EFFICIENT NEURAL OPERATOR DECOMPOSITION VIA TENSOR PRODUCT QUADRATURE

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent studies have shown that the operator eigenvalue/singular value decomposition (EVD/SVD) can be efficiently achieved via neural network-based optimization; however, the function quadrature within the objective is computationally intensive, especially in high dimensions. Thus, mini-batch Monte Carlo sampling is used to obtain unbiased quadrature estimates, yet higher accuracy relies on larger batch size, imposing a computation-accuracy trade-off that restricts the efficiency of current methods in $N \geq 2$ dimensions. We propose a tensor product quadrature (TPQ) for operator decomposition, where each spectral function is parameterized by a variable-separable neural network, and the N-D quadrature is decomposed into high-accuracy and low-cost 1D quadratures followed by tensor product assembly, which largely bypasses the computation-accuracy trade-off. We further develop TPQ-compatible treatments for sampling, gradient mask, boundary constraints, and non-separable potentials using explicit decomposition or parametric pre-training. Numerical experiments on the Schrödinger equation (2D harmonic oscillator, 2D infinite well, 2D and 3D hydrogen atom), neural tangent kernel (NTK) decomposition, and up to 5D integral kernel operator SVD validate the TPQ method. Notably, the TPQ consistently improves the efficacy of neural operator EVD/SVD baselines, including spectral inference network (SpIN), neural eigenfunctions (NeuralEF), and nested low-rank approximation (NeuralSVD).

1 INTRODUCTION

The spectral decomposition of a linear operator $\mathcal{T}$ (such as the integral kernel operator $(\mathcal{T}f)(\boldsymbol{x}) = \int k(\boldsymbol{x}, \boldsymbol{x}')f(\boldsymbol{x}')d\boldsymbol{x}'$) aims to find the eigen-pairs $\{\lambda_l, u_l(\boldsymbol{x})\}$ or singular-triplets $\{\sigma_l, u_l(\boldsymbol{x}), v_l(\boldsymbol{x})\}$ such that $\mathcal{T}$ satisfies the eigenvalue decomposition (EVD) $\mathcal{T}f = \sum_{l=1}^{\infty} \lambda_l \langle u_l, f\rangle u_l$ or the singular value decomposition (SVD) $\mathcal{T}f = \sum_{l=1}^{\infty} \sigma_l \langle u_l, f\rangle v_l$ (for non-self-adjoint $\mathcal{T}$). It has broad applications (Belkin & Niyogi, 2003; Gottlieb & Orszag, 1977; Ryu et al., 2025), such as feature extraction in machine learning (Schölkopf & Smola, 2005; Deng et al., 2022), resolving ground and excited states via Schrödinger equation in quantum mechanics (Pfau et al., 2019; 2020; Hermann et al., 2020), and modeling dynamical systems via Koopman operator SVD (Jeong et al., 2025). Classical non-parametric spectral decomposition approaches, such as the Nyström method (Williams & Seeger, 2000), rely on eigenvector interpolation over discrete support points, thus the required sampling density explodes exponentially in high-dimensions to achieve desired interpolation accuracy, making them computationally inefficient in high dimensions (Ryu et al., 2024).

Recent neural operator decomposition methods break the dimensional bottleneck by using neural networks to parameterize eigenfunctions, allowing continuous generalization to unseen points (E & Yu, 2018). For instance, Pfau et al. (Pfau et al., 2019) introduced the spectral inference network (SpIN), which formulates the EVD as an optimization problem to learn ordered eigenfunctions. Given a linear self-adjoint operator $\mathcal{T}$ and neural network ansatz $u_\theta(\boldsymbol{x}) : \Omega \to \mathbb{R}^L$ (where $\Omega \subset \mathbb{R}^N$, and to obtain L eigenvalues), SpIN seeks to maximize the generalized Rayleigh trace:

$$\text{(SpIN objective)} \quad \max_{\theta} \operatorname{Tr}(\Sigma^{-1}\Pi),\ \Sigma = \mathbb{E}_{\boldsymbol{x}}[u_\theta(\boldsymbol{x})u_\theta(\boldsymbol{x})^T],\ \Pi = \mathbb{E}_{\boldsymbol{x}}[u_\theta(\boldsymbol{x})(\mathcal{T}u_\theta)(\boldsymbol{x})^T]. \tag{1}$$

The eigenfunction integrals over the spatial domain $\boldsymbol{x} \in \Omega$ are estimated via mini-batch Monte Carlo (MC) sampling. To enforce orthogonality, SpIN computes Cholesky decomposition $\Sigma = \boldsymbol{L}\boldsymbol{L}^T$ at each iteration. By applying gradient mask to the transformed objective $\operatorname{Tr}(\boldsymbol{L}^{-1}\Pi\boldsymbol{L}^{-T})$ via Jacobian

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

computation, SpIN isolates the gradients such that the l-th eigenfunction is learned independently of lower-order ones, hence recovering the true ordered orthogonal eigenfunctions via $\boldsymbol{L}^{-1}u_\theta(\cdot)$.

Deng et al. (Deng et al., 2022) proposed the neural eigenfunctions (NeuralEF), which generalizes the EigenGame (Gemp et al., 2021) to function space to deconstruct kernel operators by using L independent neural networks $\{u_{\theta_l}(\boldsymbol{x}) : \Omega \to \mathbb{R}\}_{l=1}^L$ to parameterize eigenfunctions. Let $\Sigma_{ll} = \mathbb{E}_{\boldsymbol{x}}[u_{\theta_l}(\boldsymbol{x})^2]$ and $\Pi_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_{\theta_l}(\boldsymbol{x})(\mathcal{T}u_{\theta_m})(\boldsymbol{x})]$. NeuralEF optimizes the sequential objective

$$\text{(NeuralEF objective)} \quad \max_{\{\theta_l\}_{l=1}^L} \sum_{l=1}^{L} \Big(\Pi_{ll} - \sum_{m<l} \Pi_{lm}^2/\Pi_{mm}\Big), \text{ s.t. } \Sigma_{ll} = 1, \ \forall l = 1, \cdots, L. \tag{2}$$

The objective involves computing the quadratic function integrals defined by the kernel, which are approximated by mini-batch MC (Deng et al., 2022). NeuralEF bypasses the costly Cholesky orthogonalization and Jacobian required in SpIN by using the soft penalty $\sum_{m<l} \Pi_{lm}^2/\Pi_{mm}$.

To extend to the SVD of non-self-adjoint operators, Ryu et al. (Ryu et al., 2024) proposed the nested low-rank approximation (NeuralSVD) to learn ordered singular functions based on Schmidt's theorem (Schmidt, 1907). NeuralSVD parameterizes the right and left singular functions using two sequences of independent neural networks[1] $\{u_{\theta_l}(\boldsymbol{x})\}_{l=1}^L$ and $\{v_{\phi_l}(\boldsymbol{x})\}_{l=1}^L$. Let $\Sigma_{lm}^u = \mathbb{E}_{\boldsymbol{x}}[u_{\theta_l}(\boldsymbol{x})u_{\theta_m}(\boldsymbol{x})]$, $\Sigma_{lm}^v = \mathbb{E}_{\boldsymbol{x}}[v_{\phi_l}(\boldsymbol{x})v_{\phi_m}(\boldsymbol{x})]$, and the operator $\mathcal{T}$ quadratic $\Pi_{ll} = \mathbb{E}_{\boldsymbol{x}}[v_{\phi_l}(\boldsymbol{x})(\mathcal{T}u_{\theta_l})(\boldsymbol{x})]$. NeuralSVD minimizes the low-rank approximation (LoRA) objective:

$$\text{(NeuralSVD objective)} \quad \min_{\{\theta_l,\phi_l\}_{l=1}^L} \mathcal{L}_{\text{LoRA}}(\{\theta_l,\phi_l\}_{l=1}^L) := \min_{\{\theta_l,\phi_l\}_{l=1}^L} -2\sum_{l=1}^{L}\Pi_{ll} + \sum_{l=1}^{L}\sum_{m=1}^{L}\Sigma_{lm}^u\Sigma_{lm}^v. \tag{3}$$

To learn singular functions in the correct order, NeuralSVD employs a "nesting" technique by sequentially optimizing the l-th singular-function via $\min \mathcal{L}_{\text{LoRA}}(\{\theta_m,\phi_m\}_{m=1}^l)$, treating the previously learned $l-1$ functions as optimum. This can be equivalently implemented by optimizing a single joint objective with a weighted sum of nested objectives $\sum_{l=1}^L w_l\mathcal{L}_{\text{LoRA}}(\{\theta_m,\phi_m\}_{m=1}^l)$, where positive weights $w_l > 0$ ensure learning ordered singular functions (Ryu et al., 2024). During training, the function expectations are approximated by unbiased mini-batch MC sampling.

Our contributions Despite their effectiveness, evaluating the N-dimensional (N-D) function integrals (for an N-D problem $\Omega \subset \mathbb{R}^N$) within these optimization objectives remains computationally non-trivial in current methods (Pfau et al., 2019; Deng et al., 2022; Ryu et al., 2024). Applying dense grid numerical quadrature in high dimensions requires an exponentially growing number of neural network evaluations; thus, neural operator decomposition methods rely on mini-batch MC sampling (Pfau et al., 2019; Deng et al., 2022; Ryu et al., 2024) for unbiased integral estimation. The numerical error of MC sampling for numerical integration is bounded by $\mathcal{O}(\sigma/\sqrt{B})$, where B is the batch size and σ^2 is the variance of a single function evaluation (detailed in Appendix Sec. D.3). Since σ^2 can grow exponentially with N for certain high-dimensional oscillatory functions (Sec. D.4), maintaining a prescribed integration accuracy requires a substantially larger batch size. This imposes a computation-accuracy trade-off that restricts the efficiency of previous methods[2].

To alleviate the computation-accuracy trade-off, we propose tensor product quadrature (TPQ) for operator EVD/SVD. We parameterize the l-th spectral function ($l = 1, \cdots, L$) with a variable-separable network in canonical polyadic (CP) tensor format (Cho et al., 2023; Luo & Meng, 2026):

$$u_l(\boldsymbol{x}) = \sum_{r=1}^{R}(u_{\theta_{1,l}}(x_1))_r(u_{\theta_{2,l}}(x_2))_r\cdots(u_{\theta_{N,l}}(x_N))_r : \mathbb{R}^N \to \mathbb{R}, \tag{4}$$

where the spatial coordinate $\boldsymbol{x} = (x_1, \cdots, x_N)$ is decoupled into N independent 1D inputs. Each 1D sub-network $u_{\theta_{d,l}}(x_d) : \mathbb{R} \to \mathbb{R}^R$ processes a scalar input and outputs an R-dimensional vector, with $(u_{\theta_{d,l}}(x_d))_r$ denoting its r-th output element, and R is the preset CP rank (Kolda & Bader, 2009). This variable-separable structure provides two key computational advantages. First, it avoids the exponential explosion of network evaluations. Evaluating $u_l(\boldsymbol{x})$ on an N-D dense grid is reduced to evaluating the independent 1D function values of $\{u_{\theta_{d,l}}(x_d)\}$ over N 1D grids (Cho et al.,

[1] As noted in (Ryu et al., 2024), a shared or "joint" parameterization is also applicable, but we focus on the sequential neural parameterization here for generality.

[2] For instance, NeuralEF and NeuralSVD typically adopt a relatively small $B = 512$ to avoid becoming resource-intensive (Deng et al., 2022; Ryu et al., 2024), which may incur larger error in dimensions $N \geq 2$.

Table 1: Comparison of approaches and computational complexities among representative neural operator decomposition methods and their TPQ schemes (assuming a grid batch size of $B = B_{\text{1D}}^N$). Notations: dimension N, number of target spectral components L, 1D batch size B_{1D}, TPQ tensor rank R, network forward pass complexity F (FLOPs), and network parameter count P.

Method		Original Approach	Complexity	TPQ Approach	Complexity
SpIN (Sec. 2.1)	Objective	Rayleigh trace	$\mathcal{O}(B_{\text{1D}}^N(F+L^2))$	*Rayleigh trace via TPQ*	$\mathcal{O}(B_{\text{1D}}N(F+R^2L^2))$
	Ordering	Jacobian gradient mask	$\mathcal{O}(B_{\text{1D}}^N L^2 P)$	*Separable Jacobian mask*	$\mathcal{O}(B_{\text{1D}}NL^2R^2P)$
	Orthogonal	Cholesky decomposition	$\mathcal{O}(L^3)$	Cholesky decomposition	$\mathcal{O}(L^3)$
NeuralEF (Sec. 2.2)	Objective	Rayleigh quotient	$\mathcal{O}(B_{\text{1D}}^N(FL+L^2))$	*Rayleigh quotient via TPQ*	$\mathcal{O}(B_{\text{1D}}N(FL+R^2L^2))$
	Ordering	Sequential deflation	--	Sequential deflation	--
	Orthogonal	Soft penalty	$\mathcal{O}(B_{\text{1D}}^N L^2)$	*Soft penalty via TPQ*	$\mathcal{O}(B_{\text{1D}}NR^2L^2)$
NeuralSVD (Sec. 2.3)	Objective	Schmidt's LoRA	$\mathcal{O}(B_{\text{1D}}^N(FL+L^2))$	*Schmidt's LoRA via TPQ*	$\mathcal{O}(B_{\text{1D}}N(FL+R^2L^2))$
	Ordering	Nested optimization	--	Nested optimization	--
	Orthogonal	Soft penalty	$\mathcal{O}(B_{\text{1D}}^N L^2)$	*Soft penalty via TPQ*	$\mathcal{O}(B_{\text{1D}}NR^2L^2)$

2023; Luo & Meng, 2026), which drops the evaluation complexity from $\mathcal{O}(B_{\text{1D}}^N)$ to $\mathcal{O}(B_{\text{1D}}N)$, where B_{1D} is the number of grid points per dimension. Second, the large N-D integral over $u_l(\boldsymbol{x})$ can be mathematically factorized into a product of 1D integrals over $\{u_{\theta_{d,l}}(x_d)\}$, enabling the use of accurate and low-cost 1D quadrature along each axis. For instance, setting the batch size per dimension as B_{1D} (e.g., 512) yields a massive effective batch size of B_{1D}^N for an N-D problem. Consequently, TPQ alleviates the computation-accuracy trade-off in previous methods by allowing the computational cost to scale linearly with N (Table 1).

From a numerical integration perspective, TPQ reuses B_{1D} samples along each dimension to implicitly cover B_{1D}^N sample combinations, rather than evaluating B_{1D}^N full-dimensional samples independently. In the product and oscillatory regime relevant to higher-order spectral functions, the resulting estimator achieves the same leading-order variance as direct MC with batch size B_{1D}^N, while requiring only $B_{\text{1D}}N$ 1D factor evaluations (detailed in Sec. D.5). Therefore, at a comparable computational cost, TPQ provides a much larger effective quadrature batch size and thus lower integration noise, leading to more stable and efficient neural spectral decomposition. In summary,

- We propose an efficient tensor product quadrature scheme that largely bypasses the computation-accuracy trade-off in existing neural operator decomposition methods.
- We equip the TPQ scheme with compatible sampling strategy, gradient mask, boundary mask, and parametric potential pre-training for efficient operator decomposition.
- We evaluate the TPQ on diverse baselines (SpIN, NeuralEF, NeuralSVD) and benchmarks, including Schrödinger equations (2D harmonic oscillator, infinite well, 2D & 3D hydrogen atom) and up to 5D integral kernel operator SVD, demonstrating its improved efficacy.

We remark that there are related works that used tensor neural network-based integral methods for obtaining multi-eigenpairs of partial differential equations (Wang et al., 2024; Wang & Xie, 2024). We detail the distinctions between this type of work and our method in Appendix Sec. E. We compare TPQ with traditional non-parametric eigensolvers in Appendix Sec. B.

2 The Proposed TPQ Strategy for Operator Decomposition

Applying TPQ to SpIN, NeuralEF, and NeuralSVD shares a similar mathematical foundation, namely, the factorized assembly of the mass matrix Σ and the stiffness matrix Π using TPQ. Therefore, we will use SpIN-TPQ (Sec. 2.1) as the primary example to illustrate the tensor product quadrature. These techniques readily generalize to NeuralEF-TPQ and NeuralSVD-TPQ (Secs. 2.2-2.3), where we will follow the algorithmic results on SpIN for a more concise presentation.

2.1 SpIN-TPQ: Efficient Rayleigh Trace Assembly and Separable Jacobian

We first demonstrate how TPQ enables SpIN to bypass the computation-accuracy trade-off. The goal is to find the first L eigenfunctions $\{u_l : \mathbb{R}^N \to \mathbb{R}\}_{l=1}^L$ of a linear operator $\mathcal{T}$ (e.g., a Hamiltonian). In our SpIN-TPQ, L eigenfunctions are independently parameterized by L variable-separable neural networks in the CP format equation 4. Specifically, for each dimension index d and eigenindex l, a sub-network $u_{\theta_{d,l}}(x_d)$ outputs a vector of dimension R. The l-th trial eigenfunction

is thus formulated as $u_l(\boldsymbol{x}) = \sum_{r=1}^{R} \prod_{d=1}^{N} (u_{\theta_{d,l}}(x_d))_r$, where R is the CP rank, N is the spatial dimension, and $(u_{\theta_{d,l}}(x_d))_r$ denotes the r-th output element of the (d,l)-th 1D sub-network. The SpIN framework maximizes the Rayleigh trace $\mathrm{Tr}(\Sigma^{-1}\Pi)$, where $\Sigma \in \mathbb{R}^{L\times L}$ represents the mass matrix and $\Pi \in \mathbb{R}^{L\times L}$ is the stiffness matrix defined by $\Sigma_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})u_m(\boldsymbol{x})]$ and $\Pi_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})(\mathcal{T}u_m)(\boldsymbol{x})]$. Hereafter, we consider an N-D grid of B_{1D}^N collocation points to perform the numerical quadrature, where B_{1D} denotes the number of samples per dimension.

2.1.1 Separable Boundary Mask

To obtain boundary-compliant trial functions, we multiply each spectral function $u_l(\boldsymbol{x})$ with a predefined mask $M(\boldsymbol{x})$. A naive choice of $M(\boldsymbol{x})$, however, couples the coordinate directions and destroys the variable-separable structure on which TPQ relies. To resolve this conflict, we parameterize $M(\boldsymbol{x})$ as a separable form by a product of univariate envelopes, $M(\boldsymbol{x}) = \prod_{d=1}^{N} m_d(x_d)$, which can be absorbed directly into the 1D sub-networks without altering the CP factorization. Concretely, applying the separable mask to the network output equation 4 gives

$$M(\boldsymbol{x})\, u_l(\boldsymbol{x}) = \left(\prod_{d=1}^{N} m_d(x_d)\right)\left(\sum_{r=1}^{R}\prod_{d'=1}^{N} (u_{\theta_{d',l}}(x_{d'}))_r\right) = \sum_{r=1}^{R}\prod_{d=1}^{N} m_d(x_d)\,(u_{\theta_{d,l}}(x_d))_r, \tag{5}$$

where each 1D factor $m_d(x_d)\,(u_{\theta_{d,l}}(x_d))_r$ remains a function of a single coordinate. We will assume the mask is incorporated into the ansatz and retain the notation $u_{\theta_{d,l}}(x_d)$ in following. The specific form of $m_d(x_d)$ depends on the boundary condition and is detailed in Sec. F.4.

2.1.2 Mass and Stiffness Matrix Assembly via TPQ

Evaluating the mass Σ and stiffness Π matrices requires computing high-dimensional integrals over $\Omega \subset \mathbb{R}^N$. To calculate the mass $\Sigma_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})u_m(\boldsymbol{x})]$, we substitute the CP ansatz equation 4[3]:

$$\begin{aligned}\Sigma_{lm} &= \mathbb{E}_{\boldsymbol{x}}\left[\left(\sum_{r=1}^{R}\prod_{d=1}^{N}(u_{\theta_{d,l}}(x_d))_r\right)\left(\sum_{s=1}^{R}\prod_{d'=1}^{N}(u_{\theta_{d',m}}(x_{d'}))_s\right)\right] \\ &= \sum_{r=1}^{R}\sum_{s=1}^{R}\prod_{d=1}^{N}\mathbb{E}_{x_d}\left[(u_{\theta_{d,l}}(x_d))_r(u_{\theta_{d,m}}(x_d))_s\right] := \sum_{r=1}^{R}\sum_{s=1}^{R}\prod_{d=1}^{N} M_{rs}^{lm,d}.\end{aligned} \tag{6}$$

It can be seen that the N-D integral Σ_{lm} decomposes into NR^2 1D integrals $M_{rs}^{lm,d}$ by TPQ and independent sampling. Specifically, we approximate the 1D overlap integral $M_{rs}^{lm,d}$ via importance sampling: $M_{rs}^{lm,d} = \frac{1}{B_{\mathrm{1D}}}\sum_{b=1}^{B_{\mathrm{1D}}}(u_{\theta_{d,l}}(x_{d,b}))_r(u_{\theta_{d,m}}(x_{d,b}))_s w_d(x_{d,b})$, where each $x_{d,b}$ is sampled independently from a 1D proposal distribution $x_{d,b} \sim \rho(x_d)$ with importance weight $w_d(x_{d,b}) = 1/\rho(x_{d,b})$. In practice, we draw B_{1D} independent samples for each dimension from a 1D Gaussian distribution. The joint importance weight factorizes as $W(\boldsymbol{x}) = \prod_{d=1}^{N} w_d(x_d)$, which is absorbed into the 1D components. Calculating $M_{rs}^{lm,d}$ and the corresponding Σ_{lm} in equation 6 for all l, m costs $\mathcal{O}(B_{\mathrm{1D}}NR^2L^2)$. Compared to the dense grid integral for calculating Σ which scales with $\mathcal{O}(B_{\mathrm{1D}}^N L^2)$, the TPQ reduces the computational complexity to scale linearly with dimension N. Another dominant cost is the function evaluation of neural network (forward pass). The original integral costs $\mathcal{O}(B_{\mathrm{1D}}^N F)$ for the dense B_{1D}^N grid evaluation, while the TPQ equation 6 solely costs $\mathcal{O}(B_{\mathrm{1D}}NF)$ by evaluating 1D components $u_{\theta_{d,l}}$, where F is the FLOPs per sample.

To calculate the stiffness $\Pi_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})(\mathcal{T}u_m)(\boldsymbol{x})]$, we assume two typical shapes of separability for the linear operator $\mathcal{T}$: additive separability $\mathcal{T} = \sum_{d=1}^{N}\mathcal{T}_d$ (such as Laplacian $\Delta = \sum_{d=1}^{N}\frac{\partial^2}{\partial x_d^2}$) and multiplicative separability $\mathcal{T} = \prod_{d=1}^{N}\mathcal{T}_d$, where $\mathcal{T}_d$ operates on the d-th dimension only. We elaborate on how to transfer non-separable operators (such as potential functions) into separable surrogates via Gaussian mixture parametric pre-training in Sec. 2.1.3.

For additive separable operators $\mathcal{T} = \sum_{d=1}^{N}\mathcal{T}_d$, we have

$$(\mathcal{T}u_m)(\boldsymbol{x}) = \sum_{k=1}^{N}\mathcal{T}_k\left(\sum_{s=1}^{R}\prod_{d=1}^{N}(u_{\theta_{d,m}}(x_d))_s\right) = \sum_{s=1}^{R}\sum_{k=1}^{N}\left((\mathcal{T}_k u_{\theta_{k,m}}(x_k))_s\prod_{d\neq k}(u_{\theta_{d,m}}(x_d))_s\right). \tag{7}$$

[3]For the high-dimensional integral Σ_{lm} we will use independent 1D Gaussian sampling to sample each 1D variable x_d so that the global expectation $\mathbb{E}_{\boldsymbol{x}}$ can be decomposed into products of 1D expectations $\prod_d \mathbb{E}_{x_d}$.

Substituting this into the stiffness Π_{lm} and grouping the expectation yields:

$$\begin{aligned}\Pi_{lm} &= \mathbb{E}_{\boldsymbol{x}}\left[\left(\sum_{r=1}^{R}\prod_{d'=1}^{N}(u_{\theta_{d',l}}(x_{d'}))_r\right)\left(\sum_{s=1}^{R}\sum_{k=1}^{N}(\mathcal{T}_k u_{\theta_{k,m}}(x_k))_s\prod_{d\neq k}(u_{\theta_{d,m}}(x_d))_s\right)\right]\\ &= \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{k=1}^{N}\mathbb{E}_{\boldsymbol{x}}\left[(u_{\theta_{k,l}}(x_k))_r(\mathcal{T}_k u_{\theta_{k,m}}(x_k))_s\prod_{d\neq k}(u_{\theta_{d,l}}(x_d))_r(u_{\theta_{d,m}}(x_d))_s\right]\\ &= \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{k=1}^{N}\underbrace{\mathbb{E}_{x_k}\left[(u_{\theta_{k,l}}(x_k))_r(\mathcal{T}_k u_{\theta_{k,m}}(x_k))_s\right]}_{:=K_{rs}^{lm,k}}\prod_{d\neq k}\underbrace{\mathbb{E}_{x_d}\left[(u_{\theta_{d,l}}(x_d))_r(u_{\theta_{d,m}}(x_d))_s\right]}_{:=M_{rs}^{lm,d}}.\end{aligned} \tag{8}$$

Similarly, we use 1D importance sampling to compute each kinetic energy $K_{rs}^{lm,k}$, and $M_{rs}^{lm,d}$ is reused from equation 6. Computing Π thus costs $\mathcal{O}(B_{1\mathrm{D}}NR^2L^2)$ as well by reusing $M_{rs}^{lm,d}$.

