = 0. \tag{103}$$

All the derivative terms can be computed with a 5-jet, a 4-jet, and a 2-jet pushforward. Let

$$\begin{aligned} \mathbf{y}_1 &:= \mathrm{d}^5 u(\mathbf{x}, \mathbf{0}, \mathbf{e}_t, \mathbf{e}_x, \mathbf{0}, \mathbf{0}) \\ \mathbf{y}_2 &:= \mathrm{d}^4 u(\mathbf{x}, \mathbf{e}_x, \mathbf{0}, \mathbf{0}, \mathbf{0}) \\ \mathbf{y}_3 &:= \mathrm{d}^2 u(\mathbf{x}, \mathbf{e}_y, \mathbf{0}). \end{aligned} \tag{104}$$

Then all required derivative terms can be evaluated as follows.

$$\begin{aligned} u_{tx} &= \mathbf{y}_1^{(5)}/10, \\ u_x = \mathbf{y}_2^{(1)}, \ u_{xx} = \mathbf{y}_2^{(2)}, \ u_{xxxx} &= \mathbf{y}_2^{(4)}, \\ u_{yy} &= \mathbf{y}_3^{(2)}. \end{aligned} \tag{105}$$

Gradient-Enhanced 1D Korteweg-de Vries (g-KdV) Equation Consider the following equation

$$u_t + uu_x + \alpha u_{xxx} = 0. \tag{106}$$

Gradient-enhanced PINN (gPINN) (Yu et al., 2022) regularizes the learned PINN such that the gradient of the residual is close to the zero vector. This increases the accuracy of the solution. Specifically, the PINN loss (Eq. 47) is augmented with the term

$$\ell_{\text{gPINN}}\left(\{\mathbf{x}^{(i)}\}_{i=1}^{N_r}\right) = \frac{1}{N_r} \sum_i \sum_j^d \left| \frac{\partial}{\partial x_j} R(\mathbf{x}^{(i)}) \right|^2. \tag{107}$$

The total loss becomes

$$\ell_{\text{residual}} + c_{\text{gPINN}}\,\ell_{\text{gPINN}} \tag{108}$$

where c_{gPINN} is the g-PINN penalty weight. To perform gradient enhancement, we need to compute the gradient of the residual:

$$\begin{aligned} R(x,t) &:= u_t + u u_x + \alpha u_{xxx}, \\ \nabla R(x,t) &= \left[u_{tt} + u_t u_x + u u_{tx} + \alpha u_{txxx}, \quad u_{tx} + u_x u_x + u u_{xx} + \alpha u_{xxxx}\right]. \end{aligned} \tag{109}$$

All the derivative terms can be computed with one 2-jet and two 7-jet pushforward. Let

$$\begin{aligned} \mathbf{y}_1 &:= \mathrm{d}^7 u(\mathbf{x}, \mathbf{e}_x, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}) \\ \mathbf{y}_2 &:= \mathrm{d}^7 u(\mathbf{x}, \mathbf{e}_x, \mathbf{0}, \mathbf{0}, \mathbf{e}_t, \mathbf{0}, \mathbf{0}, \mathbf{0}) \\ \mathbf{y}_3 &:= \mathrm{d}^2 u(\mathbf{x}, \mathbf{e}_t, \mathbf{0}). \end{aligned} \tag{110}$$

Then all required derivative terms can be evaluated as follows.

$$\begin{aligned} u_x = \mathbf{y}_1^{(1)},\ u_{xx} = \mathbf{y}_1^{(2)},\ u_{xxx} = \mathbf{y}_1^{(3)},\ u_{xxxx} = \mathbf{y}_1^{(4)},\ u_{xxxxx} = \mathbf{y}_1^{(5)}, \\ u_{txxx} = (\mathbf{y}_2^{(7)} - \mathbf{y}_1^{(8)})/35,\ u_{tx} = (\mathbf{y}_2^{(5)} - u_{xxxxx})/5,\ u_t = \mathbf{y}_2^{(4)} - u_{xxxx}, \\ u_{tt} = \mathbf{y}_3^{(2)}. \end{aligned} \tag{111}$$

Speed (it/s) ↑	network size	Base	$L=8$	$L=16$	$h=256$	$h=512$	$h=1024$
2D KdV	Backward	762.86	279.19	123.20	656.01	541.10	349.23
	STDE	1372.41	642.82	303.39	1209.30	743.75	418.13
	STDE*	1357.64	606.43	272.01	1203.97	841.07	442.32
2D KP	Backward	766.79	278.53	123.67	642.34	525.23	340.94
	STDE	1518.82	676.16	304.95	1498.61	1052.62	642.21
1D g-KdV	Backward	621.04	232.35	102.39	559.65	482.52	293.97
	STDE	1307.27	593.21	253.48	1187.31	776.65	441.50

Table 12: Speed scaling for training low-dimensional high-order PDEs with different network sizes. The base network has depth $L = 4$ and width $h = 128$. STDE* is the alternative scheme using lower-order pushforwards.

I.4.3 Amortized Gradient-Enhanced PINN for High-Dimensional PDEs

It is expensive to apply gradient enhancement for high-dimensional PDEs. For example, the gradient of the residual for the inseparable Allen-Cahn equation described in I.1 is given by

$$\begin{aligned} \frac{\partial}{\partial x_j} R(\mathbf{x}) &= \frac{\partial}{\partial x_j}\left[\sum_i \frac{\partial^2}{\partial x_i^2} u(\mathbf{x}) + u(\mathbf{x}) - u^3(\mathbf{x}) - f(\mathbf{x})\right] \\ &= \sum_{i=1}^{d} \frac{\partial^3}{\partial x_j \partial x_i^2} u(\mathbf{x}) + \frac{\partial}{\partial x_j} u(\mathbf{x}) - 3u^2(\mathbf{x}) \frac{\partial}{\partial x_j} u(\mathbf{x}) - \frac{\partial}{\partial x_j} f(\mathbf{x}). \end{aligned} \tag{112}$$

With STDE randomization, we randomized the second order term $\frac{\partial^2}{\partial x_i^2}$ with index i sampled from $[1, d]$. We can also sample the gPINN penalty terms. As mentioned in Appendix I.4.1, we have

$$\mathbf{y} = \mathrm{d}^7 u(\mathbf{x}, \mathbf{0}, \mathbf{e}_i, \mathbf{e}_j, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}), \quad \frac{\partial}{\partial x_i^2 \partial x_j} u(\mathbf{x}) = \mathbf{y}^{[7]}/105. \tag{113}$$

We further have

$$\frac{\partial^2}{\partial x_i^2} u(\mathbf{x}) = \mathbf{y}^{[4]}/3, \tag{114}$$

So the STDE of the Laplacian operator can be computed together with the above pushforward. With this pushforward, we can efficiently amortize the gPINN regularization loss by minimizing the following upperbound on the original gPINN loss with randomized Laplacian

$$\begin{aligned} &\tilde{\ell}_{\text{gPINN}}\left(\{\mathbf{x}^{(i)}\}_{i=1}^{N_r}, I, J\right) \\ =&\frac{1}{N_r} \sum_{j \in J} \sum_{i \in I} \left| \frac{\partial^3}{\partial x_j \partial x_i^2} u(\mathbf{x}) + \frac{\partial}{\partial x_j} u(\mathbf{x}) - 3u^2(\mathbf{x}) \frac{\partial}{\partial x_j} u(\mathbf{x}) - \frac{\partial}{\partial x_j} f(\mathbf{x}) \right|^2 \\ \geq&\frac{1}{N_r} \sum_{j \in J} \left| \sum_{i \in I} \frac{\partial^3}{\partial x_j \partial x_i^2} u(\mathbf{x}) + \frac{\partial}{\partial x_j} u(\mathbf{x}) - 3u^2(\mathbf{x}) \frac{\partial}{\partial x_j} u(\mathbf{x}) - \frac{\partial}{\partial x_j} f(\mathbf{x}) \right|^2, \end{aligned} \tag{115}$$

where J is an independently sampled index set for sampling the gPINN terms. The total loss is

$$\tilde{\ell}_{\text{residual}}(\theta; \{\mathbf{x}^{(i)}\}_{i=1}^{N_r}, I) + \tilde{\ell}_{\text{gPINN}}\left(\{\mathbf{x}^{(i)}\}_{i=1}^{N_r}, I, J\right). \tag{116}$$