For multiplicative separable operators $\mathcal{T}=\prod_{d=1}^{N}\mathcal{T}_d$, we have

$$(\mathcal{T}u_m)(\boldsymbol{x}) = \prod_{k=1}^{N}\mathcal{T}_k\left(\sum_{s=1}^{R}\prod_{d=1}^{N}(u_{\theta_{d,m}}(x_d))_s\right) = \sum_{s=1}^{R}\prod_{d=1}^{N}(\mathcal{T}_d u_{\theta_{d,m}}(x_d))_s. \tag{9}$$

Substituting this into Π_{lm} allows us to decompose the integral into 1D quadrature:

$$\begin{aligned}\Pi_{lm} &= \mathbb{E}_{\boldsymbol{x}}\left[\left(\sum_{r=1}^{R}\prod_{d=1}^{N}(u_{\theta_{d,l}}(x_d))_r\right)\left(\sum_{s=1}^{R}\prod_{d'=1}^{N}(\mathcal{T}_{d'}u_{\theta_{d',m}}(x_{d'}))_s\right)\right]\\ &= \sum_{r=1}^{R}\sum_{s=1}^{R}\mathbb{E}_{\boldsymbol{x}}\left[\prod_{d=1}^{N}(u_{\theta_{d,l}}(x_d))_r(\mathcal{T}_d u_{\theta_{d,m}}(x_d))_s\right] = \sum_{r=1}^{R}\sum_{s=1}^{R}\prod_{d=1}^{N}\underbrace{\mathbb{E}_{x_d}\left[(u_{\theta_{d,l}}(x_d))_r(\mathcal{T}_d u_{\theta_{d,m}}(x_d))_s\right]}_{:=V_{rs}^{lm,d}}.\end{aligned} \tag{10}$$

The 1D operator integral $V_{rs}^{lm,d}$ is evaluated by importance sampling as well. Calculating the required elements of the stiffness Π for multiplicative operators costs $\mathcal{O}(B_{1\mathrm{D}}NR^2L^2)$ in equation 10.

2.1.3 Parametric Pre-training for Non-separable Operators

To show how TPQ evaluates non-separable operators, we detail the assembly of Π using the N-D Hydrogen atom as an example. The problem is governed by the time-independent Schrödinger equation with physical Hamiltonian $\mathcal{H} = -\frac{\hbar^2}{2\mu}\nabla^2 - \frac{Ze^2}{4\pi\epsilon_0\|\boldsymbol{x}\|_2}$. Following (Ryu et al., 2024), we use the rescaled form $\mathcal{H} = -\nabla^2 + V(\boldsymbol{x})$, where $V(\boldsymbol{x}) := -1/\|\boldsymbol{x}\|_2$, and optimize the sign-flipped operator $\mathcal{T} = -\mathcal{H} = \nabla^2 - V(\boldsymbol{x})$. Thus the optimized operator contains the Laplacian term ∇^2 and the repulsive sign-flipped potential term $-V(\boldsymbol{x}) = 1/\|\boldsymbol{x}\|_2$. By definition of the stiffness Π we have

$$\Pi_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})(\mathcal{T}u_m)(\boldsymbol{x})] = \underbrace{\mathbb{E}_{\boldsymbol{x}}\left[u_l(\boldsymbol{x})\left(\nabla^2 u_m(\boldsymbol{x})\right)\right]}_{:=\mathcal{K}_{lm}} - \underbrace{\mathbb{E}_{\boldsymbol{x}}\left[u_l(\boldsymbol{x})\left(V(\boldsymbol{x})u_m(\boldsymbol{x})\right)\right]}_{:=\mathcal{V}_{lm}}. \tag{11}$$

The kinetic contribution $\mathcal{K}_{lm}$ relies on the additively separable Laplacian $\nabla^2 = \sum_{d=1}^{N}\frac{\partial^2}{\partial {x_d}^2}$. Following equation 8, we can compute $\mathcal{K}_{lm} = \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{k=1}^{N}K_{rs}^{lm,k}\prod_{d\neq k}M_{rs}^{lm,d}$, which scales linearly with the dimension N by TPQ. The potential $V(\boldsymbol{x}) = -1/\|\boldsymbol{x}\|_2$ is non-separable. To preserve linear scalability, we approximate the potential using a pre-trained mixture of K isotropic Gaussians:

$$V(\boldsymbol{x}) \approx \sum_{k=1}^{K}\omega_k\exp(-\alpha_k\|\boldsymbol{x}\|_2^2) = \sum_{k=1}^{K}\omega_k\prod_{d=1}^{N}\exp(-\alpha_k x_d^2), \tag{12}$$

where $\{\omega_k, \alpha_k\}$ are Gaussian coefficients optimized offline to fit $V(\boldsymbol{x})$ prior to the main training (see PyTorch implementation in Sec. J.3; this pre-training approximation only takes several seconds to reach small approximation error in equation 12 using Adam). The Gaussian mixture transforms the potential into a sum of K multiplicatively separable operators. Combining equation 12 and equation 10, we have $\mathcal{V}_{lm} = \sum_{k=1}^{K}\omega_k\sum_{r=1}^{R}\sum_{s=1}^{R}\prod_{d=1}^{N}V_{rs,k}^{lm,d}$, where each $V_{rs,k}^{lm,d} := \frac{1}{B_{1\mathrm{D}}}\sum_{b=1}^{B_{1\mathrm{D}}}(u_{\theta_{d,l}}(x_{d,b}))_r(u_{\theta_{d,m}}(x_{d,b}))_s\exp(-\alpha_k x_{d,b}^2)$ is a 1D integral. Combining the

separately assembled terms according to $\mathcal{T} = -\mathcal{H}$ gives $\Pi_{lm} = \mathcal{K}_{lm} - \mathcal{V}_{lm}$ in equation 11; because $V(\boldsymbol{x}) < 0$ for hydrogen, this is equivalent to adding the positive sign-flipped Coulomb contribution. The TPQ assembly of the sign-flipped Hamiltonian stiffness Π is realized with computational cost staying linear with N.

Remark 1. *We remark that the Gaussian mixture we used is related to the contracted Gaussian-type orbitals (CGTO) developed in early quantum chemistry, which aims to represent Slater-type atomic orbitals as a sum of Gaussians (Hehre et al., 1969). The Gaussian mixture equation 12 holds two advantages: 1) it retains the linear scalability by avoiding point-wise evaluation of the potential (see ablation study of the Gaussian mixture strategy in Sec. G); and 2) it smooths the singularity of the Coulomb potential at origin, acting as a numerical regularization. The parametric methods for other potential functions (infinite potential well and harmonic oscillator) are illustrated in Sec. F.*

2.1.4 Separable Jacobian-based Masked Gradient

To recover ordered and orthogonal eigenfunctions, SpIN applies a Cholesky decomposition (which costs $\mathcal{O}(L^3)$) to the mass $\Sigma = \boldsymbol{L}\boldsymbol{L}^T$ and rewrites the objective as $\mathrm{Tr}(\Sigma^{-1}\Pi) = \mathrm{Tr}(\boldsymbol{L}^{-1}\Pi\boldsymbol{L}^{-T}) := \mathrm{Tr}(\Lambda)$. The masked gradient is to insulate the learning of the l-th eigenfunction from the gradients of any lower-order m-th function ($m < l$). When computing parameter gradients for diagonal elements Λ_{ll}, SpIN forces components that alter lower-order eigenfunctions to zero. This masked gradient can be derived as (see Eq. (14) in (Pfau et al., 2019)): $\tilde{\nabla}_\theta \mathrm{Tr}(\Lambda) = \mathbb{E}[\boldsymbol{J}_\Pi \odot \boldsymbol{A}_\Pi] - \mathbb{E}[\boldsymbol{J}_\Sigma \odot \boldsymbol{A}_\Sigma]$, where θ is the parameter vector of all neural networks, $\boldsymbol{A}_\Pi = \boldsymbol{L}^{-T}\mathrm{diag}(\boldsymbol{L})^{-1}$, $\boldsymbol{A}_\Sigma = \boldsymbol{L}^{-T}\mathrm{triu}(\Lambda\mathrm{diag}(\boldsymbol{L})^{-1}) \in \mathbb{R}^{L\times L}$ are derived multiplier masks, $\boldsymbol{J}_\Pi, \boldsymbol{J}_\Sigma \in \mathbb{R}^{L\times L\times|\theta|}$ are full Jacobian tensors, defined element-wise as $(\boldsymbol{J}_\Pi)_{lm} = \frac{\partial \Pi_{lm}}{\partial \theta}$ and $(\boldsymbol{J}_\Sigma)_{lm} = \frac{\partial \Sigma_{lm}}{\partial \theta}$, and $\odot$ denotes the inner product over eigenfunction indices, e.g., $\mathbb{E}[\boldsymbol{J}_\Pi \odot \boldsymbol{A}_\Pi] = \sum_{l=1}^{L}\sum_{m=1}^{L}(\boldsymbol{A}_\Pi)_{lm}\frac{\partial \Pi_{lm}}{\partial \theta}$. Based on this gradient, we elaborate on the local gradient for the (d,l)-th CP neural ansatz under TPQ strategy:

$$\nabla_{\theta_{d,l}}\mathcal{L}_\Pi := \sum_{l'=1}^{L}\sum_{m=1}^{L}(\boldsymbol{A}_\Pi)_{l'm}\frac{\partial \Pi_{l'm}}{\partial \theta_{d,l}},\ \nabla_{\theta_{d,l}}\mathcal{L}_\Sigma := \sum_{l'=1}^{L}\sum_{m=1}^{L}(\boldsymbol{A}_\Sigma)_{l'm}\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}. \tag{13}$$

We first calculate the Jacobian $\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}$ via a separable form. First, compute the inner product $M_{rs}^{l'm,k}$ in equation 6 for each eigen-pairs (l', m) and rank component (r, s) across all dimensions k except the target d, and obtain $C_{rs}^{l'm,d} = \prod_{k\neq d}^{N} M_{rs}^{l'm,k}$. The (approximated) Jacobian (denoted by $\widehat{\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}}$ for rigor) is reduced to the multiplication between the remaining 1D factors and the 1D network gradients (see Appendix Sec. D.1 and Lemma 1 for detailed derivation):

$$\widehat{\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}} := \frac{2\delta_{ml}}{B_{1\mathrm{D}}}\sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{b=1}^{B_{1\mathrm{D}}} C_{rs}^{l'm,d}(u_{\theta_{d,l'}}(x_{d,b}))_r w_d(x_{d,b})\frac{\partial(u_{\theta_{d,m}}(x_{d,b}))_s}{\partial \theta_{d,l}}, \tag{14}$$

where δ_{ml} denotes the Kronecker delta. Consequently, we use the corresponding approximate mass-gradient $\widehat{\nabla}_{\theta_{d,l}}\mathcal{L}_\Sigma = \sum_{l'=1}^{L}\sum_{m=1}^{L}(\boldsymbol{A}_\Sigma)_{l'm}\widehat{\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}}$ (code implementation in Sec. J.1). Computing the full grid Jacobian $\boldsymbol{J}_\Sigma$ requires $\mathcal{O}(B_{1\mathrm{D}}^N L^2 P)$, where P denotes the neural parameter count. The TPQ approach in equation 14 reduces to $\mathcal{O}(B_{1\mathrm{D}} N L^2 R^2 P)$ using the separable structure. Because Π involves cross-terms induced by operators, manual Jacobian derivation for $\frac{\partial \Pi_{l'm}}{\partial \theta_{d,l}}$ is practically much more complicated (but should be theoretically tractable). For simplicity, we delegate the Π-Jacobian to PyTorch's autograd (code implementation in Sec. J.1) and leave its manual separable derivation in future research. In total, the original SpIN costs $\mathcal{O}(B_{1\mathrm{D}}^N(F+L^2))+\mathcal{O}(B_{1\mathrm{D}}^N L^2 P)+\mathcal{O}(L^3)$ (Rayleigh trace, Jacobian mask, Cholesky) at each iteration. From a theoretical perspective, the SpIN-TPQ could hold linear computational costs $\mathcal{O}(B_{1\mathrm{D}} N(F + R^2L^2)) + \mathcal{O}(B_{1\mathrm{D}} N L^2 R^2 P) + \mathcal{O}(L^3)$ w.r.t. N (if neglecting the Π-Jacobian), as conceptually summarized in Table 1, and the efficiency would be empirically validated in experiments (Sec. 3).

2.2 NeuralEF-TPQ: Rayleigh Quotient via Tensor Product Quadrature

In NeuralEF-TPQ, the L eigenfunctions are parameterized by independent variable-separable neural networks in the CP format equation 4. A strict constraint in NeuralEF (equation 2) is the unit norm of each eigenfunction, i.e., $\Sigma_{ll} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})^2] = 1$. To efficiently implement the normalization under TPQ, we first calculate Σ_{ll} efficiently via the CP ansatz using equation 6: $\Sigma_{ll} =$

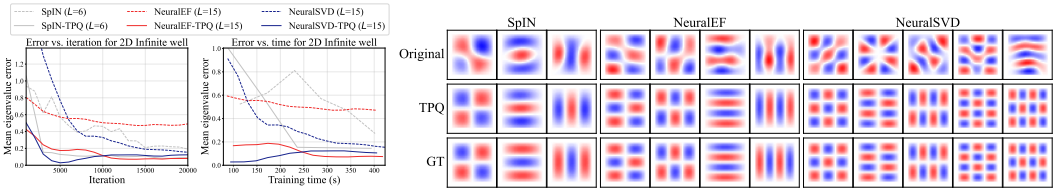


Figure 1: Results for the 2D infinite potential well. Left: estimated mean eigenvalue error w.r.t. training iteration and time. Right: native eigenfunction outputs (*without* alignment) selected from the eigenspaces associated with $n_x^2 + n_y^2$ =8,10 (SpIN), 13,17 (NeuralEF), and 18,20,25 (NeuralSVD). Ground-truth-aligned eigenfunction results are shown in Appendix Figs. 12-18.

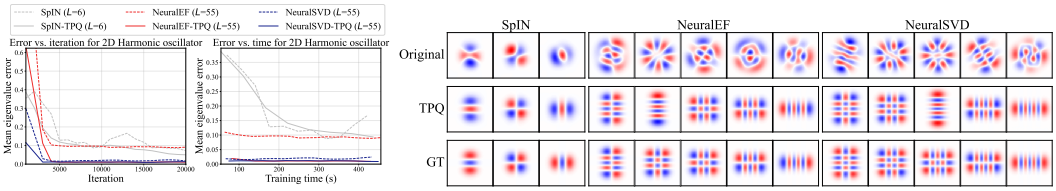


Figure 2: Results for the 2D harmonic oscillator. Left: estimated mean eigenvalue error w.r.t. training iteration and time. Right: native eigenfunction outputs (*without* alignment) corresponding to selected eigenfunctions from the 2nd (SpIN), 5th (NeuralEF), and 6th (NeuralSVD) energy levels.

$\sum_{r=1}^{R}\sum_{s=1}^{R}\prod_{d=1}^{N} M_{rs}^{ll,d}$. To enforce the unit norm constraint without evaluating dense N-D outputs, we introduce a variable-separable normalization. Especially, we normalize the output of each 1D factor $(u_{\theta_{d,l}}(x_d))_r$ during the forward pass by $\frac{(u_{\theta_{d,l}}(x_d))_r}{(\Sigma_{ll})^{\frac{1}{2N}}}$, which is equivalent to normalizing the combined N-D eigenfunction: $\sum_{r=1}^{R}\prod_{d=1}^{N}\frac{(u_{\theta_{d,l}}(x_d))_r}{(\Sigma_{ll})^{\frac{1}{2N}}} = \frac{\sum_{r=1}^{R}\prod_{d=1}^{N}(u_{\theta_{d,l}}(x_d))_r}{\left((\Sigma_{ll})^{\frac{1}{2N}}\right)^N} = \frac{u_l(\boldsymbol{x})}{\sqrt{\Sigma_{ll}}}$, making the TPQ-estimated $\Sigma_{ll} = 1$. NeuralEF optimizes the objective equation 2, which requires computing the stiffness matrix elements $\Pi_{lm} = \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})(\mathcal{T}u_m)(\boldsymbol{x})]$ for both the Rayleigh quotients ($l = m$) and the off-diagonal soft penalties ($m < l$). For a target operator containing both kinetic and non-separable potential energies, such as the sign-flipped Hamiltonian $\mathcal{T} = -\mathcal{H} = \nabla^2 - V(\boldsymbol{x})$, the TPQ assembly is performed as described in Sec. 2.1.3 by using Gaussian mixtures and the sign convention in equation 11:

$$\Pi_{lm} = \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{k=1}^{N} K_{rs}^{lm,k}\prod_{d\neq k} M_{rs}^{lm,d} - \sum_{k=1}^{K}\omega_k\sum_{r=1}^{R}\sum_{s=1}^{R}\prod_{d=1}^{N} V_{rs,k}^{lm,d}, \tag{15}$$

where $M_{rs}^{lm,d}$ is the mass between the l-th and m-th eigenfunctions, and $V_{rs,k}^{lm,d}$ is the 1D potential integrals evaluated using the pre-trained Gaussian mixtures (Sec. 2.1.3). The NeuralEF objective is simply $\max_{\{\theta_l\}_{l=1}^{L}}\sum_{l=1}^{L}\left(\Pi_{ll} - \sum_{m<l}\Pi_{lm}^2/\Pi_{mm}\right)$ (PyTorch code implementation in Sec. J.4). The original NeuralEF requires $\mathcal{O}(B_{1\mathrm{D}}^N(FL+L^2))$+$\mathcal{O}(B_{1\mathrm{D}}^N L^2)$ (Rayleigh quotient and soft penalty) per iteration, while NeuralEF-TPQ reduces to $\mathcal{O}(B_{1\mathrm{D}}N(FL+R^2L^2))$+$\mathcal{O}(B_{1\mathrm{D}}NR^2L^2)$ for the same N-D grid with $B_{1\mathrm{D}}$ points per dimension (summarized in Table 1). Also, NeuralEF-TPQ enforces orthogonality through soft penalty, avoiding the Jacobian computations in SpIN-TPQ.

2.3 NeuralSVD-TPQ: Nested Low-Rank Approximation via TPQ

We propose NeuralSVD-TPQ for SVD of non-self-adjoint operators. It parameterizes the right and left singular functions by two sequences of neural ansatzes $\{u_{\theta_{d,l}}(x_d)\}_{l=1}^{L}$ and $\{v_{\phi_{d,l}}(x_d)\}_{l=1}^{L}$ (Ryu et al., 2024) via the CP format equation 4: $u_l(\boldsymbol{x}) = \sum_{r=1}^{R}\prod_{d=1}^{N}(u_{\theta_{d,l}}(x_d))_r, v_l(\boldsymbol{x}) = \sum_{r=1}^{R}\prod_{d=1}^{N}(v_{\phi_{d,l}}(x_d))_r$, and minimizes the LoRA objective following equation 3:

$$\min_{\{\theta_{d,l},\phi_{d,l}\}_{l=1}^{L}}\mathcal{L}_{\mathrm{LoRA}}(\{\theta_{d,l},\phi_{d,l}\}_{l=1}^{L}) := \min_{\{\theta_{d,l},\phi_{d,l}\}_{l=1}^{L}} -2\sum_{l=1}^{L}\Pi_{ll} + \sum_{l=1}^{L}\sum_{m=1}^{L}\Sigma_{lm}^{u}\Sigma_{lm}^{v}, \tag{16}$$

where $\Sigma_{lm}^{u} := \mathbb{E}_{\boldsymbol{x}}[u_l(\boldsymbol{x})u_m(\boldsymbol{x})], \Sigma_{lm}^{v} := \mathbb{E}_{\boldsymbol{x}}[v_l(\boldsymbol{x})v_m(\boldsymbol{x})], \Pi_{ll} := \mathbb{E}_{\boldsymbol{x}}[v_l(\boldsymbol{x})(\mathcal{T}u_l)(\boldsymbol{x})]$. To recover ordered singular functions, we optimize a single joint nesting objective with positive weights $w_l > 0$

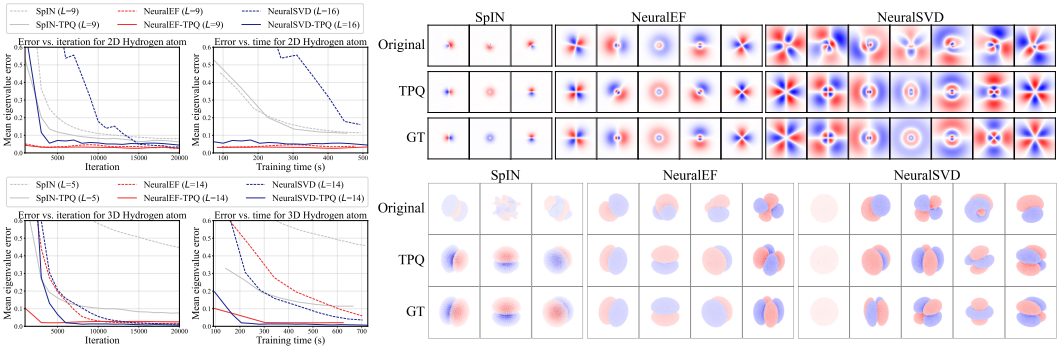


Figure 3: Results for the hydrogen atom in 2D (top) and 3D (bottom). Left: estimated mean eigenvalue error w.r.t. training iteration and time. Right: native eigenfunction outputs (*without* alignment) selected from degenerate hydrogenic energy levels. In 2D, the outputs correspond to the 2nd (SpIN), 3rd (NeuralEF), and 4th (NeuralSVD) levels. In 3D, the outputs are selected from the 2nd level for SpIN and the 3rd level for NeuralEF and NeuralSVD.

(Ryu et al., 2024): $\min \sum_{l=1}^{L} w_l \mathcal{L}_{\mathrm{LoRA}}(\{\theta_{d,m}, \phi_{d,m}\}_{m=1}^{l})$. Rearranging the summation yields a simple masked formulation:

$$\min -2 \sum_{l=1}^{L} \boldsymbol{a}_l \, \Pi_{ll} + \sum_{l=1}^{L} \sum_{m=1}^{L} \boldsymbol{Q}_{lm} \, \Sigma_{lm}^{u} \Sigma_{lm}^{v}, \;\; \boldsymbol{a}_l := \sum_{m=l}^{L} w_m, \;\; \boldsymbol{Q}_{lm} := \min(\boldsymbol{a}_l, \boldsymbol{a}_m). \tag{17}$$

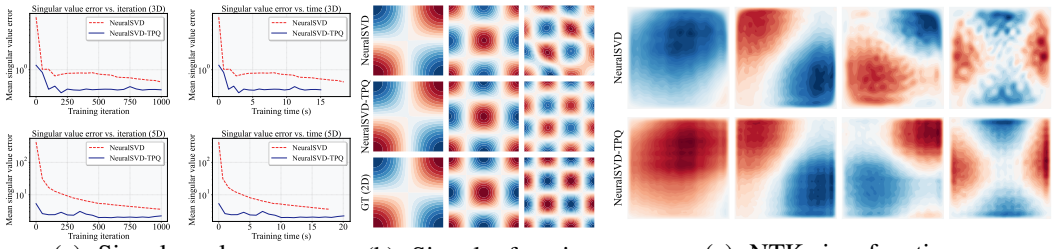
(see PyTorch implementation in Sec. J.4). Following the same TPQ as equation 6, the two mass matrices are assembled by $\Sigma_{lm}^{u} = \sum_{r=1}^{R} \sum_{s=1}^{R} \prod_{d=1}^{N} M_{rs}^{u,lm,d}$, $\Sigma_{lm}^{v} = \sum_{r=1}^{R} \sum_{s=1}^{R} \prod_{d=1}^{N} M_{rs}^{v,lm,d}$, where $M_{rs}^{u,lm,d} := \mathbb{E}_{x_d}\left[(u_{\theta_{d,l}}(x_d))_r \, (u_{\theta_{d,m}}(x_d))_s\right]$, $M_{rs}^{v,lm,d} := \mathbb{E}_{x_d}\left[(v_{\phi_{d,l}}(x_d))_r \, (v_{\phi_{d,m}}(x_d))_s\right]$. Each is estimated by importance sampling with batch size $B_{1\mathrm{D}}$, as in equation 6. Computing both mass matrices costs $\mathcal{O}(B_{1\mathrm{D}} N R^2 L^2)$.

NeuralSVD-TPQ only requires the diagonal operator terms Π_{ll} in the LoRA objective. Denote $M_{rs}^{vu,ll,d} := \mathbb{E}_{x_d}\left[(v_{\phi_{d,l}}(x_d))_r \, (u_{\theta_{d,l}}(x_d))_s\right]$ and $K_{rs}^{vu,ll,k} := \mathbb{E}_{x_k}\left[(v_{\phi_{k,l}}(x_k))_r \, (\mathcal{T}_k u_{\theta_{k,l}}(x_k))_s\right]$ for additively separable operator $\mathcal{T} = \sum_{k=1}^{N} \mathcal{T}_k$. Then we calculate the diagonal separable operator expectation (kinetic) $\mathcal{K}_{ll} = \sum_{r=1}^{R} \sum_{s=1}^{R} \sum_{k=1}^{N} K_{rs}^{vu,ll,k} \prod_{d \neq k} M_{rs}^{vu,ll,d}$ as in equation 8. For non-separable operators such as the sign-flipped Hamiltonian $\mathcal{T} = -\mathcal{H} = \nabla^2 - V(\boldsymbol{x})$, we approximate $V(\boldsymbol{x})$ using Gaussian mixtures as described in equation 12, and calculate each 1D operator integral: $V_{rs,k}^{vu,ll,d} := \mathbb{E}_{x_d}\left[(v_{\phi_{d,l}}(x_d))_r \, (u_{\theta_{d,l}}(x_d))_s \, \exp(-\alpha_k x_d^2)\right]$. The potential contribution $\mathcal{V}_{ll}$ is assembled via $\mathcal{V}_{ll} = \sum_{k=1}^{K} \omega_k \sum_{r=1}^{R} \sum_{s=1}^{R} \prod_{d=1}^{N} V_{rs,k}^{vu,ll,d}$. For the self-adjoint Hamiltonian validation, this reduces to the EVD setting with $u_{\theta_{d,l}} = v_{\phi_{d,l}}$. Finally, the overall diagonal stiffness element follows the optimized operator $\mathcal{T} = -\mathcal{H}$, i.e., $\Pi_{ll} = \mathcal{K}_{ll} - \mathcal{V}_{ll}$. Minimizing equation 17 follows a direct autograd path (a full NeuralSVD-TPQ code implementation is placed in Sec. J.5).