We call this technique **amortized gPINN**. The above formula applies to all PDEs where the differential operator is the Laplacian. For example, for the Sine-Gordon equation, we have

$$\begin{aligned} &\tilde{\ell}_{\text{gPINN}}\left(\{\mathbf{x}^{(i)}\}_{i=1}^{N_r}, I, J\right) \\ =&\frac{1}{N_r} \sum_{j \in J} \sum_{i \in I} \left| \frac{\partial^3}{\partial x_j \partial x_i^2} u(\mathbf{x}) + \cos u(\mathbf{x}) \frac{\partial}{\partial x_j} u(\mathbf{x}) - \frac{\partial}{\partial x_j} f(\mathbf{x}) \right|^2. \end{aligned} \tag{117}$$

We use $c_{\text{gPINN}} = 0.1$, and to get better convergence, we train for $20K$ steps instead of $10K$ steps as in all other experiments in this paper. The results are reported in Table 13. We implement the baseline method based on the best-performing first-order AD scheme, the parallelized backward mode SDGD, which we denote as JVP-HVP in the table. Specifically, to compute the residual gradient, we apply one more JVP to the HVP-based implementation of Laplacian (Appendix A.2). From the table, we see that STDE-based amortized gPINN performs better than the JVP-HVP implementation, and both are more efficient than applying backward mode AD in a for-loop. Furthermore, through amortization, we can apply gPINN to high-dimensional PDEs that were intractable.

Appendix J. Estimating the Biharmonic Operator

It was shown in (Hu et al., 2024a) that the Biharmonic operator

$$\Delta^2 u(\mathbf{x}) = \sum_{i=1}^{d} \sum_{j=1}^{d} \frac{\partial^4}{\partial x_i^2 \partial x_j^2} u(\mathbf{x}) \tag{118}$$

Equation	gPINN method	Metric	100 D	1K D	10K D	100K D
Allen-Cahn	JVP-HVP	Speed	256.75it/s	249.48it/s	108.80it/s	61.04it/s
		Error	3.97E-02 ±3.98E-04	1.02E-03 ±6.89E-05	3.08E-04 ±7.48E-06	1.39E-03 ±1.42E-05
	STDE	Speed	**366.46it/s**	**324.60it/s**	**207.85it/s**	**155.40it/s**
		Error	4.34E-02 ±3.72E-04	5.26E-04 ±2.26E-05	1.25E-03 ±4.07E-05	7.61E-04 ±1.03E-04
	None	Error	4.98E-02 ±3.82E-04	6.32E-03 ±4.43E-05	1.19E-04 ±1.04E-05	5.43E-04 ±4.30E-06
Sine-Gordon	JVP-HVP	Speed	1008.65it/s	788.10it/s	413.32it/s	107.68it/s
		Error	1.85E-03 ±4.61E-05	1.02E-03 ±6.89E-05	1.79E-04 ±1.06E-05	5.76E-04 ±1.37E-04
	STDE	Speed	**1165.35it/s**	**948.99it/s**	**542.36it/s**	**210.75it/s**
		Error	6.69E-03 ±1.48E-04	1.12E-03 ±1.38E-05	1.76E-04 ±5.31E-06	1.55E-03 ±4.30E-05
	None	Error	4.74E-03 ±6.68E-05	7.02E-04 ±1.69E-05	1.31E-04 ±1.22E-05	8.07E-04 ±4.01E-06

Table 13: Performance comparison of STDE-gPINN for high-dimensional inseparable PDEs. "None" in the "gPINN method" column indicates that no gPINN loss was used.

has the following unbiased estimator:

$$\Delta^2 u(\mathbf{x}) = \frac{1}{3}\mathbb{E}_{\mathbf{v}\sim p(\mathbf{v})}\left[\partial^4 u(\mathbf{x})(\mathbf{v},\mathbf{0},\mathbf{0},\mathbf{0})\right] \tag{119}$$

where p is the d-dimensional normal distribution. Therefore, its STDE estimator is

$$\tilde{\Delta^2}_N u(\mathbf{x}) = \frac{d}{3N}\sum_{j=1}^{N}\partial^4 u(\mathbf{x})(\mathbf{v}_j,\mathbf{0},\mathbf{0},\mathbf{0}), \quad \mathbf{v}_j\sim\mathcal{N}(\mathbf{0},\mathbf{I}) \tag{120}$$