The original NeuralSVD costs $\mathcal{O}(B_{1\mathrm{D}}^{N}(FL + L^2)) + \mathcal{O}(B_{1\mathrm{D}}^{N} L^2)$ per iteration (stiffness objective and mass matrices soft penalty), while NeuralSVD-TPQ reduces to $\mathcal{O}(B_{1\mathrm{D}} N(FL + R^2 L^2)) + \mathcal{O}(B_{1\mathrm{D}} N R^2 L^2)$, as summarized in Table 1. Further details on how TPQ handles other potential functions (additively separable harmonic and infinite well potentials) and decomposes the NTK operator and non-self-adjoint integral kernel operator are provided in Appendix Sec. F.

3 Numerical Experiments

We evaluate TPQ on the time-independent Schrödinger equations (Hamiltonian EVD, including 2D infinite well, harmonic oscillator, 2D/3D hydrogen atom), kernel integral operator SVD, and NTK EVD. All problem formulations and detailed settings are comprehensively given in Appendix Secs. F.1-F.3. For Hamiltonian EVD, we apply the objectives to $-\mathcal{H}$ so that low-energy states correspond to leading eigenvalues. Across the 2D infinite well, harmonic oscillator, and 2D/3D hydrogen atom benchmarks, TPQ consistently improves the convergence and eigenfunction estimation accuracy of SpIN, NeuralEF, and NeuralSVD (Figs. 1-3; more eigenfunction snapshots in Figs. 11-18). The improvement is most visible for excited states, where MC quadrature may hold high variance due to oscillatory excited eigenfunctions. In addition, TPQs recover meaningful



(a): Singular value error (b): Singular function (c): NTK eigenfunctions

Figure 4: (a)-(b) Results for kernel integral operator SVD using NeuralSVD and TPQ (right singular functions in (b)). (c) Decomposed NTK eigenfunctions learned by NeuralSVD and TPQ.

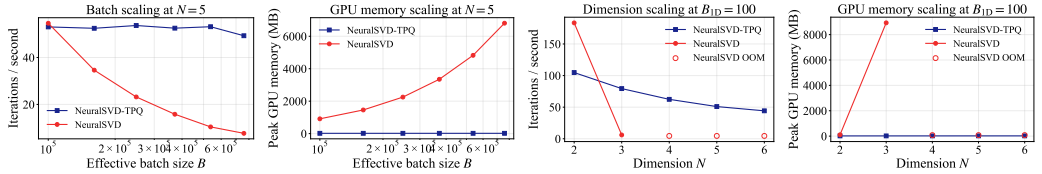


Figure 5: Scaling analysis: iterations per second or GPU peak memory vs. effective batch size B_{1D}^N or dimension N for the kernel operator SVD experiment. OOM stands for out-of-memory.

and accurate native eigenfunctions in degenerate spaces without relying on ground-truth (GT) alignment (Figs. 1-2), indicating that TPQ improves not only the eigenvalue estimation but also the learned functional subspace.

Table 2: Surrogate KRR testing MSE using reconstructed NTKs under different train/test splits.

Split	NeuralSVD	NeuralSVD-TPQ
8:2	0.304	**0.186**
6:4	0.312	**0.196**
4:6	0.341	**0.205**
2:8	0.373	**0.219**

We further test NeuralSVD-TPQ on kernel integral operator SVD up to 5D (detailed settings in Sec. F.2). As shown in Fig. 4(a)-(b), TPQ yields lower singular-value error and more stable singular functions than NeuralSVD; complete 2D-5D curves are provided in Fig. 19. Scaling analysis w.r.t. effective batch size B_{1D}^N and dimension N in Fig. 5 validates the intended linear scaling behavior of TPQ, whose iterations per second and peak GPU memory cost do not change abruptly with intensively increasing effective batch size and dimension due to lightweight 1D quadrature. For NTK operator EVD (detailed settings in Sec. F.3), Fig. 4(c) and Appendix Fig. 20 show that TPQ acts as a structural regularizer, producing smoother learned eigenfunctions of the NTK integral operator. To quantify whether this improves the fidelity of the reconstructed kernel using EVD truncation, we plug the reconstructed NTKs into a kernel ridge regression (KRR) problem to test the approximation fidelity. Table 2 shows that NeuralSVD-TPQ achieves lower test MSE than NeuralSVD across the train/test splits, demonstrating improved downstream utility of the reconstructed kernel.

More numerical comparisons with classical non-parametric eigensolvers (Lanczos (Lanczos, 1950), LOBPCG (Knyazev, 2001)) are given in Appendix Sec. B. We also combine TPQ with power method neural network-based eigensolvers (Yang et al., 2023; Wang et al., 2025) up to 6D in Sec. A. We study the effect of the CP rank R in Sec. C. Finally, we theoretically analyze the approximation error of spectral functions and truncated operators using TPQ w.r.t. rank in Sec. D.2, and provide PyTorch code snippets in Appendix Sec. J.

Discussions We introduced the TPQ framework to alleviate the computation-accuracy trade-off in neural operator decomposition. By parameterizing each spectral function via a variable-separable neural network, we effectively decouple the N-D integral into independent 1D quadratures and tensor product. This reduces the scaling of computational complexity from $\mathcal{O}(B_{1D}^N)$ to $\mathcal{O}(B_{1D}N)$. Numerical evaluations for Schrödinger equations, integral kernel operator SVD, and NTK EVD demonstrate that TPQ improves the accuracy and scalability of neural EVD/SVD baselines. Our method provides a practical solution to the open challenge raised in (Ryu et al., 2024), that "scalability challenges may require more sophisticated architectures". While the CP decomposition used in the current TPQ is efficient, modeling strongly correlated or highly entangled high-dimensional systems might require a large CP-rank (R) to maintain expressivity. A natural future direction is to integrate more expressive tensor network topologies, such as the tensor train (TT) format (Oseledets, 2011), into the neural function representation. We discuss more future directions and give preliminary results on a many-body Schrödinger equation in Sec. H.

REFERENCES

Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, 2003.

Junwoo Cho, Seungtae Nam, Hyunmo Yang, Seok-Bae Yun, Youngjoon Hong, and Eunbyung Park. Separable physics-informed neural networks. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pp. 23761 – 23788, 2023.

Zhijie Deng, Jiaxin Shi, and Jun Zhu. NeuralEF: Deconstructing kernels by deep neural networks. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162, pp. 4976–4992, 2022.

Weinan E and Bing Yu. The deep ritz method: A deep learning-based numerical algorithm for solving variational problems. *Communications in Mathematics and Statistics*, 6:1–12, 2018. doi: 10.1007/s40304-018-0127-z.

Ian Gemp, Brian McWilliams, Claire Vernade, and Thore Graepel. Eigengame: $\{\text{PCA}\}$ as a nash equilibrium. In *International Conference on Learning Representations*, 2021.

David Gottlieb and Steven A. Orszag. Spectral methods. In *Numerical Analysis of Spectral Methods*, pp. 7–19, 1977.

W. J. Hehre, R. F. Stewart, and J. A. Pople. Self-consistent molecular-orbital methods. i. use of gaussian expansions of slater-type atomic orbitals. *The Journal of Chemical Physics*, 51(6):2657–2664, 1969.

Jan Hermann, Zeno Schätzle, and Frank Noé. Deep-neural-network solution of the electronic schrödinger equation. *Nature Chemistry*, 12:891–897, 2020.

Minchan Jeong, Jongha Jon Ryu, Se-Young Yun, and Gregory W. Wornell. Efficient parametric SVD of koopman operator for stochastic dynamical systems. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Andrew V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM Journal on Scientific Computing*, 23(2):517–541, 2001.

Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51 (3):455–500, 2009.

Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4): 255–282, 1950.

Yisi Luo and Deyu Meng. Separable neural networks: Approximation theory, NTK regime, and preconditioned gradient descent. In *The Fourteenth International Conference on Learning Representations*, 2026.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, pp. 2408–2417, 2015.

I. V. Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

Tianyu Pang, Shuicheng YAN, and Min Lin. $O(N^2)$ universal antisymmetry in fermionic neural networks. In *ICML 2022 2nd AI for Science Workshop*, 2022.

David Pfau, Stig Petersen, Ashish Agarwal, David G. T. Barrett, and Kimberly L. Stachenfeld. Spectral inference networks: Unifying deep and spectral learning. In *International Conference on Learning Representations*, 2019.

David Pfau, James S. Spencer, Alexander G. D. G. Matthews, and W. M. C. Foulkes. Ab initio solution of the many-electron schrödinger equation with deep neural networks. *Phys. Rev. Res.*, 2:033429, 2020.

Jongha Jon Ryu, Xiangxiang Xu, Hasan Sabri Melihcan Erol, Yuheng Bu, Lizhong Zheng, and Gregory W. Wornell. Operator SVD with neural networks via nested low-rank approximation. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 42870–42905, 2024.

Jongha Jon Ryu, Samuel Zhou, and Gregory W. Wornell. Revisiting orbital minimization method for neural operator decomposition. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

E Schmidt. Zur theorie der linearen und nichtlinearen integralgleichungen. *Mathematische Annalen*, 63:433–476, 1907.

Bernhard Schölkopf and Alexander Smola. Learning with kernels: Support vector machines, regularization, optimization, and beyond. *IEEE Transactions on Neural Networks*, 16(3):781–781, 2005.

Hong Wang, Jiang Yixuan, Jie Wang, Xinyi Li, Jian Luo, and huanshuo dong. STNet: Spectral transformation network for solving operator eigenvalue problem. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Yifan Wang and Hehu Xie. Computing multi-eigenpairs of high-dimensional eigenvalue problems using tensor neural networks. *Journal of Computational Physics*, 506:112928, 2024.

Yifan Wang, Hehu Xie, and Pengzhan Jin. Tensor neural network and its numerical integration. *Journal of Computational Mathematics*, 42(6):1714–1742, 2024.

Christopher K. I. Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In *Proceedings of the 14th International Conference on Neural Information Processing Systems*, pp. 661–667, 2000.

Qihong Yang, Yangtao Deng, Yu Yang, Qiaolin He, and Shiquan Zhang. Neural networks based on power method and inverse power method for solving linear eigenvalue problems. *Comput. Math. Appl.*, 147:14–24, 2023.

CONTENTS OF APPENDIX

A Tensor Product Quadrature for Power Method Neural Network **13**

B Comparison to Non-parametric Approaches **14**

C Accuracy w.r.t. Rank R **14**

D Theoretical Analyses **15**

D.1 Separable Jacobian Derivation 15

D.2 Approximation Ability of TPQ 17

D.3 Variance of MC-based Numerical Integral 21

D.4 Variance Explosion Rate of High-dimensional Function 22

D.5 Variance Comparison Between Direct MC and TPQ Estimators 23

E Distinctions Between the Proposed Method and Tensor Neural Network Method **25**

F Problem Formulation **26**

F.1 Time-independent Schrödinger Equation 26

F.2 Integral Kernel Operator SVD 30

F.3 NTK Decomposition 30

F.4 Boundary Mask Choice 32

G Ablation Study of Pre-trained Separable Potential Function **32**

H Future Directions **32**

I More Numerical Results **34**

J PyTorch Code Examples **45**

J.1 PyTorch Code of the Separable Jacobian Gradient 45

J.2 PyTorch Code of the TPQ Evaluation 47

J.3 PyTorch Code of the Separable Potential Function Pre-training 48

J.4 PyTorch Code of the Optimization Objectives in TPQ 49

J.5 PyTorch Code of a NeuralSVD-TPQ Example 51

A TENSOR PRODUCT QUADRATURE FOR POWER METHOD NEURAL NETWORK

We illustrate the integration of our TPQ with the power method neural network method. The power method neural network (PMNN) (Yang et al., 2023; Wang et al., 2025) provides a continuous, differentiable framework for finding the dominant eigenvalue (or middle eigenvalues by spectral transformation (Wang et al., 2025)) of linear operators. For a given differential operator $\mathcal{T}$ (e.g., the negative Laplacian $\mathcal{T} = -\Delta$) and a target eigenfunction $u(\boldsymbol{x})$, the eigenvalue problem is defined as $\mathcal{T}u = \lambda u$. PMNN frames this as an alignment problem in function space based on the power method. By parameterizing the function with a neural network $u_\theta(\boldsymbol{x})$, the objective is to minimize the difference between the normalized operator action and the normalized function itself (Yang et al., 2023). The optimization objective is formulated using the residual loss:

$$\mathcal{L}_{\mathrm{PMNN}}(\theta) = \left\| \frac{\mathcal{T}u_\theta}{\|\mathcal{T}u_\theta\|} - \mathrm{sg}\left(\frac{u_\theta}{\|u_\theta\|}\right) \right\|^2,$$

where $\mathrm{sg}(\cdot)$ denotes the stop-gradient operation. By iteratively aligning u_θ with $\mathcal{T}u_\theta$, the network converges to the dominant eigenfunction. The corresponding eigenvalue is then estimated using the Rayleigh quotient $\lambda \approx \frac{\langle \mathcal{T}u_\theta, u_\theta \rangle}{\langle u_\theta, u_\theta \rangle}$.

In high-dimensional spaces, evaluating the norms $\|\mathcal{T}u_\theta\|$ and $\|u_\theta\|$ using standard MC or grid-based sampling suffers from the curse of dimensionality. The numerical error of standard MC grows exponentially with the dimension N (as shown in Sec. D.3). To bypass this, we integrate the TPQ with PMNN. We parameterize the eigenfunction using the variable-separable neural network (4):

$$u_\theta(\boldsymbol{x}) = \sum_{r=1}^{R} \prod_{d=1}^{N} (u_{\theta_d}(x_d))_r.$$

This variable-separable structure allows the continuous inner products within the PMNN loss to be factorized into 1D integrals on grids. For the denominator $\|u_\theta\|^2$, the integration is computed via 1D tensor contractions:

$$\|u_\theta\|^2 = \sum_{r=1}^{R}\sum_{s=1}^{R} \prod_{d=1}^{N} M_{rs}^d, \;\; M_{rs}^d = \int (u_{\theta_d}(x_d))_r (u_{\theta_d}(x_d))_s dx_d.$$

Similarly, for the operator action $\mathcal{T} = -\Delta$ (we take the Laplace eigenvalue problem as an example), the multi-dimensional Laplacian separates into 1D second-order derivatives along each 1D function (since the Laplacian acts on each dimension separately $\mathcal{T}u_\theta = -\sum_{r=1}^{R}\sum_{i=1}^{N}(u''_{\theta_i}(x_i))_r \prod_{d\neq i}(u_{\theta_d}(x_d))_r$) followed by tensor contraction. Applying the central finite difference or exact 1D automatic differentiation on the separated dimensions yields the kinetic energy components. The Laplacian norm $\|\mathcal{T}u_\theta\|^2$ is computed via

$$W_{rs}^d = \int (u''_{\theta_d}(x_d))_r (u''_{\theta_d}(x_d))_s dx_d, \;\; K_{rs}^d = -\int (u''_{\theta_d}(x_d))_r (u_{\theta_d}(x_d))_s dx_d. \tag{18}$$

$$\|\mathcal{T}u_\theta\|^2 = \sum_{r=1}^{R}\sum_{s=1}^{R} \left[\sum_{i=1}^{N} W_{rs}^i \prod_{\substack{d=1 \\ d\neq i}}^{N} M_{rs}^d + \sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} K_{rs}^i K_{sr}^j \prod_{\substack{d=1 \\ d\neq i,j}}^{N} M_{rs}^d \right]. \tag{19}$$

The PMNN residual loss is thus computed deterministically over a massive grid of batch size B_{1D}^N given the 1D batch size B_{1D}. For these 1D integrals, we can use dense MC sampling or simply dense deterministic grids to ensure high accuracy. We use the latter in this example.

To evaluate the scalability of PMNN-TPQ, we set up the N-dimensional Laplace eigenvalue problem $-\Delta u = \lambda u$ defined on the unit hypercube $\Omega = [0,1]^N$. We consider the boundary conditions $u(\boldsymbol{x}) = 0$ on $\partial\Omega$, and the true ground state eigenvalue is $\lambda = N\pi^2$. To enforce the hard Dirichlet boundary conditions while preserving the separability of the TPQ, we apply a variable-separable boundary mask $m(\boldsymbol{x}) = m(x_1, \cdots, x_N)$ to the network outputs:

$$m(\boldsymbol{x}) = \prod_{d=1}^{N} (e^{x_d} - 1)\left(e^{-(x_d-1)} - 1\right),$$

Table 3: Comparison of relative errors in estimated eigenvalues and computation times between standard PMNN (Yang et al., 2023) and PMNN-TPQ across spatial dimensions $N \in \{2, \ldots, 6\}$. To demonstrate efficiency, we compare standard PMNN against PMNN-TPQ at equivalent batch sizes ($B = 1000$) to highlight the ultra-fast execution of TPQ, and at an expanded batch size ($B = 100,000$) to showcase TPQ's superior accuracy under a matched computational time budget. Note that executing PMNN at $B = 100000$ hits OOM and hence is omitted.

Dim (N)	**True** λ	**PMNN** ($B = 200$)		**PMNN** ($B = 1000$)		**PMNN-TPQ** ($B = 1000$)		**PMNN-TPQ** ($B = 100000$)	
		Rel. error	**Time (s)**	**Rel. error**	**Time (s)**	**Rel. error**	**Time (s)**	**Rel. error**	**Time (s)**
2	$2\pi^2 \approx 19.739$	6.76e-06	7.35	4.05e-06	27.66	5.79e-07	2.61	**3.54e-07**	8.57
3	$3\pi^2 \approx 29.608$	3.16e-04	10.93	5.51e-05	38.57	1.22e-06	4.02	**6.44e-07**	11.02
4	$4\pi^2 \approx 39.478$	3.47e-04	11.54	1.16e-05	40.49	1.52e-04	6.15	**1.73e-06**	18.44
5	$5\pi^2 \approx 49.348$	4.48e-03	14.66	3.64e-05	57.34	4.34e-03	8.60	**9.27e-07**	32.52
6	$6\pi^2 \approx 59.217$	7.37e-03	17.62	3.95e-04	70.14	3.84e-03	14.28	**3.54e-06**	41.18

which ensures $u_\theta(\boldsymbol{x})$ vanishes at the boundaries without changing the CP tensor function format ($m(\boldsymbol{x})$ can be separated into each 1D integral).

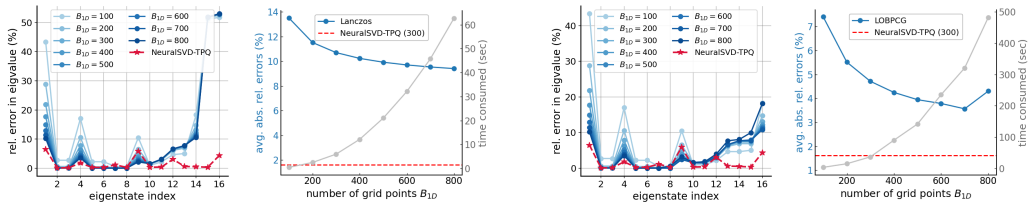
We evaluate the standard PMNN (Yang et al., 2023) against our PMNN-TPQ across spatial dimensions scaling from $N = 2$ to $N = 6$; see Table 3. To demonstrate the efficiency and scalability of our method, we conduct comparisons under two distinct settings. First, we compare the models at an equivalent batch size ($B = 1000$). In this setting, PMNN-TPQ exhibits significantly lower computation times due to the ultra-fast execution of TPQ, achieving substantial speedups (e.g., computing roughly 5 times faster at $N = 6$). Second, to showcase our model's performance under a matched computational time budget, we expand the batch size of PMNN-TPQ to $B = 100000$. As shown in Table 3, when given a similar time allowance (note PMNN-TPQ ($B = 100000$) is still faster than PMNN ($B = 1000$)), PMNN-TPQ consistently achieves superior accuracy, yielding relative errors that are up to two orders of magnitude lower than the standard PMNN across all tested dimensions.

B Comparison to Non-parametric Approaches

To further evaluate how our parametric approach compares to standard non-parametric numerical techniques, we performed baseline experiments using two standard non-parametric methods: the Lanczos algorithm (Lanczos, 1950) and the locally optimal block preconditioned conjugate gradient (LOBPCG) method (Knyazev, 2001), aiming to find the top $L = 16$ eigenstates of the 2D hydrogen atom experiment. We discretized the Hamiltonian domain and varied the number of grid points per axis, $B_{1\mathrm{D}}$, from 100 to 800. The comprehensive results of these comparisons are summarized in Fig. 6. The first panels of Fig. 6(a) and (b) present the relative errors of the estimated eigenvalues for Lanczos and LOBPCG across different grid resolutions. We can observe that for the traditional methods, smaller grid sizes ($B_{1\mathrm{D}}$) result in fluctuations in estimation errors across the eigenstates, exhibiting high relative errors at the lower and upper ends of the target spectrum. While increasing the number of grid points generally pushes the error curves downward, the proposed NeuralSVD-TPQ method (indicated by the red curve with stars) consistently maintains a stable and low error rate across most eigenstates, showcasing superior robustness. The second panels of Fig. 6(a) and (b) highlight the trade-off between accuracy and computational cost of traditional solvers. The blue line summarizes the average absolute relative error for each $B_{1\mathrm{D}}$, while the gray line tracks the time consumed. As expected, the accuracy of both Lanczos and LOBPCG improves as $B_{1\mathrm{D}}$ becomes larger; however, this improvement is heavily penalized by a drastic surge in computation time. The red dashed horizontal line indicates the error achieved by the proposed NeuralSVD-TPQ (evaluated at $B_{1\mathrm{D}} = 300$, taking < 6 minutes to achieve the error). The comparison showcases the general accuracy and scalability advantage of NeuralSVD-TPQ over traditional non-parametric methods.

C Accuracy w.r.t. Rank R

Fig. 7 evaluates the sensitivity of NeuralSVD-TPQ to the tensor-product rank R on the 2D hydrogen atom. As R increases in the early stage, both the mean relative eigenvalue error and the best achieved error (min error) decrease consistently, indicating that a larger CP rank improves the expressive



(a) vs. Lanczos (b) vs. LOBPCG

Figure 6: Comparison between NeuralSVD-TPQ and two non-parametric eigensolvers (a) Lanczos and (b) LOBPCG on the 2D hydrogen atom experiment.

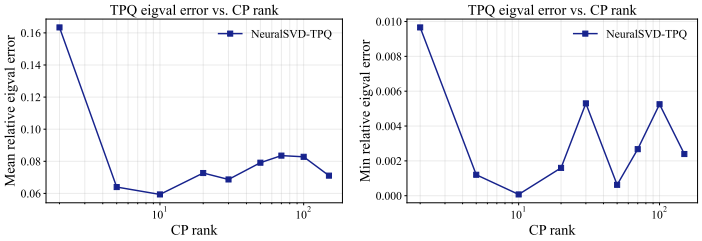


Figure 7: Rank scaling analysis on the 2D hydrogen atom using NeuralSVD-TPQ. Left: mean relative eigenvalue error across iterations; Right: min relative eigenvalue error across iterations.

capacity of the separable spectral-function ansatz when R is not too large. The improvement is pronounced when moving from very small ranks to moderate ranks, after which the curves become flatter, suggesting that the dominant eigenfunctions can already be represented accurately with a moderate tensor rank.

D Theoretical Analyses

We provide theoretical analyses to clarify that the Jacobian terms required by the SpIN gradient mask admit a separable tensor-product form, so the corresponding derivatives can be assembled from 1D quantities rather than full N-D grids. We then study the approximation capacity of the rank-R TPQ ansatz and derive a rank-dependent error characterization for the spectral decomposition problem. Finally, we analyze the variance behavior of Monte Carlo quadrature in high-dimensional spectral objectives, highlighting why direct mini-batch integration becomes unstable for oscillatory higher-order eigenfunctions. We further provide Lean formalizations of our main results including Lemma 1 (Separable Jacobian), Lemma 3 (TPQ approximation of spectral functions), and Theorem 1 (TPQ-restricted low-rank approximation error); the corresponding Lean source code is included in the supplementary attachment.

D.1 Separable Jacobian Derivation

In Sec. 2.1.4, the exact mass Jacobian $\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}$ can be decoupled into products of the remaining 1D factors and the 1D network gradients in a two-sided manner (see Lemma 1):

$$\begin{aligned}\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}} =& \frac{\delta_{l'l}}{B_{1\mathrm{D}}} \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{b=1}^{B_{1\mathrm{D}}} C_{rs}^{l'm,d} (u_{\theta_{d,m}}(x_{d,b}))_s w_d(x_{d,b}) \frac{\partial (u_{\theta_{d,l'}}(x_{d,b}))_r}{\partial \theta_{d,l}} \\ &+ \frac{\delta_{ml}}{B_{1\mathrm{D}}} \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{b=1}^{B_{1\mathrm{D}}} C_{rs}^{l'm,d} (u_{\theta_{d,l'}}(x_{d,b}))_r w_d(x_{d,b}) \frac{\partial (u_{\theta_{d,m}}(x_{d,b}))_s}{\partial \theta_{d,l}},\end{aligned} \tag{20}$$

where the Kronecker deltas $\delta_{l'l}$ and δ_{ml} ensure $\theta_{d,l}$ generates gradients only when the index l' or m matches the target parameter index l.

In practical implementation, we utilize a symmetric one-sided formulation to approximate this Jacobian. Specifically, we compute the second term and double it:

$$\widehat{\frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}}} := \frac{2\delta_{ml}}{B_{1\mathrm{D}}} \sum_{r=1}^{R} \sum_{s=1}^{R} \sum_{b=1}^{B_{1\mathrm{D}}} C_{rs}^{l'm,d} (u_{\theta_{d,l'}}(x_{d,b}))_r w_d(x_{d,b}) \frac{\partial (u_{\theta_{d,m}}(x_{d,b}))_s}{\partial \theta_{d,l}}. \tag{21}$$

While the population-level mass Jacobian is strictly symmetric ($\frac{\partial \Sigma_{l'm}}{\partial \theta} = \frac{\partial \Sigma_{ml'}}{\partial \theta}$), the multiplier mask $\boldsymbol{A}_\Sigma$ used in the gradient trace is non-symmetric due to the triangular structure of the Cholesky decomposition. Consequently, replacing the two-sided Jacobian with the doubled one-sided estimator equation 21 acts as a biased gradient estimator. However, we empirically find that this formulation improves convergence stability. The possible reason is that by avoiding the independent estimation of both sides of equation 20, the one-sided approximation mitigates the sampling variance under finite batch sizes, serving as a beneficial architectural regularization.

Lemma 1 (Separable Jacobian Derivation)**.** *Let the l-th and m-th CP eigenfunction ansatzes be $u_l(\boldsymbol{x}) = \sum_{r=1}^{R} \prod_{d=1}^{N} (u_{\theta_{d,l}}(x_d))_r$ and $u_m(\boldsymbol{x}) = \sum_{s=1}^{R} \prod_{d=1}^{N} (u_{\theta_{d,m}}(x_d))_s$. For the mass matrix $\Sigma_{l'm} = \mathbb{E}_{\boldsymbol{x}}[u_{l'}(\boldsymbol{x}) u_m(\boldsymbol{x})]$, define $M_{rs}^{l'm,k} := \mathbb{E}_{x_k}\left[(u_{\theta_{k,l'}}(x_k))_r (u_{\theta_{k,m}}(x_k))_s\right], C_{rs}^{l'm,d} := \prod_{k \neq d} M_{rs}^{l'm,k}$. Then the derivative w.r.t. the parameters of the (d,l)-th 1D network is*

$$\begin{aligned} \frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}} &= \delta_{l'l} \sum_{r=1}^{R} \sum_{s=1}^{R} C_{rs}^{l'm,d}\, \mathbb{E}_{x_d}\left[(u_{\theta_{d,m}}(x_d))_s \frac{\partial (u_{\theta_{d,l'}}(x_d))_r}{\partial \theta_{d,l}}\right] \\ &\quad + \delta_{ml} \sum_{r=1}^{R} \sum_{s=1}^{R} C_{rs}^{l'm,d}\, \mathbb{E}_{x_d}\left[(u_{\theta_{d,l'}}(x_d))_r \frac{\partial (u_{\theta_{d,m}}(x_d))_s}{\partial \theta_{d,l}}\right]. \end{aligned} \tag{22}$$

Proof. Note that

$$\begin{aligned} \Sigma_{l'm} &= \mathbb{E}_{\boldsymbol{x}}\left[\left(\sum_{r=1}^{R} \prod_{k=1}^{N} (u_{\theta_{k,l'}}(x_k))_r\right)\left(\sum_{s=1}^{R} \prod_{k=1}^{N} (u_{\theta_{k,m}}(x_k))_s\right)\right] \\ &= \sum_{r=1}^{R} \sum_{s=1}^{R} \mathbb{E}_{\boldsymbol{x}}\left[\prod_{k=1}^{N} (u_{\theta_{k,l'}}(x_k))_r (u_{\theta_{k,m}}(x_k))_s\right]. \end{aligned} \tag{23}$$

Since the integration measure factorizes independently across dimensions, the expectation of the product factorizes into the product of 1D expectations $\Sigma_{l'm} = \sum_{r=1}^{R} \sum_{s=1}^{R} \prod_{k=1}^{N} M_{rs}^{l'm,k}$. We differentiate this expression w.r.t. the target parameter $\theta_{d,l}$:

$$\begin{aligned} \frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}} &= \frac{\partial}{\partial \theta_{d,l}}\left(\sum_{r=1}^{R} \sum_{s=1}^{R} \prod_{k=1}^{N} M_{rs}^{l'm,k}\right) \\ &= \sum_{r=1}^{R} \sum_{s=1}^{R} \left(\prod_{k \neq d} M_{rs}^{l'm,k}\right) \frac{\partial M_{rs}^{l'm,d}}{\partial \theta_{d,l}} = \sum_{r=1}^{R} \sum_{s=1}^{R} C_{rs}^{l'm,d} \frac{\partial M_{rs}^{l'm,d}}{\partial \theta_{d,l}}. \end{aligned} \tag{24}$$

For the term $\frac{\partial M_{rs}^{l'm,d}}{\partial \theta_{d,l}}$ we have

$$\begin{aligned} \frac{\partial M_{rs}^{l'm,d}}{\partial \theta_{d,l}} &= \mathbb{E}_{x_d}\left[\frac{\partial (u_{\theta_{d,l'}}(x_d))_r}{\partial \theta_{d,l}} (u_{\theta_{d,m}}(x_d))_s + (u_{\theta_{d,l'}}(x_d))_r \frac{\partial (u_{\theta_{d,m}}(x_d))_s}{\partial \theta_{d,l}}\right] \\ &= \delta_{l'l} \mathbb{E}_{x_d}\left[(u_{\theta_{d,m}}(x_d))_s \frac{\partial (u_{\theta_{d,l'}}(x_d))_r}{\partial \theta_{d,l}}\right] + \delta_{ml} \mathbb{E}_{x_d}\left[(u_{\theta_{d,l'}}(x_d))_r \frac{\partial (u_{\theta_{d,m}}(x_d))_s}{\partial \theta_{d,l}}\right]. \end{aligned} \tag{25}$$

Substituting this expression yields

$$\begin{aligned} \frac{\partial \Sigma_{l'm}}{\partial \theta_{d,l}} &= \delta_{l'l} \sum_{r=1}^{R} \sum_{s=1}^{R} C_{rs}^{l'm,d}\, \mathbb{E}_{x_d}\left[(u_{\theta_{d,m}}(x_d))_s \frac{\partial (u_{\theta_{d,l'}}(x_d))_r}{\partial \theta_{d,l}}\right] \\ &\quad + \delta_{ml} \sum_{r=1}^{R} \sum_{s=1}^{R} C_{rs}^{l'm,d}\, \mathbb{E}_{x_d}\left[(u_{\theta_{d,l'}}(x_d))_r \frac{\partial (u_{\theta_{d,m}}(x_d))_s}{\partial \theta_{d,l}}\right], \end{aligned} \tag{26}$$

which is the claimed separable Jacobian formula. If the x_d expectations are estimated using 1D importance samples $x_{d,b} \sim \rho(x_d)$ with importance weights $w_{d,b} = 1/\rho(x_{d,b})$, then equation 20 holds natively. □

D.2 Approximation Ability of TPQ

We provide an approximation-capacity result for TPQ-based spectral decomposition by combining the tensor-product approximation property of variable-separable CP neural networks with the low-rank approximation characterization used in NeuralSVD (Ryu et al., 2024). This result formalizes that, when the leading spectral functions have mixed Sobolev regularity, the CP-separable TPQ ansatz can approximate both the spectral functions and the corresponding truncated spectral operator at an explicit rate with respect to the CP rank.

For the approximation argument, we work on the rectangular reference domain $\Omega = I^N$ with $I = [0, 2\pi]$; rectangular domains with finite side lengths can be reduced to this case by an affine rescaling. Let

$$\mathcal{A}_R := \left\{ h(\boldsymbol{x}) = \sum_{r=1}^{R} \prod_{d=1}^{N} h_{d,r}(x_d) : h_{d,r} \in L^2(I) \right\}$$

denote the idealized rank-R TPQ function class.

We first recall the mixed Sobolev space used in the tensor neural network approximation theory (Wang et al., 2024). For each one-dimensional variable $x_i \in I$, define the Fourier basis

$$\varphi_{k_i}(x_i) := (2\pi)^{-1/2} e^{\sqrt{-1} k_i x_i}, \ k_i \in \mathbb{Z}.$$

For a multi-index $\boldsymbol{k} = (k_1, \ldots, k_N) \in \mathbb{Z}^N$, define

$$\varphi_{\boldsymbol{k}}(\boldsymbol{x}) := \prod_{i=1}^{N} \varphi_{k_i}(x_i) = (2\pi)^{-N/2} e^{\sqrt{-1} \boldsymbol{k} \cdot \boldsymbol{x}}.$$

Furthermore, define

$$\lambda_{\mathrm{mix}}(\boldsymbol{k}) := \prod_{i=1}^{N} (1 + |k_i|), \ \lambda_{\mathrm{iso}}(\boldsymbol{k}) := 1 + \sum_{i=1}^{N} |k_i|.$$

For $t, \rho \in \mathbb{R}$, the mixed Sobolev space $H^{t,\rho}_{\mathrm{mix}}(I^N)$ is defined as

$$H^{t,\rho}_{\mathrm{mix}}(I^N) := \left\{ f(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathbb{Z}^N} c_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{x}) : \|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)} < \infty \right\},$$

where

$$\|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)} := \left(\sum_{\boldsymbol{k} \in \mathbb{Z}^N} \lambda_{\mathrm{mix}}(\boldsymbol{k})^{2t} \lambda_{\mathrm{iso}}(\boldsymbol{k})^{2\rho} |c_{\boldsymbol{k}}|^2 \right)^{1/2}.$$

Here t measures the dominating mixed smoothness and ρ measures the isotropic smoothness.

We also use the standard isotropic Sobolev norm of order m_0 on I^N, defined through the same Fourier basis by

$$\|f\|_{H^{m_0}(I^N)} := \left(\sum_{\boldsymbol{k} \in \mathbb{Z}^N} \lambda_{\mathrm{iso}}(\boldsymbol{k})^{2m_0} |c_{\boldsymbol{k}}|^2 \right)^{1/2}.$$

For $m_0 \geq 0$, this norm dominates the $L^2(I^N)$ norm.

We next detail the sparse tensor-product approximation space used by the tensor neural network approximation theory. For an integer $K \geq 1$ and a parameter $T \in (-\infty, 1]$, define the admissible Fourier index set

$$D_{K,T} := \left\{ \boldsymbol{k} \in \mathbb{Z}^N : \lambda_{\mathrm{mix}}(\boldsymbol{k}) \lambda_{\mathrm{iso}}(\boldsymbol{k})^{-T} \leq K^{1-T} \right\},$$

and the corresponding sparse Fourier space

$$V_{K,T} := \mathrm{span} \left\{ \varphi_{\boldsymbol{k}} : \boldsymbol{k} \in D_{K,T} \right\}.$$

We write $p_{K,T} := |D_{K,T}| = \dim(V_{K,T})$. For fixed N and $0 < T < 1$, Eq. (17) of (Wang et al., 2024) gives the upper bound $p_{K,T} = O(K + 1)$ for this admissible sparse-grid index set. The matching lower bound is elementary: if $\boldsymbol{k} = (n, 0, \ldots, 0)$ with $0 \leq n \leq K - 1$, then $\lambda_{\mathrm{mix}}(\boldsymbol{k}) =$

$\lambda_{\mathrm{iso}}(\boldsymbol{k}) = 1 + n$ and hence $\boldsymbol{k} \in D_{K,T}$. Therefore $p_{K,T} \geq K \geq (K+1)/2$. Thus there exist constants $0 < c_1 \leq c_2 < \infty$, depending only on N and T and independent of K, such that

$$c_1(K+1) \leq p_{K,T} \leq c_2(K+1).$$

The parameter T controls the shape of the retained frequency set: it interpolates between hyperbolic-cross-type and more isotropically weighted sparse tensor-product spaces. In the argument below, T is only an auxiliary parameter used to invoke the approximation theorem of (Wang et al., 2024), while the TPQ ansatz itself is parameterized by the rank R, which corresponds to the number $p_{K,T}$ of retained product Fourier modes.

Let $\mathcal{T} : L^2(\Omega) \to L^2(\Omega)$ be a compact Hilbert-Schmidt operator with kernel $K_{\mathcal{T}} \in L^2(\Omega \times \Omega)$, i.e.,

$$(\mathcal{T}f)(\boldsymbol{y}) = \int_{\Omega} K_{\mathcal{T}}(\boldsymbol{y}, \boldsymbol{x}) f(\boldsymbol{x})\, d\boldsymbol{x}.$$

Since $\mathcal{T}$ is compact and Hilbert-Schmidt, it admits the singular value decomposition

$$K_{\mathcal{T}}(\boldsymbol{y}, \boldsymbol{x}) = \sum_{j=1}^{\infty} \sigma_j v_j(\boldsymbol{y}) u_j(\boldsymbol{x}),\ \sigma_1 \geq \sigma_2 \geq \cdots \geq 0,$$

where $\{u_j\}_{j\geq 1}$ and $\{v_j\}_{j\geq 1}$ are orthonormal systems in $L^2(\Omega)$. Equivalently,

$$\mathcal{T}f = \sum_{j=1}^{\infty} \sigma_j v_j \langle u_j, f \rangle_{L^2(\Omega)}.$$

For a self-adjoint operator, we have $u_j = v_j$ up to signs and $\sigma_j = |\lambda_j|$. The theory is derived for compact Hilbert-Schmidt operators by following (Ryu et al., 2024) and is not a strict guarantee for unbounded differential operators such as Schrödinger Hamiltonians. For these physical problems, the approximation bound is intended as an approximation-capacity analogue that describes the role of the TPQ rank, while the numerical results provide further empirical evaluations.

Assumption 1 (Mixed regularity of leading spectral functions)**.** *We assume that there exist parameters $t > 0$, $\rho \in \mathbb{R}$, and an approximation norm order $m_0 \geq 0$ such that $0 < m_0 - \rho < t$, $s := t + \rho - m_0 > 0$, and for all $j = 1, \ldots, L$, $u_j, v_j \in H^{t,\rho}_{\mathrm{mix}}(I^N)$. These inequalities make the admissible interval required by Lemma 1 of (Wang et al., 2024),*

$$\mathcal{I}_T := \left(0, \frac{m_0 - \rho}{t}\right],$$

nonempty and contained in $(0, 1)$. We fix an arbitrary $T \in \mathcal{I}_T$ when applying the sparse Fourier approximation result in Lemma 2.

Lemma 2 (Sparse Fourier approximation, adapted from (Wang et al., 2024))**.** *Let $t > 0$, $\rho \in \mathbb{R}$, and $m_0 \geq 0$ satisfy the parameter inequalities in Assumption 1, and fix any $T \in \mathcal{I}_T$. For any $f \in H^{t,\rho}_{\mathrm{mix}}(I^N)$, let $f_{K,T}$ be the best approximation to f from $V_{K,T}$ with respect to the $H^{m_0}(I^N)$ norm. Then*

$$\|f - f_{K,T}\|_{H^{m_0}(I^N)} \leq C_N p_{K,T}^{-s} \|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)},\ s = t + \rho - m_0,$$

where C_N depends on the dimension and on the parameters of the approximation theorem, but not on K or $p_{K,T}$. Moreover, $f_{K,T}$ has a rank-$p_{K,T}$ separable Fourier representation

$$f_{K,T}(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in D_{K,T}} a_{\boldsymbol{k}} \varphi_{\boldsymbol{k}}(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in D_{K,T}} a_{\boldsymbol{k}} \prod_{d=1}^{N} \varphi_{k_d}(x_d).$$

Proof. This follows directly from Lemma 1 of (Wang et al., 2024), while the final display follows from the tensor-product identity $\varphi_{\boldsymbol{k}}(\boldsymbol{x}) = \prod_{d=1}^{N} \varphi_{k_d}(x_d)$. □

The following lemma transfers the sparse Fourier approximation into the idealized TPQ class $\mathcal{A}_R$.

Lemma 3 (TPQ approximation of spectral functions)**.** *Under Assumption 1, for each $j = 1, \dots, L$ and every integer rank $R \geq 1$, there exist TPQ functions $\tilde{u}_j, \tilde{v}_j \in \mathcal{A}_R$ such that*

$$\|u_j - \tilde{u}_j\|_{L^2(\Omega)} \leq C_N R^{-s} \|u_j\|_{H^{t,\rho}_{\mathrm{mix}}(\Omega)}, \ \|v_j - \tilde{v}_j\|_{L^2(\Omega)} \leq C_N R^{-s} \|v_j\|_{H^{t,\rho}_{\mathrm{mix}}(\Omega)},$$

where C_N may depend on N, t, ρ, m_0 and the fixed admissible T, but not on R.

Proof. Fix $T \in \mathcal{I}_T$. By Lemma 2, for every sparse index set $D_{K,T}$ with $p_{K,T} = |D_{K,T}|$, the best sparse Fourier approximation satisfies

$$\|f - f_{K,T}\|_{H^{m_0}(I^N)} \leq C_N^{\mathrm{sp}} p_{K,T}^{-s} \|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)}, \ s = t + \rho - m_0 > 0.$$

The approximant $f_{K,T}$ has the form

$$f_{K,T}(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in D_{K,T}} a_{\boldsymbol{k}} \prod_{d=1}^{N} \varphi_{k_d}(x_d),$$

which belongs to the TPQ class $\mathcal{A}_{p_{K,T}}$ after absorbing each scalar coefficient $a_{\boldsymbol{k}}$ into one of the one-dimensional factors. Since the spaces $\mathcal{A}_R$ are nested in R, the same approximation is feasible for any larger CP rank.

It remains to translate the estimate from the sparse-grid rank $p_{K,T}$ to an arbitrary CP rank R. The approximation theorem is naturally indexed by the auxiliary sparse-grid parameter K, and therefore gives bounds only for the admissible ranks $p_{K,T} = |D_{K,T}|$. Our TPQ statement, however, is formulated for any prescribed rank R. For a general $R \geq 1$, choose the largest integer $K_R \geq 1$ such that $p_{K_R,T} \leq R$; this integer exists because $p_{1,T} = 1$. The approximation using $p_{K_R,T}$ product terms belongs to $\mathcal{A}_R$ because $\mathcal{A}_{p_{K_R,T}} \subseteq \mathcal{A}_R$.

The only remaining issue is to express the rate in terms of the prescribed rank R, rather than the selected admissible rank $p_{K_R,T}$. This requires a lower bound on $p_{K_R,T}$ in terms of R; otherwise the estimate $C_N^{\mathrm{sp}} p_{K_R,T}^{-s}$ would not imply an R^{-s} estimate. By maximality,

$$R < p_{K_R+1,T} \leq c_2(K_R + 2),$$

while the lower cardinality bound gives

$$p_{K_R,T} \geq c_1(K_R + 1).$$

Since $K_R + 2 \leq 2(K_R + 1)$, these two inequalities imply

$$p_{K_R,T} \geq \frac{c_1}{2c_2} R.$$

With $c := c_1/(2c_2)$, we therefore have, for every $R \geq 1$,

$$p_{K_R,T}^{-s} \leq c^{-s} R^{-s}.$$

Thus, for every rank $R \geq 1$, the sparse-grid approximation chosen at level K_R yields an element of $\mathcal{A}_R$ and satisfies

$$\|f - f_{K_R,T}\|_{H^{m_0}(I^N)} \leq C_N^{\mathrm{sp}} c^{-s} R^{-s} \|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)}.$$

Finally, the H^{m_0} estimate implies the desired L^2 estimate. Indeed, since $0 < m_0 - \rho < t$, $\lambda_{\mathrm{iso}}(\boldsymbol{k}) \leq \lambda_{\mathrm{mix}}(\boldsymbol{k})$, and both weights are at least one,

$$\lambda_{\mathrm{iso}}(\boldsymbol{k})^{m_0-\rho} \leq \lambda_{\mathrm{iso}}(\boldsymbol{k})^t \leq \lambda_{\mathrm{mix}}(\boldsymbol{k})^t.$$

Multiplying both sides by $\lambda_{\mathrm{iso}}(\boldsymbol{k})^\rho$ gives

$$\lambda_{\mathrm{iso}}(\boldsymbol{k})^{m_0} \leq \lambda_{\mathrm{mix}}(\boldsymbol{k})^t \lambda_{\mathrm{iso}}(\boldsymbol{k})^\rho.$$

After squaring, multiplying by $|c_{\boldsymbol{k}}|^2$, and summing over $\boldsymbol{k}$, the mixed norm controls the isotropic norm:

$$\|f\|_{H^{m_0}(I^N)} \leq \|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)}.$$

Set $C_N := C_N^{\mathrm{sp}} c^{-s}$. Then C_N depends only on N, t, ρ, m_0 and the fixed T, and for every $R \geq 1$ there exists $f_R \in \mathcal{A}_R$ such that

$$\|f - f_R\|_{H^{m_0}(I^N)} \leq C_N R^{-s} \|f\|_{H^{t,\rho}_{\mathrm{mix}}(I^N)}.$$

Since the H^{m_0} norm dominates the L^2 norm for $m_0 \geq 0$, we have

$$\|f - f_R\|_{L^2(\Omega)} \leq \|f - f_R\|_{H^{m_0}(\Omega)} \leq C_N R^{-s} \|f\|_{H^{t,\rho}_{\mathrm{mix}}(\Omega)}.$$

Applying this estimate to $f = u_j$ and $f = v_j$ gives the result. □

We next lift the function-wise approximation bound to the operator level. Since $\mathcal{T}$ is Hilbert-Schmidt, its Hilbert-Schmidt norm is equivalent to the $L^2(\Omega \times \Omega)$ norm of its kernel:

$$\|\mathcal{T}\|_{\mathrm{HS}} = \|K_{\mathcal{T}}\|_{L^2(\Omega\times\Omega)}.$$

Define the TPQ-restricted rank-L LoRA approximation error as

$$\mathcal{E}^{\mathrm{TPQ}}_{R,L} := \inf_{\substack{f_j, g_j \in \mathcal{A}_R \\ j=1,\dots,L}} \left\| K_{\mathcal{T}}(\boldsymbol{y}, \boldsymbol{x}) - \sum_{j=1}^{L} g_j(\boldsymbol{y}) f_j(\boldsymbol{x}) \right\|_{L^2(\Omega\times\Omega)}.$$

The corresponding unrestricted rank-L error is

$$\mathcal{E}^{\star}_{L} := \inf_{\mathrm{rank}(\mathcal{S})\leq L} \|\mathcal{T} - \mathcal{S}\|_{\mathrm{HS}}.$$

By Schmidt's low-rank approximation theorem, equivalently Theorem 3.1 of (Ryu et al., 2024), the unrestricted optimum is the truncated spectral operator $\mathcal{T}_L$ with kernel

$$K_L(\boldsymbol{y}, \boldsymbol{x}) = \sum_{j=1}^{L} \sigma_j v_j(\boldsymbol{y}) u_j(\boldsymbol{x}),$$

and

$$\mathcal{E}^{\star}_{L} = \|\mathcal{T} - \mathcal{T}_L\|_{\mathrm{HS}} = \|K_{\mathcal{T}} - K_L\|_{L^2(\Omega\times\Omega)} = \left(\sum_{j>L} \sigma_j^2 \right)^{1/2}.$$

If $\sigma_L > \sigma_{L+1}$, this top-L spectral subspace is uniquely identified; otherwise, the same approximation error holds, while the singular functions inside degenerate subspaces are identifiable only up to an orthogonal rotation. We now give the explicit operator approximation error w.r.t. TPQ-rank R.

Theorem 1 (TPQ-restricted low-rank approximation error)**.** *Suppose Assumption 1 holds. Define*

$$M_j^u := \|u_j\|_{H^{t,\rho}_{\mathrm{mix}}(\Omega)}, M_j^v := \|v_j\|_{H^{t,\rho}_{\mathrm{mix}}(\Omega)}.$$

Then

$$\mathcal{E}^{\mathrm{TPQ}}_{R,L} \leq \left(\sum_{j>L} \sigma_j^2 \right)^{1/2} + C_N R^{-s} \sum_{j=1}^{L} \sigma_j (M_j^u + M_j^v) + C_N^2 R^{-2s} \sum_{j=1}^{L} \sigma_j M_j^u M_j^v.$$

In particular, for a self-adjoint EVD problem with normalized eigenfunctions $\{u_j\}_{j=1}^{\infty}$ and eigenvalues $\{\lambda_j\}_{j=1}^{\infty}$ ordered by nonincreasing magnitude, this becomes

$$\mathcal{E}^{\mathrm{TPQ}}_{R,L} \leq \left(\sum_{j>L} |\lambda_j|^2 \right)^{1/2} + 2C_N R^{-s} \sum_{j=1}^{L} |\lambda_j| \|u_j\|_{H^{t,\rho}_{\mathrm{mix}}(\Omega)} + C_N^2 R^{-2s} \sum_{j=1}^{L} |\lambda_j| \|u_j\|^2_{H^{t,\rho}_{\mathrm{mix}}(\Omega)}.$$

Proof. By Lemma 3, for each $j \leq L$ there exist $\tilde{u}_j, \tilde{v}_j \in \mathcal{A}_R$ such that

$$\delta_j^u := \|u_j - \tilde{u}_j\|_{L^2(\Omega)} \leq C_N R^{-s} M_j^u,\ \delta_j^v := \|v_j - \tilde{v}_j\|_{L^2(\Omega)} \leq C_N R^{-s} M_j^v.$$

Fubini's theorem gives

$$\|a(\boldsymbol{y}) b(\boldsymbol{x})\|_{L^2(\Omega\times\Omega)} = \|a\|_{L^2(\Omega)} \|b\|_{L^2(\Omega)}$$

for $a, b \in L^2(\Omega)$. Since u_j and v_j are normalized in $L^2(\Omega)$, the triangle inequality also gives $\|\tilde{v}_j\|_{L^2(\Omega)} \le \|v_j\|_{L^2(\Omega)} + \|v_j - \tilde{v}_j\|_{L^2(\Omega)} = 1 + \delta_j^v$. In the following display, $\|\cdot\|_2$ denotes the $L^2(\Omega \times \Omega)$ norm. Therefore,

$$\begin{aligned}\|v_j(\boldsymbol{y})u_j(\boldsymbol{x}) - \tilde{v}_j(\boldsymbol{y})\tilde{u}_j(\boldsymbol{x})\|_2 &\le \big\|\big(v_j(\boldsymbol{y}) - \tilde{v}_j(\boldsymbol{y})\big)u_j(\boldsymbol{x})\big\|_2 + \big\|\tilde{v}_j(\boldsymbol{y})\big(u_j(\boldsymbol{x}) - \tilde{u}_j(\boldsymbol{x})\big)\big\|_2 \\ &\le \delta_j^v + \|\tilde{v}_j\|_{L^2(\Omega)}\,\delta_j^u \le \delta_j^v + (1+\delta_j^v)\delta_j^u \\ &= \delta_j^u + \delta_j^v + \delta_j^u\delta_j^v.\end{aligned}$$

Now construct a feasible TPQ rank-L kernel

$$\widetilde{K}_{R,L}(\boldsymbol{y}, \boldsymbol{x}) := \sum_{j=1}^{L} \sigma_j \tilde{v}_j(\boldsymbol{y})\tilde{u}_j(\boldsymbol{x}).$$

This is feasible in the definition of $\mathcal{E}_{R,L}^{\mathrm{TPQ}}$ by taking $g_j = \sigma_j\tilde{v}_j$ and $f_j = \tilde{u}_j$, since $\mathcal{A}_R$ is closed under scalar multiplication. Hence,

$$\mathcal{E}_{R,L}^{\mathrm{TPQ}} \le \left\|K_{\mathcal{T}} - \widetilde{K}_{R,L}\right\|_{L^2(\Omega\times\Omega)}.$$

Using the triangle inequality and the definition of K_L,

$$\begin{aligned}\left\|K_{\mathcal{T}} - \widetilde{K}_{R,L}\right\|_{L^2(\Omega\times\Omega)} &\le \|K_{\mathcal{T}} - K_L\|_{L^2(\Omega\times\Omega)} + \left\|K_L - \widetilde{K}_{R,L}\right\|_{L^2(\Omega\times\Omega)} \\ &\le \left(\sum_{j>L}\sigma_j^2\right)^{1/2} + \sum_{j=1}^{L}\sigma_j\,\|v_j(\boldsymbol{y})u_j(\boldsymbol{x}) - \tilde{v}_j(\boldsymbol{y})\tilde{u}_j(\boldsymbol{x})\|_{L^2(\Omega\times\Omega)} \\ &\le \left(\sum_{j>L}\sigma_j^2\right)^{1/2} + \sum_{j=1}^{L}\sigma_j(\delta_j^u + \delta_j^v + \delta_j^u\delta_j^v).\end{aligned}$$

Substituting the bounds for δ_j^u and δ_j^v gives the stated estimate. The self-adjoint case follows by taking $u_j = v_j$ and $\sigma_j = |\lambda_j|$. □

Theorem 1 is an approximation-capacity result. It states that the TPQ ansatz approaches the unrestricted NeuralSVD low-rank approximation as the CP rank increases, provided that the leading spectral functions admit mixed-Sobolev tensor-product approximation. The first term is the unavoidable spectral truncation error, while the remaining two terms are the additional cost of representing the leading spectral functions by rank-R TPQ functions.