Appendix K. Why Gaussian Dense STDE Is Not Generalizable

Specifically, we will prove that it is impossible to construct a dense STDE for the fourth-order diagonal operator $\mathcal{L}u=\sum_{i=1}^d \frac{\partial^4 u}{\partial x_i^4}$. The coefficient tensor of $\mathcal{L}$ is the rank-4 identity tensor $\mathbf{I}_4\in\mathbb{R}^{d\times d\times d\times d}$, so the condition for unbiasedness is

$$\mathbb{E}_{\mathbf{v}\sim p}[v_i^{(a)}v_j^{(b)}v_k^{(c)}v_l^{(d)}] = M_{ijkl} = \delta_{ijkl}, \quad a,b,c,d\in\{1,2,3,4\} \tag{121}$$

where $\delta_{ijkl}=1$ when $i=j=k=l$, and is 0 otherwise.

In the most general case where $a\neq b\neq c\neq d$, we can sample $\mathbf{v}\in\mathbb{R}^{4d}$ and split it into four $\mathbb{R}^d$ vectors. In this case we can define blocks of covariance as $\mathbb{E}_{\mathbf{v}\sim p}[\mathbf{v}^{(a)}\mathbf{v}^{(b)}]=\mathbf{\Sigma}^{ab}$, and $\mathbf{\Sigma}=\left[\mathbf{\Sigma}^{ab}\right]_{ab}$. Denote the fourth-moment tensor of p as μ_{ijkl}, then Eq. 121 states that the block $\boldsymbol{\mu}^{abcd}$ in the fourth moment tensor should match $\mathbf{C}$. Because p is assumed to be centered Gaussian, Isserlis' theorem gives the fourth-moment decomposition

$$M_{ijkl} = \mu_{ijkl}^{abcd} = \Sigma_{ij}^{ab}\Sigma_{kl}^{cd} + \Sigma_{ik}^{ac}\Sigma_{jl}^{bd} + \Sigma_{il}^{ad}\Sigma_{jk}^{bc} \tag{122}$$

So finding the p that satisfies Eq. 121 is equivalent to finding a zero-mean Gaussian distribution p with covariance that satisfies the above equation. In the case of $\mathcal{L}$, the coefficient tensor is block-diagonal: $M_{ijkl} = \sigma_{ij}\delta_{ij,kl}$. So in the case where $a \neq b$, set $a = 1, b = 2$, we have

$$\sigma_{ij} = \mu_{ijij}^{1212} = \Sigma_{ii}^{11}\Sigma_{jj}^{22} + 2(\Sigma_{ij}^{12})^2 \tag{123}$$

and $\mathbf{\Sigma} = \begin{bmatrix} \mathbf{\Sigma}^{11} & \mathbf{\Sigma}^{12} \\ \mathbf{\Sigma}^{21} & \mathbf{\Sigma}^{22} \end{bmatrix} \in \mathbb{R}^{2d \times 2d}$. Firstly, consider the diagonal entries of σ:

$$\sigma_{ii} = \mu_{iiii}^{aaaa} = 3(\Sigma_{ii}^{aa})^2, \quad a \in \{1, 2\} \tag{124}$$

This can always be satisfied by setting the diagonal entries of the covariance blocks $\mathbf{\Sigma}^{11}$ and $\mathbf{\Sigma}^{22}$ as follows:

$$\Sigma_{ii}^{aa} = \sqrt{\sigma_{ii}/3}, \quad a \in \{1, 2\} \tag{125}$$

Next, consider the entire σ matrix. We have

$$\sigma_{ij} = \mu_{ijij}^{1212} = \Sigma_{ii}^{11}\Sigma_{jj}^{22} + 2(\Sigma_{ij}^{12})^2 = \frac{1}{3}\sqrt{\sigma_{ii}\sigma_{jj}} + 2(\Sigma_{ij}^{12})^2 \tag{126}$$

In the case of $\mathcal{L}$, we have $\sigma_{ij} = \delta_{ij}$, so for $i \neq j$ we have

$$0 = \frac{1}{3} + 2(\Sigma_{ij}^{12})^2 \tag{127}$$

which is impossible to satisfy since entries in a covariance matrix must be real. Therefore, no centered Gaussian dense-STDE construction can match the required fourth-moment tensor for this operator.

Appendix L. Sparse vs. Dense STDE

The variance of the sparse STDE estimator comes from the variance of selected derivative tensor elements, whereas the variance of the dense estimator comes from the derivative tensor elements that are not selected. For example, in the case of Laplacian, as also discussed in (Hu et al., 2024a), the variance of the sparse STDE estimator comes from the diagonal element of the Hessian, whereas the variance of the dense STDE estimator comes from all the off-diagonal elements of the Hessian.