Remark 2. *The mixed-regularity assumption is the essential condition in this argument. It should not be interpreted as saying that every spectral function of every operator automatically satisfies the hypotheses of Lemma 1 of (Wang et al., 2024). Rather, the theorem shows that when the leading spectral functions belong to, or are well approximated by, the mixed Sobolev tensor-product class $H^{t,\rho}_{\mathrm{mix}}$, TPQ admits an explicit rank-dependent approximation rate. For operators with singular potentials or cusp singularities, the mixed-Sobolev norm may be large or finite only at low smoothness; in such cases the bound still explains the role of TPQ rank, but the rate may be weaker.*

D.3 Variance of MC-based Numerical Integral

In the neural operator EVD/SVD framework, the core computation within the objective function involves evaluating the continuous inner product between parameterized eigenfunctions or singular functions. In this section, we evaluate the variance of standard MC sampling to approximate such inner products. Given the target functions $\phi^l(\boldsymbol{x})$ and $\phi^m(\boldsymbol{x})$, their inner product over a spatial domain $\Omega \subset \mathbb{R}^N$ is defined as

$$I = \langle \phi^l, \phi^m \rangle = \int_{\Omega} \phi^l(\boldsymbol{x})\phi^m(\boldsymbol{x})d\boldsymbol{x}.$$

To optimize this integral via automatic differentiation in neural networks, existing parametric methods employ importance sampling (Deng et al., 2022; Ryu et al., 2024). By introducing a proposal

distribution $\rho(\boldsymbol{x})$ that is positive on the support of the integrand, the integral can be reformulated as an expectation

$$I = \mathbb{E}_{\boldsymbol{x}\sim\rho}\left[\phi^l(\boldsymbol{x})\phi^m(\boldsymbol{x})W(\boldsymbol{x})\right],$$

where $W(\boldsymbol{x}) = \frac{1}{\rho(\boldsymbol{x})}$ represents the importance weight. This is the generic unbiased importance-sampling form. In our PyTorch implementation in Sec. J, the square root of the proposal density is absorbed into the factors used by the finite-difference operator, acting as an implicit envelope rather than appearing as an explicit inverse-density weight. For brevity, we define $h(\boldsymbol{x}) := \phi^l(\boldsymbol{x})\phi^m(\boldsymbol{x})W(\boldsymbol{x})$. In each training iteration, the mini-batch MC method draws a mini-batch of i.i.d. samples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(B)}\} \sim \rho(\boldsymbol{x})$ of size B. The MC empirical estimator is then defined as

$$\hat{I}_B = \frac{1}{B}\sum_{b=1}^{B} h(\boldsymbol{x}^{(b)}).$$

Because the samples are i.i.d., this estimator is unbiased $\mathbb{E}[\hat{I}_B] = I$. Assuming the intrinsic variance of a single sample is a finite constant $\mathrm{Var}(h(\boldsymbol{x})) = \sigma^2 < \infty$, the variance of the MC estimator is given by

$$\mathrm{Var}(\hat{I}_B) = \mathrm{Var}\left(\frac{1}{B}\sum_{b=1}^{B} h(\boldsymbol{x}^{(b)})\right) = \frac{1}{B^2}\sum_{b=1}^{B}\mathrm{Var}(h(\boldsymbol{x}^{(b)})) = \frac{\sigma^2}{B}.$$

Hence, the variance of the MC integral estimator scales as $\mathcal{O}(\sigma^2/B)$. Since the estimator is unbiased, its root mean square error (RMSE) equals its standard deviation and scales as $\mathcal{O}(\sigma/\sqrt{B})$:

$$\mathrm{RMSE}(\hat{I}_B) = \sqrt{\mathbb{E}\left[(\hat{I}_B - I)^2\right]} = \sqrt{\mathrm{Var}(\hat{I}_B)} = \frac{\sigma}{\sqrt{B}}.$$

The calculation below illustrates that, even in a separable setting, the single-sample variance σ^2 can grow exponentially with the spatial dimension N. In such cases, an exponentially large batch size B is needed to maintain a small MC error $\sigma/\sqrt{B}$.

D.4 Variance Explosion Rate of High-dimensional Function

Suppose we perform MC integration in an N-dimensional space $\Omega \subset \mathbb{R}^N$. To facilitate analysis, assume that both the weighted integrand $h(\boldsymbol{x})$ and the proposal density $\rho(\boldsymbol{x})$ factorize across dimensions:

$$h(\boldsymbol{x}) = \prod_{d=1}^{N} h_d(x_d),\ \rho(\boldsymbol{x}) = \prod_{d=1}^{N}\rho_d(x_d).$$

For simplicity, assume the one-dimensional factors have the same mean and variance under their corresponding proposal densities:

$$\mathbb{E}_{x_d\sim\rho_d}[h_d(x_d)] = \mu,\ \mathrm{Var}_{x_d\sim\rho_d}(h_d(x_d)) = \gamma^2 > 0.$$

Consequently, the second moment of the 1D function is given by $\mathbb{E}[h_d^2] = \mu^2 + \gamma^2$. We now compute the intrinsic variance of a single MC sample in the full N-dimensional space, denoted as $\sigma_N^2 = \mathrm{Var}_{\boldsymbol{x}\sim\rho}(h(\boldsymbol{x}))$. Due to dimensional independence, the multi-dimensional expectation is the product of the 1D expectations:

$$\mathbb{E}[h(\boldsymbol{x})] = \prod_{d=1}^{N}\mathbb{E}[h_d(x_d)] = \mu^N.$$

Similarly, the multi-dimensional second moment is the product of the 1D second moments:

$$\mathbb{E}[h(\boldsymbol{x})^2] = \prod_{d=1}^{N}\mathbb{E}[h_d(x_d)^2] = (\mu^2 + \gamma^2)^N.$$

Using the definition of variance $\mathrm{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$, we obtain the intrinsic variance for N-dimensional sampling:

$$\mathrm{Var}[h(\boldsymbol{x})] = \sigma_N^2 = (\mu^2 + \gamma^2)^N - \mu^{2N}.$$

When $\mu \neq 0$, the relative variance is

$$\frac{\sigma_N^2}{\mathbb{E}[h(\boldsymbol{x})]^2} = \frac{(\mu^2+\gamma^2)^N - \mu^{2N}}{\mu^{2N}} = \left(1+\frac{\gamma^2}{\mu^2}\right)^N - 1.$$

Thus, whenever $\gamma^2 > 0$ and $\mu \neq 0$, the base $1+\gamma^2/\mu^2$ is larger than one, and the relative variance grows exponentially with N. If $\mu = 0$, the relative variance is not defined, but the absolute variance is still $\sigma_N^2 = \gamma^{2N}$. For ordinary smooth functions, the relative-variance base may be close to one, making the growth mild. However, operator decomposition seeks ordered, orthogonal eigenfunctions. Higher-order eigenfunctions, such as excited states in the Schrödinger equation, can have strong oscillations across the spatial domain, increasing the one-dimensional variance γ^2 and making the one-dimensional mean small. This makes the variance more severe and requires larger batches to keep the MC error $\sigma/\sqrt{B}$ small. TPQ provides an implicit tensor product batch size B_{1D}^N while evaluating only one-dimensional factors at cost linear in N, which alleviates this computation-accuracy trade-off, as detailed below.

D.5 Variance Comparison Between Direct MC and TPQ Estimators

We now reuse the variance calculation in Sec. D.3 to compare direct MC with TPQ under a product structure. Let $\Omega = \prod_{d=1}^N \Omega_d$, and assume that the integrand and proposal density factorize as

$$h(\boldsymbol{x}) = \prod_{d=1}^N h_d(x_d),\ \rho(\boldsymbol{x}) = \prod_{d=1}^N \rho_d(x_d).$$

For each dimension, define

$$\mu_d := \mathbb{E}_{x_d\sim\rho_d}[h_d(x_d)],\ \nu_d^2 := \mathrm{Var}_{x_d\sim\rho_d}[h_d(x_d)].$$

Then the target expectation is

$$I = \mathbb{E}_{\boldsymbol{x}\sim\rho}[h(\boldsymbol{x})] = \prod_{d=1}^N \mu_d.$$

For direct MC, drawing B i.i.d. full-dimensional samples $\boldsymbol{x}^{(b)} \sim \rho$ gives

$$\widehat{I}_{\mathrm{MC}} = \frac{1}{B}\sum_{b=1}^B h(\boldsymbol{x}^{(b)}).$$

By the result of Sec. D.3,

$$\mathrm{Var}(\widehat{I}_{\mathrm{MC}}) = \frac{\sigma_N^2}{B},\ \sigma_N^2 = \mathrm{Var}_{\boldsymbol{x}\sim\rho}[h(\boldsymbol{x})] = \prod_{d=1}^N(\mu_d^2+\nu_d^2) - \prod_{d=1}^N \mu_d^2. \tag{27}$$

For TPQ, we draw independent one-dimensional sample sets $\{x_{d,b}\}_{b=1}^{B_{\mathrm{1D}}} \sim \rho_d$ and estimate each factor mean by

$$\widehat{\mu}_d = \frac{1}{B_{\mathrm{1D}}}\sum_{b=1}^{B_{\mathrm{1D}}} h_d(x_{d,b}).$$

The TPQ estimator is the product of these one-dimensional estimates:

$$\widehat{I}_{\mathrm{TPQ}} = \prod_{d=1}^N \widehat{\mu}_d.$$

Since $\mathbb{E}[\widehat{\mu}_d] = \mu_d$, and the one-dimensional sample sets are independent across dimensions, TPQ is unbiased:

$$\mathbb{E}[\widehat{I}_{\mathrm{TPQ}}] = \mathbb{E}\left[\prod_{d=1}^N \widehat{\mu}_d\right] = \prod_{d=1}^N \mathbb{E}[\widehat{\mu}_d] = \prod_{d=1}^N \mu_d = I.$$

Moreover, each $\widehat{\mu}_d$ is an average of $B_{1\mathrm{D}}$ i.i.d. one-dimensional samples, so

$$\mathrm{Var}(\widehat{\mu}_d) = \frac{\nu_d^2}{B_{1\mathrm{D}}},\ \mathbb{E}[\widehat{\mu}_d^2] = \mathrm{Var}(\widehat{\mu}_d) + \mathbb{E}[\widehat{\mu}_d]^2 = \mu_d^2 + \frac{\nu_d^2}{B_{1\mathrm{D}}}.$$

Using independence across dimensions again, the second moment of the TPQ estimator is

$$\mathbb{E}[\widehat{I}_{\mathrm{TPQ}}^2] = \mathbb{E}\left[\prod_{d=1}^{N} \widehat{\mu}_d^2\right] = \prod_{d=1}^{N} \mathbb{E}[\widehat{\mu}_d^2] = \prod_{d=1}^{N} \left(\mu_d^2 + \frac{\nu_d^2}{B_{1\mathrm{D}}}\right).$$

Therefore,

$$\mathrm{Var}(\widehat{I}_{\mathrm{TPQ}}) = \mathbb{E}[\widehat{I}_{\mathrm{TPQ}}^2] - \mathbb{E}[\widehat{I}_{\mathrm{TPQ}}]^2 = \prod_{d=1}^{N} \left(\mu_d^2 + \frac{\nu_d^2}{B_{1\mathrm{D}}}\right) - \prod_{d=1}^{N} \mu_d^2. \tag{28}$$

Equivalently, TPQ forms the Cartesian-product estimator using $B_{1\mathrm{D}}^N$ effective batch size

$$\widehat{I}_{\mathrm{TPQ}} = \frac{1}{B_{1\mathrm{D}}^N} \sum_{b_1=1}^{B_{1\mathrm{D}}} \cdots \sum_{b_N=1}^{B_{1\mathrm{D}}} \prod_{d=1}^{N} h_d(x_{d,b_d}),$$

but the $B_{1\mathrm{D}}^N$ product terms are not independent because they reuse the same $B_{1\mathrm{D}}N$ one-dimensional samples. Hence equation 28, rather than $\sigma_N^2/B_{1\mathrm{D}}^N$, is the exact TPQ variance.

We now compare direct MC and TPQ under the same nominal tensor-product batch size $B = B_{1\mathrm{D}}^N$. From equation 27, direct MC gives

$$\mathrm{Var}(\widehat{I}_{\mathrm{MC}}) = \frac{1}{B_{1\mathrm{D}}^N} \left[\prod_{d=1}^{N} (\mu_d^2 + \nu_d^2) - \prod_{d=1}^{N} \mu_d^2\right], \tag{29}$$

whereas TPQ gives the product-estimator variance

$$\mathrm{Var}(\widehat{I}_{\mathrm{TPQ}}) = \prod_{d=1}^{N} \left(\mu_d^2 + \frac{\nu_d^2}{B_{1\mathrm{D}}}\right) - \prod_{d=1}^{N} \mu_d^2. \tag{30}$$

For the homogeneous case $\mu_d = \mu$ and $\nu_d^2 = \gamma^2$, the two absolute variances reduce to

$$\mathrm{Var}(\widehat{I}_{\mathrm{MC}}) = \frac{(\mu^2 + \gamma^2)^N - \mu^{2N}}{B_{1\mathrm{D}}^N},\ \mathrm{Var}(\widehat{I}_{\mathrm{TPQ}}) = \left(\mu^2 + \frac{\gamma^2}{B_{1\mathrm{D}}}\right)^N - \mu^{2N}.$$

These formulas show that direct MC averages independent full-dimensional products, while TPQ averages one-dimensional factors and then reuses them to assemble the tensor-product estimator. The two variances are not identical in general. However, for highly oscillatory functions such as higher-order eigenfunctions, the one-dimensional mean is typically close to zero relative to its variance ($\mu^2 \ll \gamma^2$). By factoring out γ^{2N}, we can express the two variances as:

$$\mathrm{Var}(\widehat{I}_{\mathrm{MC}}) = \frac{\gamma^{2N}}{B_{1\mathrm{D}}^N} \left[\left(1 + \frac{\mu^2}{\gamma^2}\right)^N - \left(\frac{\mu^2}{\gamma^2}\right)^N\right],$$

$$\mathrm{Var}(\widehat{I}_{\mathrm{TPQ}}) = \frac{\gamma^{2N}}{B_{1\mathrm{D}}^N} \left[\left(1 + \frac{B_{1\mathrm{D}}\mu^2}{\gamma^2}\right)^N - \left(\frac{B_{1\mathrm{D}}\mu^2}{\gamma^2}\right)^N\right].$$

When the mean approaches zero ($\mu \to 0$), both variances asymptotically converge to the exact same leading term $\gamma^{2N}/B_{1\mathrm{D}}^N$. To see this rigorously for small but non-zero means, assume μ^2 is small enough such that $B_{1\mathrm{D}}\mu^2/\gamma^2 \ll 1$ (which inherently implies $\mu^2/\gamma^2 \ll 1$ since $B_{1\mathrm{D}} \geq 1$). Applying the first-order binomial expansion $(1+x)^N = 1 + Nx + \mathcal{O}(x^2)$, the bracketed terms expand as:

$$\mathrm{Var}(\widehat{I}_{\mathrm{MC}}) = \frac{\gamma^{2N}}{B_{1\mathrm{D}}^N} \left[1 + N\frac{\mu^2}{\gamma^2} + \mathcal{O}\left(\frac{\mu^4}{\gamma^4}\right) - \left(\frac{\mu^2}{\gamma^2}\right)^N\right],$$

$$\mathrm{Var}(\widehat{I}_{\mathrm{TPQ}}) = \frac{\gamma^{2N}}{B_{1\mathrm{D}}^N} \left[1 + N\frac{B_{1\mathrm{D}}\mu^2}{\gamma^2} + \mathcal{O}\left(\frac{B_{1\mathrm{D}}^2\mu^4}{\gamma^4}\right) - \left(\frac{B_{1\mathrm{D}}\mu^2}{\gamma^2}\right)^N\right].$$

For high-dimensional problems ($N \geq 2$), the subtracted N-th power terms are absorbed into the higher-order $\mathcal{O}$ terms. Because the remaining linear terms $N\mu^2/\gamma^2$ and $NB_{1\mathrm{D}}\mu^2/\gamma^2$ are strictly bounded and vanishingly small under this condition, the zeroth-order constant 1 would dominate the brackets. Thus, both variances are dominated by the identical $\gamma^{2N}/B_{1\mathrm{D}}^N$ leading term in this case.

Under this oscillatory condition, $\mathrm{Var}(\widehat{I}_{\mathrm{TPQ}})$ matches the scale of $\mathrm{Var}(\widehat{I}_{\mathrm{MC}})$ at $B = B_{1\mathrm{D}}^N$, making TPQ highly computationally preferable because it achieves similar variance under only $O(B_{1\mathrm{D}}N)$ complexity. Its reusable one-dimensional integrations require only $B_{1\mathrm{D}}N$ factor evaluations, whereas direct MC requires $B_{1\mathrm{D}}^N$ evaluations of the full N-dimensional integrand. This is why TPQ alleviates the computational-accuracy trade-off of existing neural operator decomposition methods: it evaluates a massive $B_{1\mathrm{D}}^N$ effective batch size at the linear computational cost of 1D quadratures.

Remark 3 (Scope and practical implication of the variance analysis)**.** *The preceding near-zero-mean analysis characterizes an important limiting regime, particularly for rank-one or factorwise centered cross terms, rather than all mass and stiffness entries encountered during training. For example, off-diagonal inner products between oscillatory or nearly orthogonal spectral components may have small factor means, whereas diagonal mass and stiffness entries generally do not. Moreover, a complete CP-assembled matrix entry is a sum of multiple product estimators whose rank-pair terms reuse the same one-dimensional samples and may therefore be statistically correlated. Accordingly, the nominal Cartesian-product size $B_{1\mathrm{D}}^N$ should not be interpreted as $B_{1\mathrm{D}}^N$ independent MC samples, and the direct $\mathcal{O}(\sigma^2/B_{1\mathrm{D}}^N)$ variance scale derived in Sec. D.3 is not claimed for the TPQ estimate. Rather, the analysis identifies a favorable regime in which TPQ can attain the leading variance scale of direct MC with $B_{1\mathrm{D}}^N$ full-dimensional samples using only $NB_{1\mathrm{D}}$ one-dimensional factor evaluations.*

More generally, the practical advantage of TPQ arises from exploiting separability: one-dimensional estimates are computed once and reused to assemble all tensor-product contributions, avoiding repeated full-dimensional network evaluations. In this way, TPQ can allow a much larger effective tensor-product batch size without incurring the exponential computational overhead. This computation–accuracy advantage is also supported empirically by our experiments, rather than being inferred solely from the limiting variance analysis above.

E Distinctions Between the Proposed Method and Tensor Neural Network Method

We remark that there are related works utilizing tensor neural network (TNN)-based integral methods for obtaining multi-eigenpairs of partial differential equations (PDEs) (Wang et al., 2024; Wang & Xie, 2024). These methods and our TPQ framework share the motivation of using variable-separable tensor structures to bypass the curse of dimensionality in high-dimensional integration. However, they use this separability in different algorithmic roles: TNN constructs a finite tensor basis and extracts eigenpairs through a Rayleigh-Ritz projection, whereas our method uses tensor product quadrature as an integration module inside neural EVD/SVD objectives such as SpIN, NeuralEF, and NeuralSVD. The distinctions lie primarily in the following three aspects:

Numerical Stability (Avoiding Matrix Inversions) The most critical distinction lies in how the functional subspace is optimized and how eigenvalues are extracted. The TNN method (Wang & Xie, 2024) relies on a Rayleigh-Ritz-style formulation, which constructs a tensor basis and involves algebraic operations with the mass matrix, e.g., computing $\boldsymbol{B}^{-1}\boldsymbol{A}$ by solving linear systems, and solving the projected generalized eigenvalue problem $\boldsymbol{A}\boldsymbol{y} = \lambda\boldsymbol{B}\boldsymbol{y}$. In the context of solving complex PDEs or performing Galerkin projections, if the learned basis is nearly linearly dependent, the mass matrix $\boldsymbol{B}$ can become ill-conditioned, and computing matrix inverses or generalized Ritz projections may introduce numerical instability. Thus, the accuracy of TNN depends not only on the expressivity of the learned tensor factors, but also on the numerical rank and conditioning of $\boldsymbol{B}$. In contrast, by integrating TPQ with frameworks like NeuralSVD or NeuralEF, our method optimizes trace-based or matrix-product-based quadratic objectives and directly parameterizes the target spectral functions. This avoids unstable algebraic operations such as backpropagation through matrix inversion or generalized eigensolvers during training and gives a more stable optimization path for constructing continuous eigenspace representations.

Broader Applicability for Operator SVD The TNN method in (Wang & Xie, 2024) is designed for eigenvalue decomposition, specifically, the eigenpairs of self-adjoint operators. Our TPQ method generalizes beyond this limitation by combining with the NeuralSVD framework (Ryu et al., 2024). Consequently, our method is mathematically equipped to perform singular value decomposition on non-self-adjoint operators and asymmetric integral kernel operators. This may broaden the applicability of our method to a wider class of dynamical systems and machine learning tasks beyond eigenvalue decomposition.

Algorithmic Flexibility: Pure Optimization Framework The proposed TPQ method benefits from an end-to-end neural optimization framework. The TNN approach (Wang & Xie, 2024) is a hybrid method: after learning a tensor basis, it still explicitly projects the operator onto this finite-dimensional subspace and solves a generalized algebraic eigenvalue problem to extract the final eigenpairs. Our method, conversely, uses TPQ to assemble the function integrals required by existing neural spectral objectives, so the network outputs themselves are trained to represent ordered eigenfunctions or singular functions. In NeuralEF-TPQ, for instance, the reported eigenvalues can be obtained directly from the learned outputs through Rayleigh quotients, while the off-diagonal penalties encourage orthogonality during training. This makes TPQ a general quadrature-and-assembly module rather than a standalone Rayleigh-Ritz eigensolver. It can be plugged into SpIN, NeuralEF, and NeuralSVD, supports non-self-adjoint operator SVD, and incorporates a parametric separable surrogate for non-separable potentials. The suggested parametric separable surrogate (Gaussian mixture) serves as a novel numerical technique to bypass the non-separability and singularity of the Coulomb potential function.

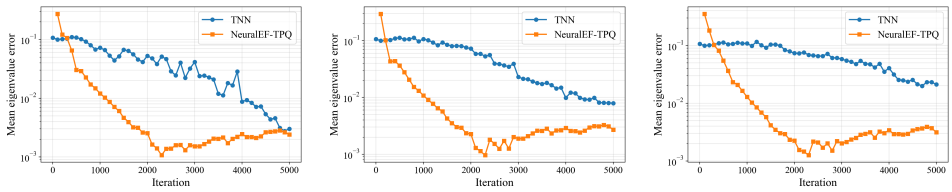


Figure 8: Convergence curves of TNN and NeuralEF-TPQ for the 2D infinite well experiment with 12 targeted eigenpairs, 15 eigenpairs, and 18 eigenpairs (left, middle, and right).

Figure 8 provides a numerical example to further compare the proposed method (e.g., NeuralEF-TPQ) with the TNN-based multi-eigenpair solver (Wang & Xie, 2024) on the 2D infinite well under matched neural network capacity. For TNN, we report the Rayleigh-Ritz eigenvalues projected onto the learned tensor basis, whereas NeuralEF-TPQ reports the eigenvalues directly from its learned eigenfunction outputs using the Rayleigh quotient. NeuralEF-TPQ exhibits more stable convergence, and the advantage becomes more evident as the number of targeted eigenpairs increases. This suggests that the TPQ-based NeuralEF objective provides a more robust representation of the eigenspace, while the TNN solver may be more sensitive to the expressivity and conditioning of the learned tensor basis used in the Ritz projection.

F Problem Formulation

All experiments are conducted on a platform with an RTX 2080TI GPU and 24 GB system memory.

F.1 Time-independent Schrödinger Equation

Following the operator decomposition formulation introduced in the main text, each time-independent Schrödinger experiment is posed as an eigenvalue decomposition problem for the Hamiltonian $\mathcal{H} = -\Delta + V(\boldsymbol{x})$: we seek eigenpairs $\{E_l, u_l\}_{l=1}^{L}$ satisfying $\mathcal{H}u_l = E_l u_l$ under the corresponding domain and boundary conditions. Since the neural EVD objectives are formulated as maximization problems, we optimize the sign-flipped operator $\mathcal{T} = -\mathcal{H}$, so that $\mathcal{T}u_l = -E_l u_l$ and the low-energy physical states become the leading eigenmodes of $\mathcal{T}$. SpIN, NeuralEF, NeuralSVD, and their TPQ variants are compared for this common EVD task.

This section gives the mathematical problem definitions, numerical discretizations, and experimental protocols used in our empirical study. For each comparison between SpIN, NeuralEF, NeuralSVD

and their TPQ variants, we choose sampling batch size so that the original dense and TPQ methods are compared under comparable iterations per second (i.e., optimization speed). The main computational bottleneck of the original multilayer perceptron (MLP)-based baseline is the rapid increase in memory consumption with larger batch sizes, which can lead to out-of-memory (OOM) errors. We therefore tune the TPQ batch size so that TPQ and the MLP-based baseline achieve comparable iteration speeds for a fair comparison. Because TPQ evaluates 1D factors and then contracts tensor-product quantities, B_{1D} samples per coordinate correspond to an effective tensor product quadrature size B_{1D}^N. Hence, TPQ allows generally much larger effective batch size (see Tables 4-6 for our detailed settings[4]). The main settings are summarized below.

Throughout this appendix, N denotes the ambient dimension, L the number of target spectral modes, R the CP separation rank, and $\boldsymbol{x} = (x_1, \ldots, x_N)$ and $\boldsymbol{y} = (y_1, \ldots, y_N)$ denote input coordinates. The indices d, (l, m), r, s, and b index dimensions, spectral modes, rank components, and quadrature samples, respectively. We use B for the number of dense Monte Carlo samples and B_{1D} for the number of samples per coordinate in TPQ.

For the Schrödinger equation experiments, we follow the operator convention:

$$\mathcal{H} = -\Delta + V(\boldsymbol{x}), \qquad \mathcal{T} = -\mathcal{H} = \Delta - V(\boldsymbol{x}). \tag{31}$$

Here $\Delta = \sum_{d=1}^{N} \partial^2/\partial x_d^2$ is the Laplacian, $V(\boldsymbol{x})$ is the potential, $\mathcal{H}$ is the physical Hamiltonian, and $\mathcal{T}$ is the sign-flipped operator optimized by the neural EVD objectives. If $\mathcal{H}u = Eu$, then $\mathcal{T}u = -Eu$. Therefore all EVD objectives are applied to $\mathcal{T}$, and the eigenvalues reported by the neural solvers are compared with the negated analytical energies under the same affine scaling

$$\lambda_{\mathrm{truth}} = s_{\mathrm{op}}(-E) + c_{\mathrm{op}}. \tag{32}$$

In this affine comparison, E denotes an analytical energy of $\mathcal{H}$, while λ_{truth} denotes the corresponding target eigenvalue after the sign flip, optional scale s_{op}, and shift c_{op}. For the harmonic oscillator and infinite well, $s_{\mathrm{op}} = 1$, with shifts $c_{\mathrm{op}} = 16$ and $c_{\mathrm{op}} = 50$, respectively. For hydrogen, $s_{\mathrm{op}} = 100$ and $c_{\mathrm{op}} = 0$. The shift does not alter eigenfunctions; it is used to keep the optimized spectrum in a numerically convenient range by following (Ryu et al., 2024).

All Schrödinger experiments use a single-particle coordinate $\boldsymbol{x} \in \mathbb{R}^N$. Dense MLP baselines approximate the required integrals with B full N-D samples, whereas TPQ draws independent 1D marginal samples $\{x_{d,b}\}_{b=1}^{B_{\mathrm{1D}}}$ for each coordinate and induces an effective product quadrature of size B_{1D}^N. With Gaussian sampling, the proposal density is factorized as

$$\rho(\boldsymbol{x}) = \prod_{d=1}^{N} \rho_d(x_d), \;\; \rho_d(x) = \frac{1}{s\sqrt{2\pi}} \exp\!\left(-\frac{x^2}{2s^2}\right). \tag{33}$$

Here ρ is the N-D proposal density, ρ_d is its 1D marginal on coordinate d, and s is the Gaussian sampling scale. In the reported configurations, dense MLP baselines use direct full-dimensional Gaussian sampling, whereas TPQ configurations with tensor-integral assembly use Gaussian grid sampling as in equation 33, i.e., independent 1D Gaussian draws in each dimension, with the resulting 1D factor integrals assembled through tensor product contractions. Accordingly, the importance weights and all TPQ mass/stiffness terms are assembled from 1D factors. We sample $x_{d,b} \sim \mathcal{N}(0, s^2)$ and use the Cartesian product of the 1D samples as the integration rule, while the neural networks are evaluated only on the 1D factors before tensor contractions. For harmonic and hydrogen problems, validation integrals are approximated on uniform grids over $[-\ell, \ell]^N$, with density $1/(2\ell)^N$; for the infinite well, validation is performed on the physical box $[0, l]^2$ (detailed below).

The Laplacian is discretized by a central finite-difference stencil. For a dense MLP $u_l(\boldsymbol{x})$, we sum the coordinate-wise finite differences of the full network. For TPQ, the same stencil is applied to each 1D factor,

$$\frac{\partial^2 (u_{\theta_{d,l}}(x_d))_r}{\partial x_d^2} \approx \frac{(u_{\theta_{d,l}}(x_d + \epsilon))_r + (u_{\theta_{d,l}}(x_d - \epsilon))_r - 2(u_{\theta_{d,l}}(x_d))_r}{\epsilon^2}. \tag{34}$$

[4]For SpIN, the Jacobian computation limits TPQ's ability to use a substantially larger batch size in 2D. Nevertheless, TPQ can use a larger batch size in 3D and achieves competitive performance in both the 2D and 3D cases.

In this notation, u_l is the l-th learned spectral function, $u_{\theta_{d,l}}$ is its 1D factor network along dimension d, $(\cdot)_r$ selects the r-th rank component, and ϵ is the finite-difference step. These derivatives are then inserted into the stiffness assembly described in Sec. 2.1.2. Thus the operator approximation is matched to the dense discretization while preserving the separable TPQ assembly.

The dense MLP baselines use Fourier features followed by Softplus MLPs by following (Ryu et al., 2024). A Fourier map with a fixed matrix $\boldsymbol{B}_F$ returns $[\sin(\boldsymbol{x}\boldsymbol{B}_F), \cos(\boldsymbol{x}\boldsymbol{B}_F)]$, and this Fourier representation is used consistently across the reported runs. The TPQ networks use the CP parameterization in equation 4 with the same Fourier encoding. The main parameter configurations, including the dimension N, number of target eigenvalues L, rank R, batch size, ϵ, s, ℓ, network architecture, and Fourier encoding size (i.e., the number of columns of $\boldsymbol{B}_F$), are summarized in Tables 4-6 for reproducibility. We note that TPQ is not highly sensitive to these choices, and moderate perturbations around the reported settings generally yield comparable performance.

Table 4: NeuralSVD Schrödinger experimental settings.

Problem/architecture	N	L	R	Batch	ϵ	Main settings
Harmonic MLP	2	55	–	512	0.05	$s = 4$, $\ell = 5$, hidden 128,128, Fourier 256
Harmonic TPQ	2	55	10	250^2	0.05	$s = 4$, $\ell = 5$, hidden 128, Fourier 256
Infinite well MLP	2	15	–	512	0.05	$s = \pi$, hidden 128,128, Fourier 256
Infinite well TPQ	2	15	50	300^2	0.05	$s = \pi$, hidden 128, Fourier 256
Hydrogen 2D MLP	2	16	–	4096	0.02	$s = 16$, $\ell = 50$, hidden 128,128, Fourier 1024
Hydrogen 2D TPQ	2	16	10	300^2	0.02	$s = 16$, $\ell = 50$, hidden 128, Fourier 1024
Hydrogen 3D MLP	3	14	–	4096	0.05	$s = 16$, $\ell = 50$, hidden 128,128, Fourier 1024
Hydrogen 3D TPQ	3	14	20	300^3	0.05	$s = 16$, $\ell = 50$, hidden 64, Fourier 1024

Table 5: NeuralEF Schrödinger experimental settings.

Problem/architecture	N	L	R	Batch	ϵ	Main settings
Harmonic MLP	2	55	–	512	0.05	$s = 4$, $\ell = 5$, hidden 128,128, Fourier 256
Harmonic TPQ	2	55	15	250^2	0.05	$s = 4$, $\ell = 5$, hidden 128, Fourier 256
Infinite well MLP	2	15	–	512	0.05	$s = \pi$, hidden 128,128, Fourier 256
Infinite well TPQ	2	15	20	300^2	0.05	$s = \pi$, hidden 128, Fourier 256
Hydrogen 2D MLP	2	9	–	4096	0.02	$s = 16$, $\ell = 50$, hidden 128,128, Fourier 1024
Hydrogen 2D TPQ	2	9	15	300^2	0.02	$s = 16$, $\ell = 50$, hidden 128, Fourier 1024
Hydrogen 3D MLP	3	14	–	4096	0.05	$s = 16$, $\ell = 50$, hidden 128,128, Fourier 1024
Hydrogen 3D TPQ	3	14	20	300^3	0.05	$s = 16$, $\ell = 50$, hidden 64, Fourier 1024

Table 6: SpIN Schrödinger experimental settings.

Problem/architecture	N	L	R	Batch	ϵ	Main settings
Harmonic MLP	2	6	–	128	0.05	$s = 4$, $\ell = 5$, hidden 128,128, Fourier 256
Harmonic TPQ	2	6	20	10^2	0.05	$s = 4$, $\ell = 5$, hidden 64, Fourier 256
Infinite well MLP	2	8	–	128	0.02	$s = \pi$, hidden 128,128, Fourier 256
Infinite well TPQ	2	8	10	10^2	0.02	$s = \pi$, hidden 64, Fourier 256
Hydrogen 2D MLP	2	9	–	128	0.02	$s = 16$, $\ell = 50$, hidden 128,128, Fourier 256
Hydrogen 2D TPQ	2	9	10	10^2	0.02	$s = 16$, $\ell = 50$, hidden 64, Fourier 256
Hydrogen 3D MLP	3	5	–	256	0.05	$s = 16$, $\ell = 50$, hidden 128,128, Fourier 256
Hydrogen 3D TPQ	3	5	20	10^3	0.05	$s = 16$, $\ell = 50$, hidden 64, Fourier 256

The Schrödinger experiments are optimized with RMSProp for 20000 iterations using learning rate 10^{-4}, decay 0.995, momentum 0, and an exponential moving average of parameters with decay 0.99 by following (Ryu et al., 2024). For hydrogen, TPQ uses tensor-integral assembly with a separable Gaussian mixture approximation of the Coulomb potential. For the harmonic oscillator and infinite well, the potentials admit direct evaluation on the effective product quadrature induced by the 1D CP factors, as detailed below.

Harmonic oscillator. The N-D harmonic oscillator uses the potential

$$V_{\mathrm{ho}}(\boldsymbol{x}) = \|\boldsymbol{x}\|_2^2 = \sum_{d=1}^{N} x_d^2, \quad E_{\boldsymbol{n}} = 2|\boldsymbol{n}|_1 + N, \quad \boldsymbol{n} \in \mathbb{N}_0^N. \tag{35}$$

Here $\boldsymbol{x} = (x_1, \ldots, x_N)$ is the spatial coordinate, N is the spatial dimension, x_d is the coordinate along dimension d, V_{ho} is the harmonic potential, $E_{\boldsymbol{n}}$ is the energy of the state indexed by $\boldsymbol{n}$, and

$|\boldsymbol{n}|_1 = \sum_{d=1}^{N} n_d$ is the ℓ_1-norm of the excitation vector $\boldsymbol{n}$. The reported harmonic experiment is 2D. Because V_{ho} is additively separable, the operator $\mathcal{T} = \Delta - \sum_d x_d^2$ can be assembled from 1D kinetic and potential terms. The harmonic validation grid is truncated to $[-\ell, \ell]^2$ with $\ell = 5$.

The harmonic potential is handled directly rather than by a surrogate Gaussian mixture. Because V_{ho} is additive, the same contribution can be written in separable form. For the optimized operator $\mathcal{T} = \Delta - V_{\mathrm{ho}}$, the potential part of the stiffness has the form

$$\mathcal{V}_{lm}^{\mathrm{ho}} = \sum_{r=1}^{R}\sum_{s=1}^{R}\sum_{k=1}^{N} P_{rs}^{lm,k} \prod_{d\neq k} M_{rs}^{lm,d},\ P_{rs}^{lm,k} = \mathbb{E}_{x_k}\left[(u_{\theta_{k,l}}(x_k))_r x_k^2 (u_{\theta_{k,m}}(x_k))_s\right]. \tag{36}$$

Here $\mathcal{V}_{lm}^{\mathrm{ho}}$ is the harmonic-potential contribution to the stiffness entry between modes l and m, $P_{rs}^{lm,k}$ is the 1D potential-weighted correlation on coordinate k, and $M_{rs}^{lm,d}$ denotes the corresponding 1D mass correlation of the two rank factors on coordinate d. This is the additive-separable construction in equation 8 with the 1D multiplication operator x_k^2. Hence the harmonic oscillator does not require a Gaussian-mixture surrogate; the final operator combines the finite-difference Laplacian term with the sign of the optimized operator, i.e., $\Pi_{lm} = \Pi_{lm}^{\Delta} - \mathcal{V}_{lm}^{\mathrm{ho}}$ in the separable view.

Infinite well. The 2D infinite square well is posed on $\Omega = [0, l]^2$ with $l = \pi$, zero interior potential, and homogeneous Dirichlet boundary condition. Here Ω is the physical domain, l is the side length, V_{iw} is the interior infinite-well potential, $u|_{\partial\Omega} = 0$ denotes the Dirichlet boundary condition, and E_{n_x,n_y} is the energy of the state with positive integer mode indices n_x, n_y.

$$V_{\mathrm{iw}}(\boldsymbol{x}) = 0,\ u|_{\partial\Omega} = 0, \quad E_{n_x,n_y} = \frac{\pi^2(n_x^2 + n_y^2)}{l^2}, \quad n_x, n_y \in \mathbb{N}_+. \tag{37}$$

The analytical eigenfunctions are $\psi_{n_x,n_y}(x, y) = 2l^{-1}\sin(n_x\pi x/l)\sin(n_y\pi y/l)$, with quantum-number pairs ordered by $n_x^2 + n_y^2$. The optimized operator is $\mathcal{T} = \Delta$ inside the box, followed by the affine shift $c_{\mathrm{op}} = 50$. Dirichlet conditions are imposed by multiplying the neural output by the positive-box square-root mask $m_d(x) = \sqrt{x(l-x)}/(l/2)$, evaluated on $x \in [0, l]$; the general 1D mask family is listed in Sec. F.4. In the implementation, the TPQ training rule for this experiment uses Gaussian grid proposal samples $x_{d,b} \sim \mathcal{N}(0, \pi^2)$ on the real line and forms their Cartesian product; it is therefore an importance sampling rule for the masked extension of the box problem, rather than uniform sampling restricted to $[0, \pi]^2$.

The infinite barrier is represented through the function space rather than by assigning a numerical value to a $+\infty$ potential. Inside the box, $V_{\mathrm{iw}} = 0$, so the stiffness only contains the Laplacian term assembled by TPQ. The wall is imposed through a hard Dirichlet constraint, written as

$$u_l(\boldsymbol{x}) = M(\boldsymbol{x})\tilde{u}_l(\boldsymbol{x}),\ M(\boldsymbol{x}) = \prod_{d=1}^{N} m_d(x_d), \tag{38}$$

where $\tilde{u}_l$ is the unconstrained neural output, $M(\boldsymbol{x})$ is the full separable boundary mask, and m_d is the 1D mask factor for coordinate d. which follows the separable boundary mask in equation 5 and Sec. F.4. Because the mask factors over dimensions, it is absorbed into the 1D TPQ factors before forming the mass matrix and the Laplacian stiffness. The finite-difference derivatives are therefore applied to the masked 1D factors, and no non-separable potential term is introduced for the infinite well. Although the Gaussian proposal has support on $\mathbb{R}^N$, the infinite-well trial functions are extended by zero outside the physical box through the Dirichlet mask. Validation, eigenvalue reporting, and analytical eigenfunction comparison are performed on the physical uniform grid $[0, \pi]^2$.

Equivalently, the positive-box mask is understood as $m_d(x) = \sqrt{\max(0, x(l-x))}/(l/2)$ on the full real line, so that it vanishes outside $[0, l]$.

Hydrogen atom. For both 2D and 3D hydrogen, the Coulomb potential is

$$V_{\mathrm{hy}}(\boldsymbol{x}) = -\frac{Z}{\|\boldsymbol{x}\|_2},\ Z = 1. \tag{39}$$

Here $\boldsymbol{x} \in \mathbb{R}^N$ is the electron coordinate, $N \in \{2, 3\}$ is the spatial dimension, Z is the nuclear charge, and V_{hy} is the attractive Coulomb potential. The quantities E_n^{2D} and E_n^{3D} are the analytical

energy levels in two and three dimensions, respectively, and n is the principal quantum number with the indexing conventions stated below. The analytical energies used in evaluation are

$$E_n^{\mathrm{2D}} = -\frac{Z^2}{4(n+1/2)^2}, \quad n = 0, 1, \ldots, \; E_n^{\mathrm{3D}} = -\frac{Z^2}{4n^2}, \quad n = 1, 2, \ldots. \tag{40}$$

The Coulomb potential is non-separable. TPQ therefore replaces it by a separable Gaussian mixture,

$$V_{\mathrm{hy}}(\boldsymbol{x}) \approx \sum_{k=1}^{K} \omega_k \exp(-\alpha_k \|\boldsymbol{x}\|_2^2) = \sum_{k=1}^{K} \omega_k \prod_{d=1}^{N} \exp(-\alpha_k x_d^2). \tag{41}$$

In this surrogate, K is the number of Gaussian terms, ω_k is the fitted mixture weight, and $\alpha_k > 0$ is the fitted radial decay rate of the k-th term. The mixture coefficients are fitted before the main eigenfunction training by minimizing a relative squared error over logarithmically spaced radii $r \in [10^{-2}, 60]$. We initialize $\log \alpha_k$ uniformly from $-2 \log 10$ to $4 \log 10$, initialize the weights to 0.1, optimize with Adam at learning rate 0.05, and use negative mixture weights to match the attractive Coulomb sign. The 2D TPQ experiments use $K = 15$ Gaussian terms and 2001 radial samples; the 3D NeuralSVD-TPQ experiment uses $K = 15$ and 6001 radial samples. This is the separable potential pre-training procedure in Sec. 2.1.3, and its ablation is reported in Sec. G.

F.2 Integral Kernel Operator SVD

We evaluate NeuralSVD-TPQ on a non-self-adjoint integral operator $\mathcal{K} : L^2(\Omega) \to L^2(\Omega)$,

$$(\mathcal{K}f)(\boldsymbol{x}) = \int_{\Omega} k(\boldsymbol{x}, \boldsymbol{y}) f(\boldsymbol{y}) d\boldsymbol{y}, \; \Omega = [0, \pi]^N. \tag{42}$$

Here Ω is the integration domain, $k(\boldsymbol{x}, \boldsymbol{y})$ is the kernel, f is the input function, and $\mathcal{K}f$ is evaluated at the output coordinate $\boldsymbol{x}$. In this non-self-adjoint setting, $\boldsymbol{x}$ and $\boldsymbol{y}$ play the roles of left- and right-side coordinates, respectively. The kernel is chosen to have a known low-rank SVD:

$$k(\boldsymbol{x}, \boldsymbol{y}) = \sum_{m=1}^{L} \lambda_m \prod_{d=1}^{N} \sin(m x_d) \prod_{d=1}^{N} \cos(m y_d), \; \lambda = (1, 1/2, 1/3). \tag{43}$$

The coefficients λ_m in the kernel definition are unnormalized amplitudes; after integrating the squared sine and cosine factors over Ω, they induce the theoretical singular values σ_m below. The left and right singular functions are products of 1D sine and cosine factors, respectively: $g_m(\boldsymbol{x}) = \prod_{d=1}^{N} \sin(m x_d)$ and $f_m(\boldsymbol{y}) = \prod_{d=1}^{N} \cos(m y_d)$. Since $\int_0^{\pi} \sin^2(mx) dx = \int_0^{\pi} \cos^2(mx) dx = \pi/2$, the theoretical singular values are

$$\sigma_m = \lambda_m (\pi/2)^N, \; m = 1, \ldots, L. \tag{44}$$

In the reported experiments, $L = 3$, $N \in \{2, 3, 4, 5\}$, $\lambda = (1, 1/2, 1/3)$. Both the original NeuralSVD and NeuralSVD-TPQ optimize the same nested low-rank objective for 1000 epochs. All kernel-operator samples are drawn uniformly from $[0, \pi]^N$; the original NeuralSVD estimates both operator and metric terms by Monte Carlo quadrature, whereas TPQ evaluates the 1D factors on independently sampled coordinates and assembles the corresponding operator and metric terms, Π and Σ, through tensor contractions. NeuralSVD uses a sine-activated architecture with input dimension N, two hidden layers of width 128, and output dimension L, with sine frequency $w_0 = 2$, Adam learning rate 10^{-3}, and batch size 100000. TPQ uses 1D networks with input dimension 1, two hidden layers of width 64, and output dimension LR, again with sine frequency $w_0 = 2$, CP rank $R = 15$, Adam learning rate 10^{-3}, and $B_{\mathrm{1D}} = 2000$ 1D samples. Only the output-side (left) singular function is multiplied by the separable boundary factor $b(\boldsymbol{x}) = \prod_d x_d(\pi - x_d)$ before forming the low-rank kernel approximation. Test errors are measured on 5000 random points by comparing the learned singular values with σ_m, as shown in Fig. 19.

F.3 NTK Decomposition

We also test NeuralSVD-TPQ by decomposing the neural tangent kernel induced by a pretrained target network on $\Omega = [0, 6]^2$. The target network is

$$f_\theta(\boldsymbol{x}) = \boldsymbol{W}_2 \sin(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1), \tag{45}$$

where f_θ is a scalar target network with parameters $\theta = \{\boldsymbol{W}_1, \boldsymbol{W}_2, \boldsymbol{b}_1\}$ and hidden width 32. It is first pretrained to fit $u(x_1, x_2) = \sin(x_1) + \cos(x_2)$, and then frozen. The NTK is

$$\Theta(\boldsymbol{x}, \boldsymbol{y}) = \langle \nabla_\theta f_\theta(\boldsymbol{x}), \nabla_\theta f_\theta(\boldsymbol{y}) \rangle . \tag{46}$$

Thus Θ is the kernel obtained by the inner product of parameter gradients at two input locations, which defines an integral operator on functions over Ω. The target network is pretrained for 1500 epochs with Adam learning rate 10^{-2} and batch size 2048. After pretraining, θ is frozen, and NTK values are obtained from parameter-gradient inner products of f_θ.

Since $\Theta(\boldsymbol{x}, \boldsymbol{y}) = \Theta(\boldsymbol{y}, \boldsymbol{x})$, the associated NTK integral operator is self-adjoint and positive semidefinite. Following the NeuralSVD notation, the two branches v and u correspond to the left and right singular functions; in this self-adjoint formulation, they approximate the same eigenspaces, yielding

$$\Theta(\boldsymbol{x}, \boldsymbol{y}) \approx \sum_{\ell=1}^{L} v_\ell(\boldsymbol{x}) u_\ell(\boldsymbol{y}). \tag{47}$$

For the two-dimensional NTK decomposition, we use the separable mask $b(\boldsymbol{x}) = \prod_{d=1}^{2} x_d(6 - x_d)$ on $\Omega = [0, 6]^2$. The eigenvalues are implicitly represented by the scales of the two NeuralSVD branches u_ℓ and v_ℓ. The original NeuralSVD baseline represents u and v by MLPs on Ω. NeuralSVD-TPQ uses the variable-separable CP parameterization: for each branch,

$$u_\ell(\boldsymbol{x}) = \sum_{r=1}^{R} \prod_{d=1}^{2} \phi_{\ell r}^{u,d}(x_d), \; v_\ell(\boldsymbol{x}) = \sum_{r=1}^{R} \prod_{d=1}^{2} \phi_{\ell r}^{v,d}(x_d), \tag{48}$$

where R is the tensor-product rank.

Both variants optimize the same self-adjoint NeuralSVD objective with an operator-alignment term and a metric regularizer:

$$\begin{aligned} \mathcal{L}_{\mathrm{NTK}} = & -2 \sum_{\ell=1}^{L} w_\ell \int_\Omega \int_\Omega v_\ell(\boldsymbol{x}) \Theta(\boldsymbol{x}, \boldsymbol{y}) u_\ell(\boldsymbol{y}) d\boldsymbol{x} d\boldsymbol{y} \\ & + \sum_{i,j=1}^{L} M_{ij} \left(\int_\Omega u_i(\boldsymbol{x}) u_j(\boldsymbol{x}) d\boldsymbol{x} \right) \left(\int_\Omega v_i(\boldsymbol{x}) v_j(\boldsymbol{x}) d\boldsymbol{x} \right). \end{aligned} \tag{49}$$

Here w_ℓ and M_{ij} are the vector and matrix nesting weights used to stabilize ordered spectral learning. In TPQ, the metric integrals decompose into products of 1D factor integrals, while the NTK operator term is estimated on the induced tensor-product quadrature with full grid sampling, since the NTK is not guaranteed to be separable.