Appendix M. Further Ablation Study

We ran all three equations from the Inseparable and effectively high-dimensional PDEs (Appendix I.1) with moderately high dimensions (100k), with different randomization batch sizes (1, 4, 16, 64, 100, 256). And we compared their final L2 error and convergence time. The result is shown in Figure 6. As expected, with a smaller batch size, the iterations per second are higher (third row of the figure). Memory cost remains roughly the same since the computation graph was not changed. Rather surprisingly, the final L2 error and the time for convergence did not exhibit a linear relationship to randomization batch size, as can be seen in the first row and last row of the figure. Specifically, one can see that the batch size of 1 provides good L2 error and convergence time, regardless of the equation chosen.

One explanation is that stochastic optimizers like Adam already have built-in mechanisms to control the variance and are robust to noise during training, so smaller batch sizes can perform well. This warrants further investigation that is out of the scope of this paper.

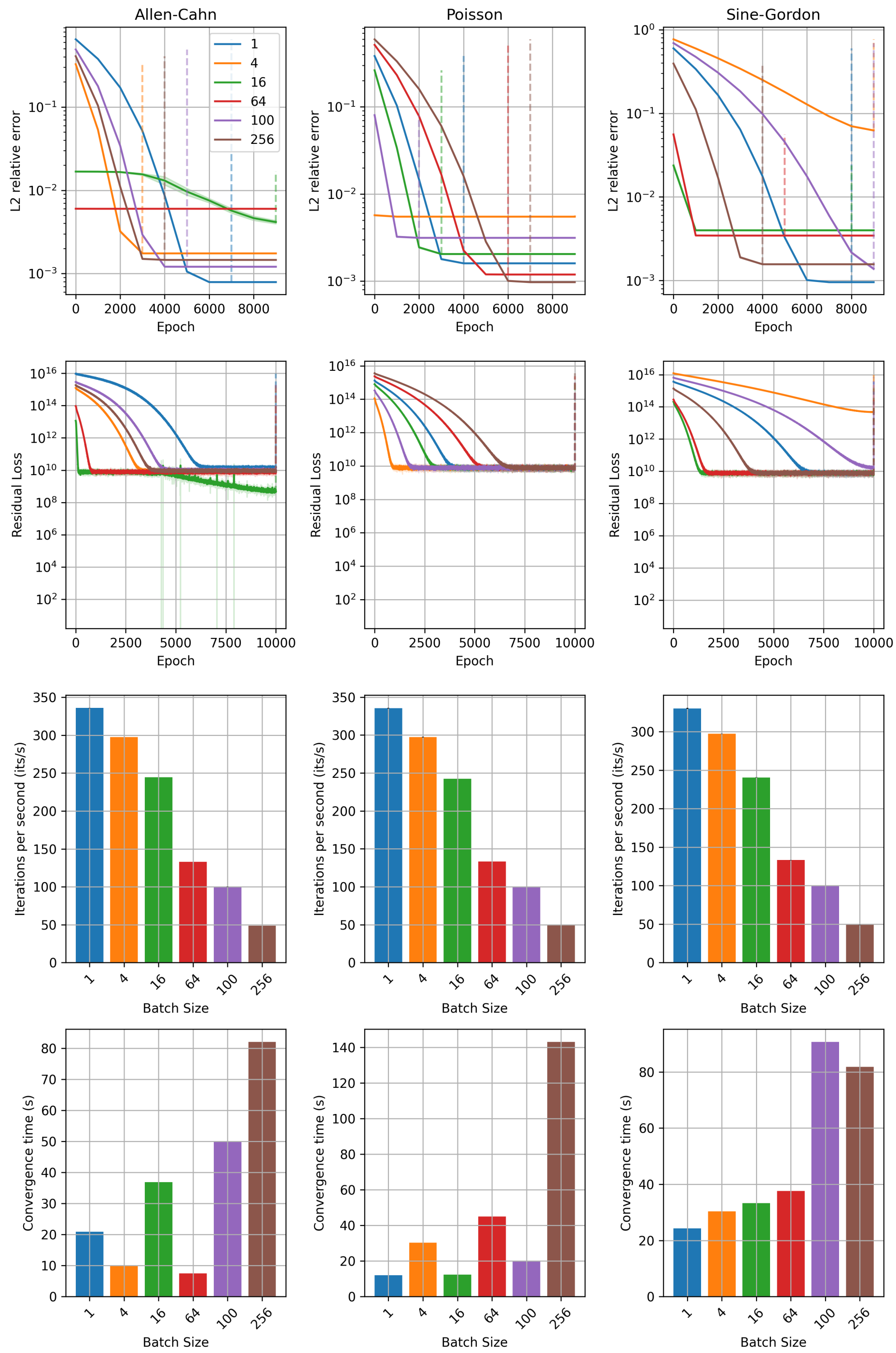

Figure 6: Ablation on randomization batch size with *Inseparable and effectively high-dimensional PDEs*, dim=100k, 5 runs with different random seeds. Model converges when the difference of L2 error is below 1e-7.

References

Brandon Amos. Tutorial on amortized optimization, April 2023. URL http://arxiv.org/abs/2202.00665. arXiv:2202.00665 [cs, math].

Atılım Güneş Baydin, Barak A. Pearlmutter, Don Syme, Frank Wood, and Philip Torr. Gradients without Backpropagation, February 2022. URL http://arxiv.org/abs/2202.08587. arXiv:2202.08587 [cs, stat].

Christian Beck, Sebastian Becker, Patrick Cheridito, Arnulf Jentzen, and Ariel Neufeld. Deep splitting method for parabolic PDEs. *SIAM Journal on Scientific Computing*, 43 (5):A3135–A3154, January 2021. ISSN 1064-8275, 1095-7197. doi: 10.1137/19M1297919. URL http://arxiv.org/abs/1907.03452. arXiv:1907.03452 [cs, math, stat].