The NTK EVD uses $L = 5$, $R = 30$, and 5000 training epochs. The original NeuralSVD uses two sine-activated MLP branches with 128 hidden dimension, sine frequency $w_0 = 4$, batch size 80000, and Adam learning rate 10^{-3}. NeuralSVD-TPQ uses two variable-separable CP branches, each consisting of two 1D factor networks per branch with the same architecture, sine frequency $w_0 = 4$, rank $R = 30$, and Adam learning rate 10^{-3}. Since $N = 2$, we use $B_{\mathrm{1D}} = 300$, corresponding to an effective product quadrature of about 90000 points. The NTK operator term is evaluated on this tensor-product quadrature, while the metric terms are assembled from 1D integrals using the TPQ.

For the downstream proxy KRR evaluation, the low-rank NTK approximation is

$$K_{\mathrm{approx}}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{\ell=1}^{L} v_\ell(\boldsymbol{x}) u_\ell(\boldsymbol{y}). \tag{50}$$

Given training labels y_{tr}, KRR solves

$$\alpha = (K_{\mathrm{tr,tr}} + \lambda I)^{-1} y_{\mathrm{tr}}, \; \hat{y}_{\mathrm{te}} = K_{\mathrm{te,tr}} \alpha. \tag{51}$$

In the KRR formulation above, $K_{\mathrm{tr,tr}}$ and $K_{\mathrm{te,tr}}$ are kernel matrices on training-training and test-training pairs, I is the identity matrix, α is the fitted dual coefficient vector, and $\hat{y}_{\mathrm{te}}$ denotes test predictions. The λ in the KRR formulation is the ridge parameter. We use ridge parameter $\lambda = 10^{-4}$ and one shared dataset of 1000 points on $[0, 6]^2$ for all methods. The proxy labels are generated from the same $\sin x_1 + \cos x_2$ target family used in pretraining. The train/test splits are 20:80, 40:60, 60:40, and 80:20, and the reported metric is test MSE. The approximated NeuralSVD KRR and the approximated NeuralSVD-TPQ KRR are evaluated on the same split to test the effect of the kernel approximation, as shown in Table 2.

F.4 Boundary Mask Choice

The specific form of the 1D boundary mask $m_d(x_d)$ in equation 5 falls into two categories. For *unbounded domains* where the solution is expected to decay at infinity, we could impose an ℓ_1-style envelope $m_d(x_d) = \exp(-|x_d|/s_l)$ with a learnable scale s_l. This choice decomposes the decay into independent 1D components and thus preserves dimensional decoupling. For *finite box domains* with Dirichlet conditions that require the solution to vanish on $\partial\Omega$, we impose 1D envelopes. On a positive domain $\Omega = [0, l]^N$, two natural choices are the square-root envelope $m_d(x_d) = \frac{2}{l}\sqrt{\max(0,\, x_d(l - x_d))}$ and the exponential envelope $m_d(x_d) = (1 - e^{-x_d})(1 - e^{-(l-x_d)})$. On a symmetric domain $\Omega = [-l, l]^N$, these are adapted to $m_d(x_d) = \max\big(0,\ (\sqrt{2l^2 - x_d^2} - l)/l\big)$ and $m_d(x_d) = (1-e^{-(l-x_d)})(1-e^{-(x_d+l)})$, respectively. All envelope functions vanish on the boundary and can be evaluated independently on 1D grids. Since $M(\boldsymbol{x}) = \prod_{d=1}^{N} m_d(x_d)$ remains separable, the mask is absorbed into the 1D TPQ factors before forming mass and stiffness matrices. Once the mask has been absorbed, all subsequent derivations in Sec. 2.1.2 apply without modification.

For the harmonic oscillator, we adopt the unbounded-domain exponential decay mask $m_d(x_d) = \exp(-|x_d|/s_l)$. For the reported infinite-well experiment, the computational box is $[0, l]^N$ with $l = \pi$, and the hard Dirichlet mask is the positive-domain square-root envelope $m_d(x_d) = \sqrt{\max(0, x_d(l - x_d))}/(l/2)$. For hydrogen, no explicit boundary mask is applied: the eigenfunctions are represented on $\mathbb{R}^N$, with Gaussian proposal sampling during training and validation on the truncated domain $[-\ell, \ell]^N$ with $\ell = 50$. For the NTK decomposition and the non-self-adjoint integral-kernel SVD, the boundary factor is applied. Specifically, on $[0, a]^N$ we use $b(\boldsymbol{x}) = \prod_{d=1}^{N} x_d(a - x_d)$, with $a = 6$ for NTK and $a = \pi$ for the integral kernel experiment. This factor is separable and is applied independently to the factors in TPQ; for the integral-kernel SVD, it is applied only to the left singular-function factors.

G Ablation Study of Pre-trained Separable Potential Function

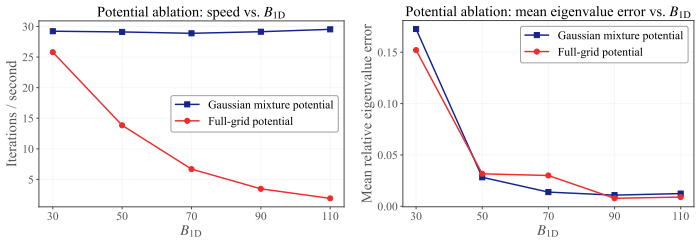


Figure 9: Ablation study of the Gaussian mixture pre-training (vs. full-grid original potential calculation) for the 3D hydrogen atom experiment.

As shown in Fig. 9, the pre-trained Gaussian mixture potential preserves the efficiency advantage of TPQ in the 3D hydrogen atom experiment. When B_{1D} increases, the Gaussian mixture implementation keeps the training speed nearly constant, because the potential term is evaluated through separable 1D contractions. In contrast, the full-grid potential evaluation becomes increasingly expensive as the grid batch size B_{1D} grows. The accuracy trend confirms that this speedup does not come at the cost of degraded spectral estimation, or even brings accuracy improvements when $B_{1D} = 70$ in this particular case. This supports the use of the pre-trained separable potential as a computationally efficient surrogate for the non-separable Coulomb potential, while also benefiting from the smoothing of the singularity near the origin.

H Future Directions

Extending TPQ to realistic many-body quantum systems remains non-trivial. Bosonic wavefunctions must be invariant under particle exchange, whereas fermionic wavefunctions must change sign when two identical particles are exchanged. The latter requires anti-symmetric inductive biases, such as neural Slater determinants (Pfau et al., 2020; Hermann et al., 2020) or continuous fermionic exchange operators (Pang et al., 2022). Incorporating these exchange symmetries into the variable-

separable TPQ architecture, while retaining efficient 1D quadrature, is an important direction for future research.

As a preliminary step, TPQ can first be evaluated on a controlled many-body Schrödinger benchmark with a known exact solution. Let P denote the number of particles and D the spatial dimension of each particle. The many-body wavefunction is then defined over a configuration space of total dimension $N = PD$. Here, we consider the simple setting of P distinguishable particles moving in one spatial dimension, i.e., $D = 1$ and hence $N = P$. The position of particle i is denoted by x_i, and the complete many-body configuration is $\boldsymbol{x} = (x_1, \ldots, x_P) \in \mathbb{R}^P$. The particles are confined by individual harmonic potentials and interact through nearest-neighbor harmonic couplings (i.e., the coupled harmonic oscillator system). The corresponding Schrödinger equation is

$$\mathcal{H}_{\mathrm{mb}} = -\sum_{i=1}^{P} \frac{\partial^2}{\partial x_i^2} + \sum_{i=1}^{P} \omega_i^2 x_i^2 + \kappa \sum_{i=1}^{P-1} (x_i - x_{i+1})^2, \ \mathcal{H}_{\mathrm{mb}} \psi_l(\boldsymbol{x}) = E_l \psi_l(\boldsymbol{x}). \tag{52}$$

Here, ω_i is the trapping frequency of particle i, $\kappa \geq 0$ controls the interaction strength, ψ_l is the l-th many-body eigenfunction, and E_l is its energy. When $\kappa = 0$, the particles are independent and the wavefunction can be separated into single-particle functions (the harmonic oscillator considered in Fig. 2). When $\kappa > 0$, the term $(x_i - x_{i+1})^2$ couples neighboring particles, so the eigenfunctions are generally non-separable in the original particle coordinates.

The potential can be written compactly as $\boldsymbol{x}^\top \boldsymbol{A} \boldsymbol{x}$, where $\boldsymbol{A} \in \mathbb{R}^{P \times P}$ is a symmetric tridiagonal matrix. Its diagonal entries are

$$A_{ii} = \begin{cases} \omega_i^2 + \kappa, & i = 1 \text{ or } i = P, \\ \omega_i^2 + 2\kappa, & 1 < i < P, \end{cases} \tag{53}$$

and its neighboring off-diagonal entries are $A_{i,i+1} = A_{i+1,i} = -\kappa$. Let $\lambda_\alpha(\boldsymbol{A})$ denote the α-th eigenvalue of $\boldsymbol{A}$. The corresponding normal-mode frequency is $\Omega_\alpha = \sqrt{\lambda_\alpha(\boldsymbol{A})}$, and the exact many-body energy levels are

$$E_{\boldsymbol{n}} = \sum_{\alpha=1}^{P} (2n_\alpha + 1)\Omega_\alpha, \ n_\alpha \in \mathbb{N}_0. \tag{54}$$

Here, $\boldsymbol{n} = (n_1, \ldots, n_P)$ records the excitation number of each normal mode. To apply TPQ, we represent the l-th many-body eigenfunction using the same rank-R CP neural ansatz as in Eq. equation 4. Instead of evaluating one network on a full P-dimensional grid, TPQ evaluates P one-dimensional subnetworks, one for each particle coordinate. For every coordinate x_i, we draw B_{1D} independent 1D samples. Their tensor product corresponds to an effective quadrature containing B_{1D}^P points, while only PB_{1D} one-dimensional network inputs need to be evaluated. The mass matrix, kinetic energy, and the terms involving x_i^2 can be assembled directly using the 1D TPQ quantities introduced in equation 6 and equation 8. The interaction between particles i and $i+1$ can be expanded as $\kappa(x_i - x_{i+1})^2 = \kappa x_i^2 + \kappa x_{i+1}^2 - 2\kappa x_i x_{i+1}$. The first two terms are ordinary 1D quadratic-potential terms. To evaluate the cross term $x_i x_{i+1}$, we additionally compute the 1D first moment

$$X_{rs}^{lm,i} = \mathbb{E}_{x_i}\left[\left(\psi_{\theta_{i,l}}(x_i)\right)_r \left(\psi_{\theta_{i,m}}(x_i)\right)_s x_i \right]. \tag{55}$$

The cross-coordinate contribution is assembled as

$$\mathcal{V}_{lm}^{(i,i+1)} = -2\kappa \sum_{r=1}^{R} \sum_{s=1}^{R} X_{rs}^{lm,i} X_{rs}^{lm,i+1} \prod_{d \notin \{i,i+1\}} M_{rs}^{lm,d}, \tag{56}$$

where $M_{rs}^{lm,d}$ is the 1D overlap integral defined in equation 6. Therefore, the coupled potential can be assembled from 1D integrals, without using the surrogate Gaussian potential or evaluating the full P-dimensional grid. A concrete experiment is set $\omega_i = 1$ for all particles and consider $P = 3, \kappa = 0.2$ with $L = 20$ target eigenstates. We choose the batch sizes such that NeuralEF and NeuralEF-TPQ have comparable iterations per second. Fig. 10 presents the preliminary results on this coupled system. TPQ improves the convergence efficiency of NeuralEF.

Further improving the convergence of eigenfunction learning by preconditioning the operator (such as NTK-based preconditioning (Luo & Meng, 2026) and parameter-space curvature-based preconditioning (Martens & Grosse, 2015)), or using spectral transformations (Wang et al., 2025) is another promising direction.

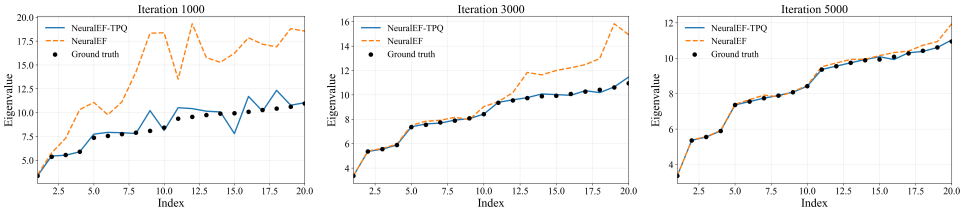


Figure 10: Results on a many-body Schrödinger equation (coupled harmonic oscillator) using NeuralEF and its TPQ version. TPQ improves the convergence efficiency.

I More Numerical Results

More eigenfunction output results (with and without ground-truth alignment) for the 2D & 3D Schrödinger equations are shown in Figs. 11-18. TPQ consistently improves the convergence efficiency and stability of current neural operator decomposition methods SpIN, NeuralEF, and NeuralSVD with or without ground-truth alignment. Here, the ground-truth alignment was conducted by following (Ryu et al., 2024). Since eigenfunctions inside a degenerate eigenspace are only identifiable up to an orthogonal change of basis, the learned eigenfunctions need not coincide pointwise with a particular analytical basis even when they span the correct subspace. Therefore, for each degenerate block of multiplicity k, let $\boldsymbol{A} \in \mathbb{R}^{n\times k}$ and $\boldsymbol{B} \in \mathbb{R}^{n\times k}$ denote the predicted and analytical eigenfunction values evaluated on the same n validation points, respectively. We align $\boldsymbol{A}$ to $\boldsymbol{B}$ by solving the orthogonal Procrustes problem

$$\Omega^\star = \underset{\Omega^\top\Omega=\boldsymbol{I}_k}{\arg\min} \|\boldsymbol{A}\Omega - \boldsymbol{B}\|_F^2 = \underset{\Omega^\top\Omega=\boldsymbol{I}_k}{\arg\max} \operatorname{Tr}(\Omega^\top \boldsymbol{A}^\top \boldsymbol{B}). \tag{57}$$

Let $\boldsymbol{A}^\top\boldsymbol{B} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^\top$ be the singular value decomposition. The optimal rotation is then given by $\Omega^\star = \boldsymbol{U}\boldsymbol{V}^\top$, and the aligned eigenfunctions are $\boldsymbol{A}_{\text{aligned}} = \boldsymbol{A}\Omega^\star$. We apply this procedure independently to each degenerate eigenspace and concatenate the aligned blocks for visualization (Figs. 12, 14, 16, 18). For non-degenerate states ($k = 1$), this reduces to the usual sign correction.

More results for kernel integral operator SVD and NTK EVD using NeuralSVD-TPQ are shown in Figs. 19-20, which further validate the improved convergence and numerical stability brought by the TPQ framework.

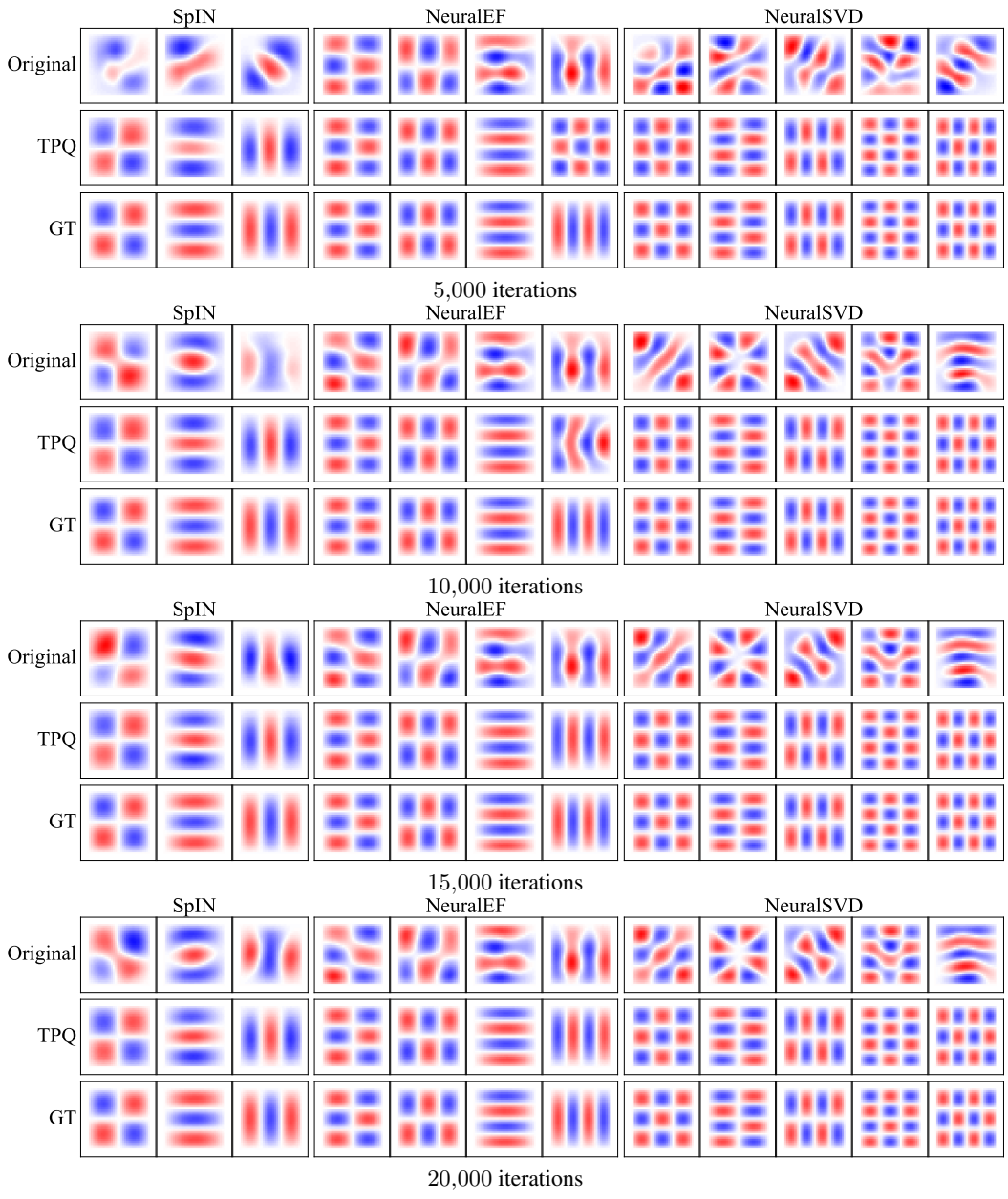


Figure 11: Native eigenfunction outputs (*without* alignment) for the 2D infinite potential well across training iterations.

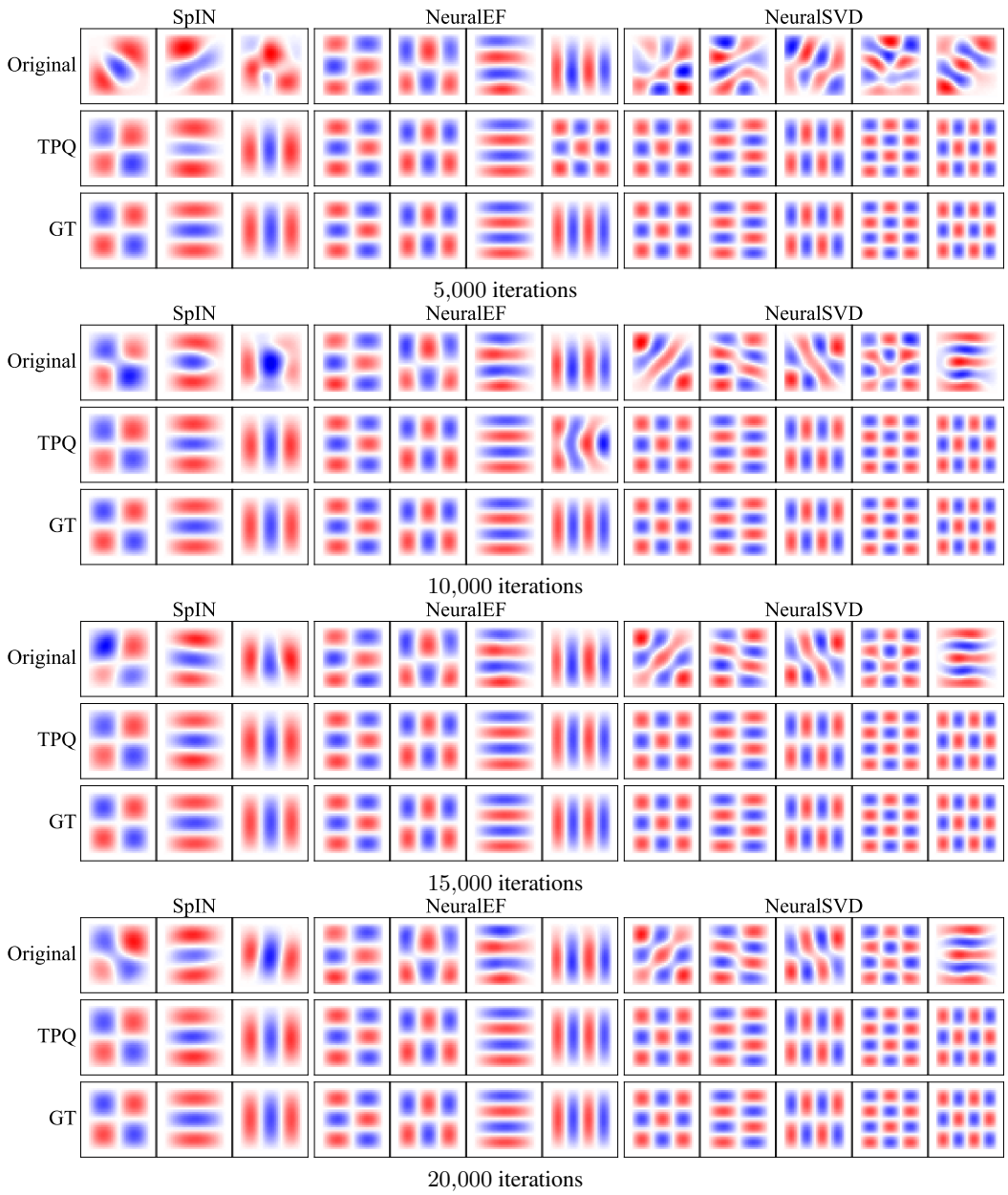


Figure 12: Ground-truth-aligned eigenfunction outputs for the 2D infinite potential well across training iterations.

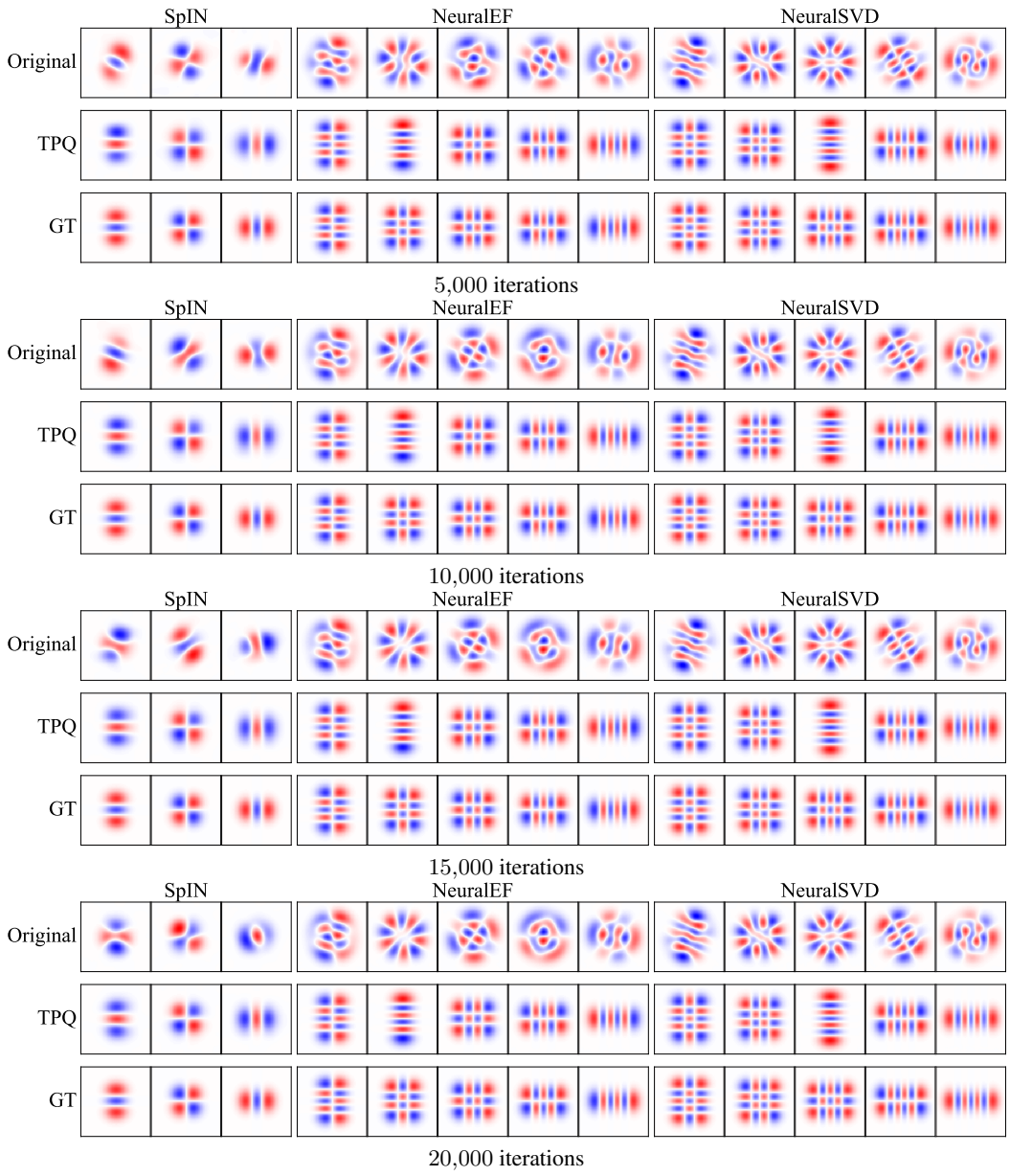


Figure 13: Native eigenfunction outputs (*without* alignment) for the 2D harmonic oscillator across training iterations.