Sebastian Becker, Ramon Braunwarth, Martin Hutzenthaler, Arnulf Jentzen, and Philippe von Wurstemberger. Numerical simulations for full history recursive multilevel Picard approximations for systems of high-dimensional partial differential equations. *Communications in Computational Physics*, 28(5):2109–2138, June 2020. ISSN 1815-2406, 1991-7120. doi: 10.4208/cicp.OA-2020-0130. URL http://arxiv.org/abs/2005.10206. arXiv:2005.10206 [cs, math].

Claus Bendtsen and Ole Stauning. Tadiff , a flexible c + + package for automatic differentiation using taylor series expansion. 1997. URL https://api.semanticscholar.org/CorpusID:62828904.

Jesse Bettencourt, Matthew J. Johnson, and David Duvenaud. Taylor-mode automatic differentiation for higher-order derivatives in JAX. In *Program Transformations for ML Workshop at NeurIPS 2019*, 2019. URL https://openreview.net/forum?id=SkxEF3FNPH.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Benyamin Ghojogh, Ali Ghodsi, Fakhri Karray, and Mark Crowley. Johnson-Lindenstrauss Lemma, Linear and Nonlinear Random Projections, Random Fourier Features, and Random Kitchen Sinks: Tutorial and Survey, August 2021. URL http://arxiv.org/abs/2108.04172. arXiv:2108.04172 [cs, math, stat].

Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, second edition, 2008. doi: 10.1137/1.9780898717761. URL https://epubs.siam.org/doi/abs/10.1137/1.9780898717761.

Andreas Griewank, Jean Utke, and Andrea Walther. Evaluating higher derivative tensors by forward propagation of univariate taylor series. *Mathematics of Computation*, 69(231): 1117–1131, February 2000. ISSN 0025-5718. doi: 10.1090/s0025-5718-00-01120-0. URL http://dx.doi.org/10.1090/s0025-5718-00-01120-0.

Jiequn Han, Arnulf Jentzen, and Weinan E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115 (34):8505–8510, Aug 2018. ISSN 1091-6490. doi: 10.1073/pnas.1718942115. URL http://dx.doi.org/10.1073/pnas.1718942115.

Di He, Shanda Li, Wenlei Shi, Xiaotian Gao, Jia Zhang, Jiang Bian, Liwei Wang, and Tie-Yan Liu. Learning Physics-Informed Neural Networks without Stacked Back-propagation, February 2023. URL http://arxiv.org/abs/2202.09340. arXiv:2202.09340 [cs].

Zheyuan Hu, Zhouhao Yang, Yezhen Wang, George Em Karniadakis, and Kenji Kawaguchi. Bias-Variance Trade-off in Physics-Informed Neural Networks with Randomized Smoothing for High-Dimensional PDEs, November 2023. URL http://arxiv.org/abs/2311.15283. arXiv:2311.15283 [cs, math, stat].

Zheyuan Hu, Zekun Shi, George Em Karniadakis, and Kenji Kawaguchi. Hutchinson Trace Estimation for High-Dimensional and High-Order Physics-Informed Neural Networks. *Computer Methods in Applied Mechanics and Engineering*, 424:116883, May 2024a. ISSN 00457825. doi: 10.1016/j.cma.2024.116883. URL http://arxiv.org/abs/2312.14499. arXiv:2312.14499 [cs, math, stat].

Zheyuan Hu, Khemraj Shukla, George Em Karniadakis, and Kenji Kawaguchi. Tackling the curse of dimensionality with physics-informed neural networks. *Neural Networks*, 176: 106369, 2024b.

Zheyuan Hu, Zhongqiang Zhang, George Em Karniadakis, and Kenji Kawaguchi. Score-Based Physics-Informed Neural Networks for High-Dimensional Fokker-Planck Equations, February 2024c. URL http://arxiv.org/abs/2402.07465. arXiv:2402.07465 [cs, math, stat].

M.F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics - Simulation and Computation*, 18(3): 1059–1076, January 1989. ISSN 1532-4141. doi: 10.1080/03610918908812806. URL http://dx.doi.org/10.1080/03610918908812806.

Martin Hutzenthaler, Arnulf Jentzen, Thomas Kruse, Tuan Anh Nguyen, and Philippe von Wurstemberger. Overcoming the curse of dimensionality in the numerical approximation of semilinear parabolic partial differential equations, July 2018. URL https://arxiv.org/abs/1807.01212v3.

Jerzy Karczmarczuk. Functional differentiation of computer programs. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 195–203, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130244. doi: 10.1145/289423.289442. URL https://doi.org/10.1145/289423.289442.

George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, Jun 2021. ISSN 2522-5820. doi: 10.1038/s42254-021-00314-5. URL https://doi.org/10.1038/s42254-021-00314-5.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Chieh-Hsin Lai, Yuhta Takida, Naoki Murata, Toshimitsu Uesaka, Yuki Mitsufuji, and Stefano Ermon. Regularizing score-based models with score fokker-planck equations. In *NeurIPS 2022 Workshop on Score-Based Methods*, 2022.

Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. A general construction for abstract interpretation of higher-order automatic differentiation. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi: 10.1145/3563324. URL https://doi-org.libproxy1.nus.edu.sg/10.1145/3563324.

Ruichen Li, Haotian Ye, Du Jiang, Xuelan Wen, Chuwei Wang, Zhe Li, Xiang Li, Di He, Ji Chen, Weiluo Ren, and Liwei Wang. Forward Laplacian: A New Computational Framework for Neural Network-based Variational Monte Carlo, July 2023. URL http://arxiv.org/abs/2307.08214. arXiv:2307.08214 [physics].

Ruichen Li, Chuwei Wang, Haotian Ye, Di He, and Liwei Wang. DOF: Accelerating high-order differential operators with forward propagation. In *ICLR 2024 Workshop on AI4DifferentialEquations In Science*, 2024. URL https://openreview.net/forum?id=yQsLKpkRoS.

Sijia Liu, Pin-Yu Chen, Bhavya Kailkhura, Gaoyuan Zhang, Alfred Hero, and Pramod K. Varshney. A Primer on Zeroth-Order Optimization in Signal Processing and Machine Learning, June 2020. URL http://arxiv.org/abs/2006.06224. arXiv:2006.06224 [cs, eess, stat].

Lu Lu, Raphaël Pestourie, Wenjie Yao, Zhicheng Wang, Francesc Verdugo, and Steven G. Johnson. Physics-informed neural networks with hard constraints for inverse design. *SIAM Journal on Scientific Computing*, 43(6):B1105–B1132, 2021. doi: 10.1137/21M1397908. URL https://doi.org/10.1137/21M1397908.

Sadhika Malladi, Tianyu Gao, Eshaan Nichani, Alex Damian, Jason D. Lee, Danqi Chen, and Sanjeev Arora. Fine-Tuning Language Models with Just Forward Passes, January 2024. URL http://arxiv.org/abs/2305.17333. arXiv:2305.17333 [cs].

Per-Gunnar Martinsson and Joel Tropp. Randomized Numerical Linear Algebra: Foundations & Algorithms, March 2021. URL http://arxiv.org/abs/2002.01387. arXiv:2002.01387 [cs, math].

Riley Murray, James Demmel, Michael W. Mahoney, N. Benjamin Erichson, Maksim Melnichenko, Osman Asif Malik, Laura Grigori, Piotr Luszczek, Michał Dereziński, Miles E. Lopes, Tianyu Liang, Hengrui Luo, and Jack Dongarra. Randomized Numerical Linear Algebra : A Perspective on the Field With an Eye to Software, April 2023. URL http://arxiv.org/abs/2302.11474. arXiv:2302.11474 [cs, math].

Deniz Oktay, Nick McGreivy, Joshua Aduol, Alex Beatson, and Ryan P. Adams. Randomized Automatic Differentiation, March 2021. URL http://arxiv.org/abs/2007.10412. arXiv:2007.10412 [cs, stat].

Tianyu Pang, Kun Xu, Chongxuan Li, Yang Song, Stefano Ermon, and Jun Zhu. Efficient Learning of Generative Models via Finite-Difference Score Matching, November 2020. URL http://arxiv.org/abs/2007.03317. arXiv:2007.03317 [cs, stat].

Juncai Pu and Yong Chen. Lax pairs informed neural networks solving integrable systems, January 2024. URL http://arxiv.org/abs/2401.04982. arXiv:2401.04982 [nlin].

M. Raissi, P. Perdikaris, and G.E. Karniadakis. Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, February 2019. ISSN 0021-9991. doi: 10.1016/j.jcp.2018.10.045. URL http://dx.doi.org/10.1016/j.jcp.2018.10.045.

Maziar Raissi. Forward-Backward Stochastic Neural Networks: Deep Learning of High-dimensional Partial Differential Equations, April 2018. URL http://arxiv.org/abs/1804.07010. arXiv:1804.07010 [cs, math, stat].

Justin Sirignano and Konstantinos Spiliopoulos. Dgm: a deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

Yang Song, Sahaj Garg, Jiaxin Shi, and Stefano Ermon. Sliced Score Matching: A Scalable Approach to Density and Score Estimation, June 2019. URL http://arxiv.org/abs/1905.07088. arXiv:1905.07088 [cs, stat].

Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-Based Generative Modeling through Stochastic Differential Equations, February 2021. URL http://arxiv.org/abs/2011.13456. arXiv:2011.13456 [cs, stat].

Charles M. Stein. Estimation of the Mean of a Multivariate Normal Distribution. *The Annals of Statistics*, 9(6):1135 – 1151, 1981. doi: 10.1214/aos/1176345632. URL https://doi.org/10.1214/aos/1176345632.

Matthias Troyer and Uwe-Jens Wiese. Computational complexity and fundamental limitations to fermionic quantum Monte Carlo simulations, August 2004. URL https://arxiv.org/abs/cond-mat/0408370v1.

Mu Wang. *High Order Reverse Mode of Automatic Differentiation.* PhD thesis, 2017. URL http://libproxy1.nus.edu.sg/login?url=https://www.proquest.com/dissertations-theses/high-order-reverse-mode-automatic-differentiation/docview/1975367062/se-2. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-03-04.

E Weinan and Ting Yu. The deep ritz method: a deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6:1 – 12, 2017. URL https://api.semanticscholar.org/CorpusID:2988078.

Jeremy Yu, Lu Lu, Xuhui Meng, and George Em Karniadakis. Gradient-enhanced physics-informed neural networks for forward and inverse PDE problems. *Computer Methods in Applied Mechanics and Engineering*, 393:114823, April 2022. ISSN 00457825. doi: 10.1016/j.cma.2022.114823. URL `http://arxiv.org/abs/2111.02801`. arXiv:2111.02801 [physics].

Yaohua Zang, Gang Bao, Xiaojing Ye, and Haomin Zhou. Weak Adversarial Networks for High-dimensional Partial Differential Equations. *Journal of Computational Physics*, 411:109409, June 2020. ISSN 00219991. doi: 10.1016/j.jcp.2020.109409. URL `http://arxiv.org/abs/1907.08272`. arXiv:1907.08272 [cs, math].