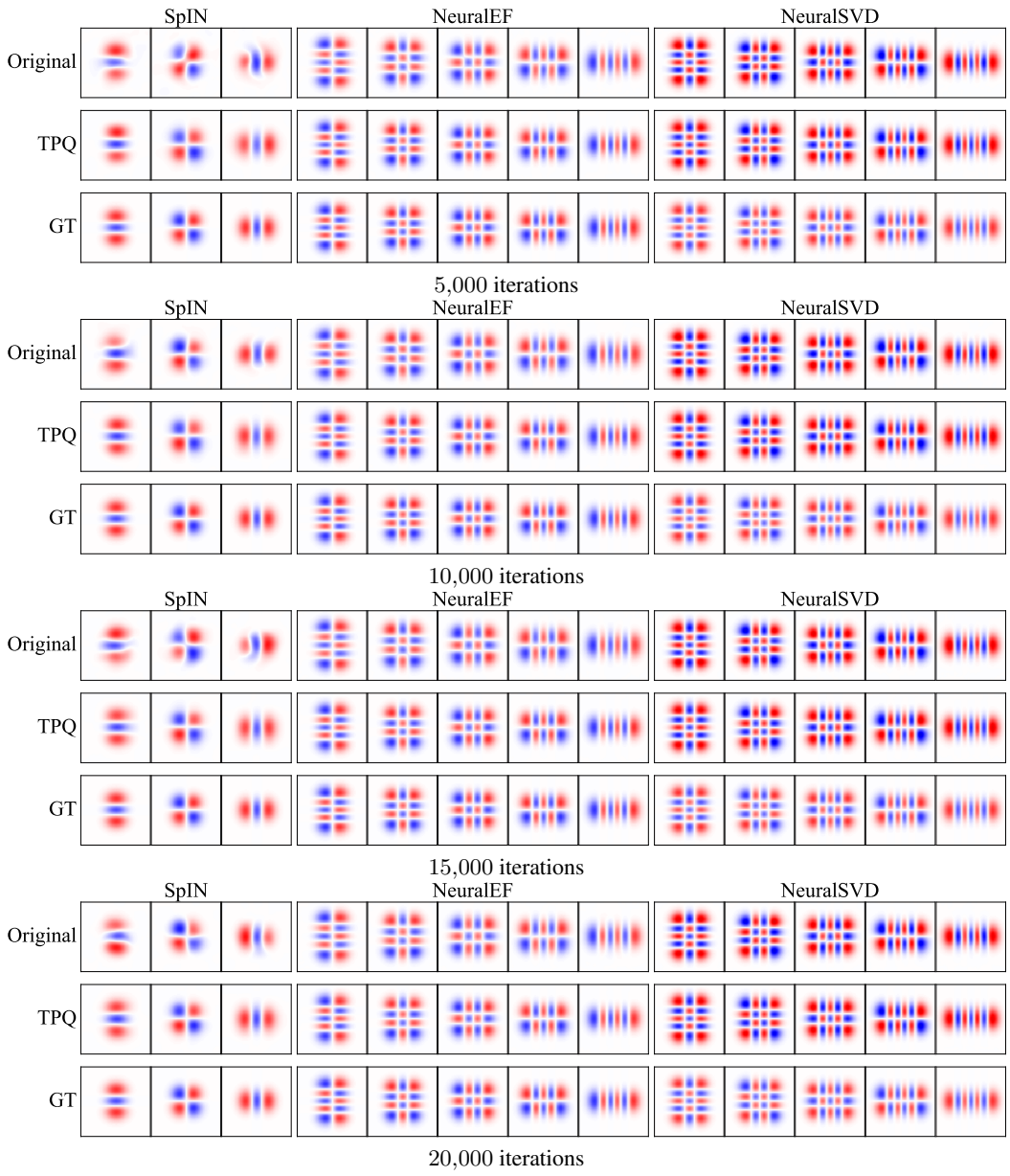


Figure 14: Ground-truth-aligned eigenfunction outputs for the 2D harmonic oscillator across training iterations.

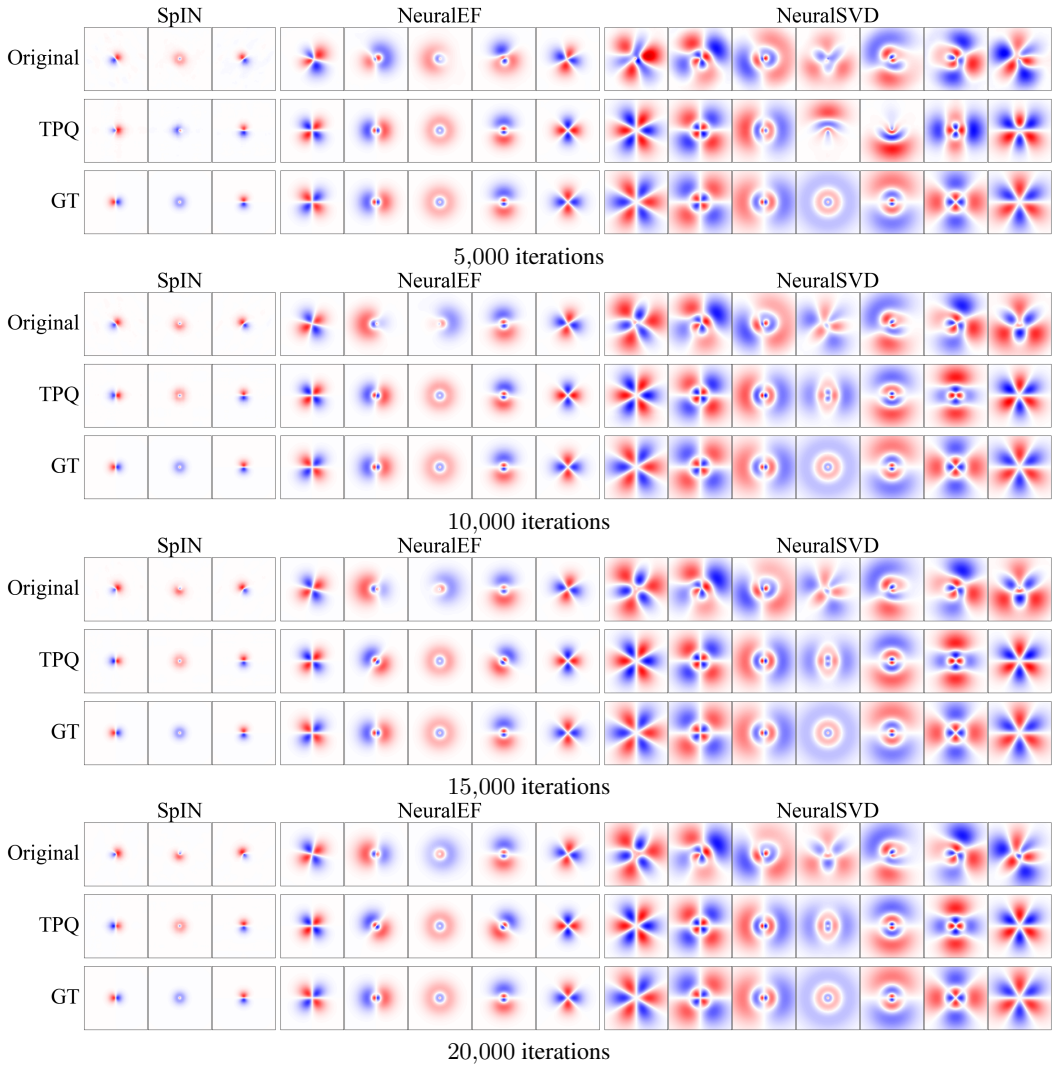


Figure 15: Native eigenfunction outputs (*without* alignment) for the 2D hydrogen atom across training iterations.

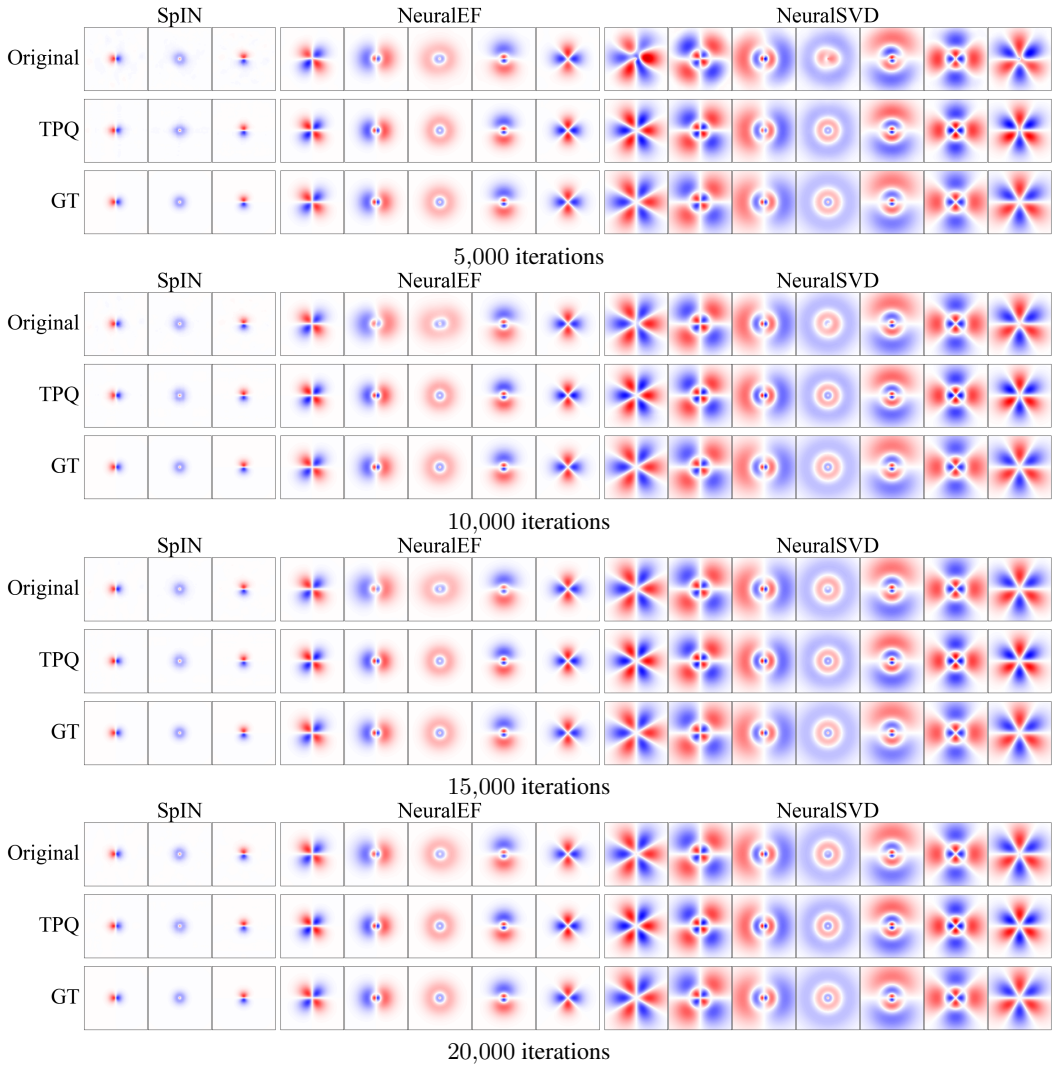


Figure 16: Ground-truth-aligned eigenfunction outputs for the 2D hydrogen atom across training iterations.

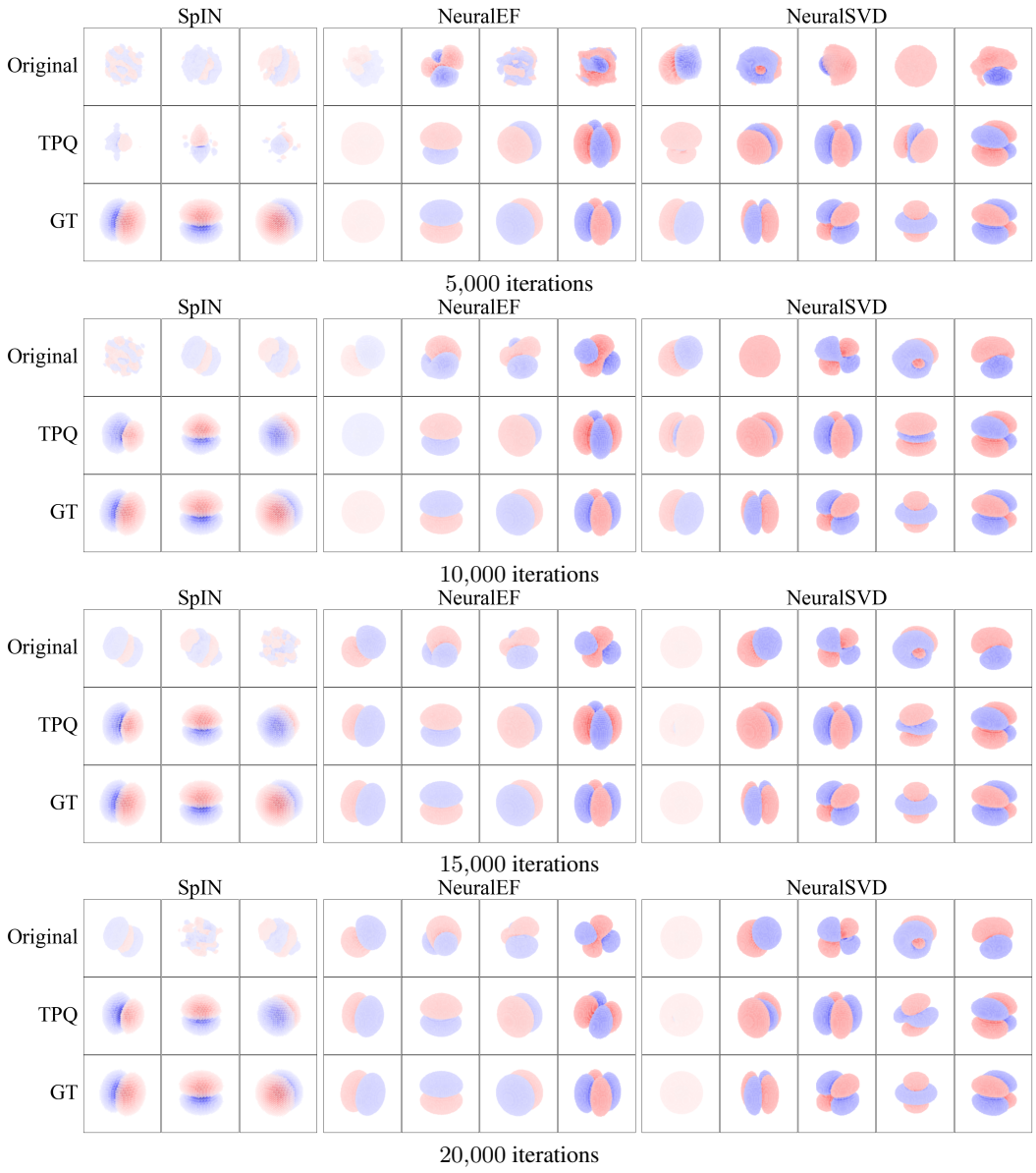


Figure 17: Native eigenfunction outputs (*without* alignment) for the 3D hydrogen atom across training iterations.

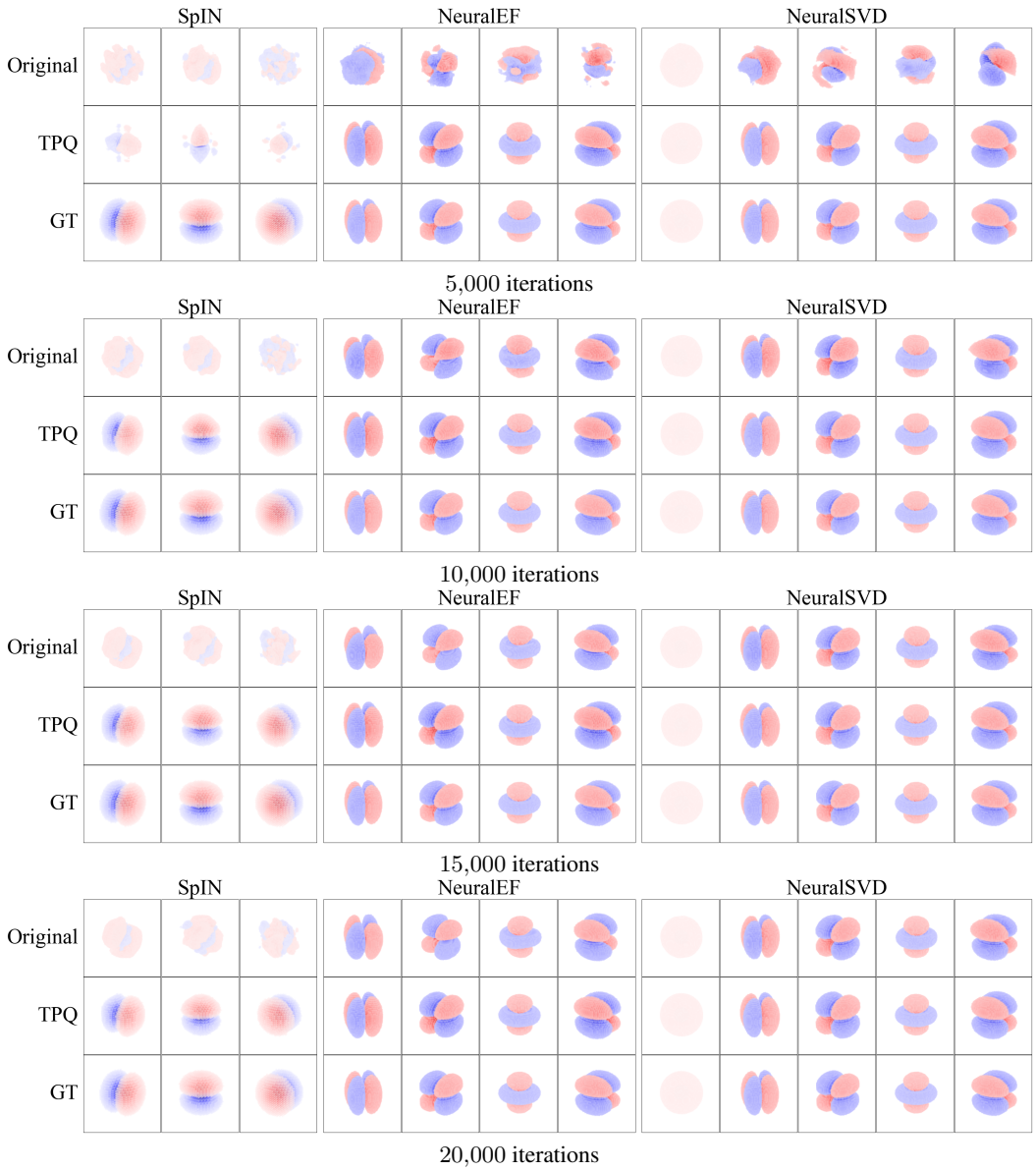


Figure 18: Ground-truth-aligned eigenfunction outputs for the 3D hydrogen atom across training iterations.

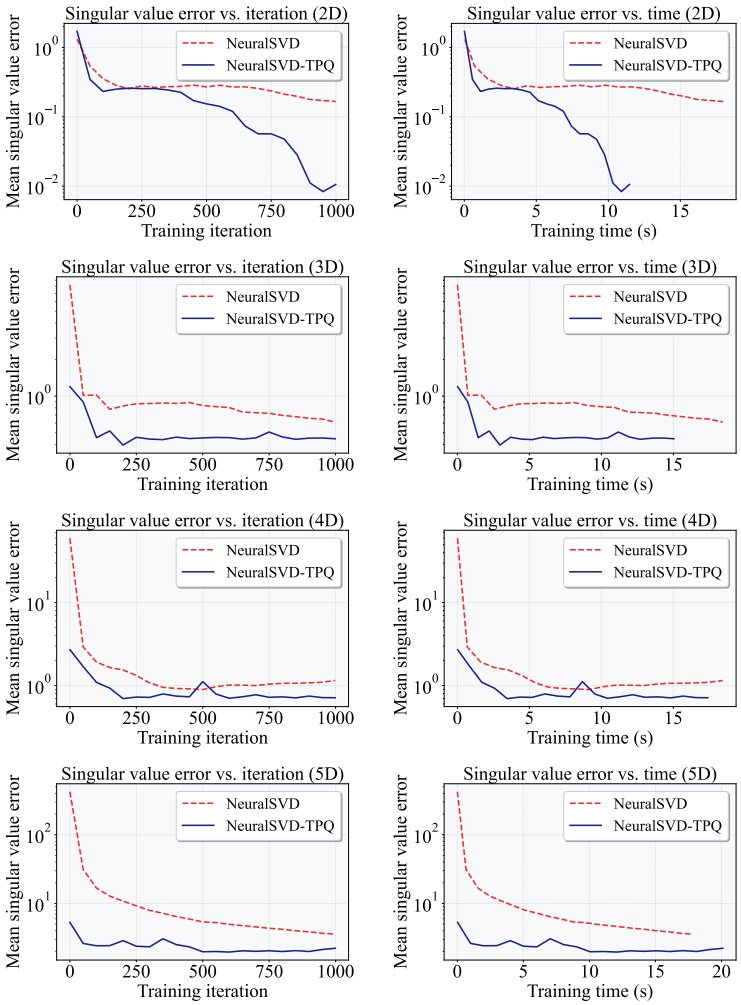


Figure 19: Results for kernel integral operator SVD (singular value error vs. iteration & time) using NeuralSVD-TPQ across various dimensions.

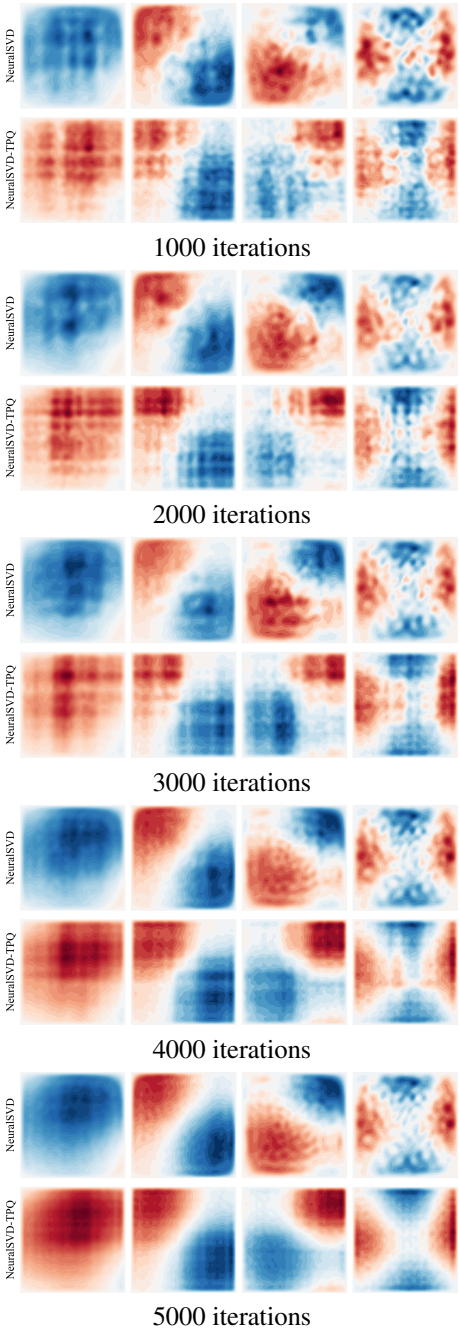


Figure 20: Decomposed NTK eigenfunctions by NeuralSVD and TPQ across iterations. TPQ serves as a low-rank structural regularization during training.

J PyTorch Code Examples

J.1 PyTorch Code of the Separable Jacobian Gradient

Listing 1 gives the essential PyTorch implementation of the separable Jacobian gradient used in the SpIN-TPQ 3D hydrogen experiment. The stiffness matrix `ham` (i.e., Π) is differentiated by ordinary PyTorch autograd through the TPQ operator assembly, while the mass-matrix Jacobian $\partial\Sigma/\partial\theta$ is assembled explicitly from 1D CP factors and 1D parameter Jacobians, which corresponds to the practical one-sided approximation in equation 14.

Listing 1: Separable Jacobian gradient used by SpIN-TPQ.

```python
def spin_step(sigma, pi):
    # Cholesky-normalize Sigma and build Lambda=L^{-1} Pi L^{-T}.
    chol = torch.linalg.cholesky(
        sigma + 1e-3 * torch.eye(sigma.shape[0], device=sigma.device,
                                 dtype=sigma.dtype)
    )
    chol_inv = torch.linalg.inv(chol)
    lam = chol_inv @ pi @ chol_inv.T
    return chol, chol_inv, lam, torch.diagonal(lam)


class SpINFunction(torch.autograd.Function):
    # Custom autograd wrapper for the SpIN Rayleigh trace.
    # The forward pass returns the objective and the two multiplier masks.
    # During backward, PyTorch receives masked matrix gradients dL/dSigma
    # and dL/dPi instead of the dense, unconstrained trace gradients.
    @staticmethod
    @torch.cuda.amp.custom_fwd
    def forward(ctx, sigma, pi):
        chol, chol_inv, lam, eigvals = spin_step(sigma, pi)
        diag = torch.diag(torch.diagonal(chol_inv))
        gsigma = chol_inv.T @ torch.triu(lam @ diag)
        gpi = -chol_inv.T @ diag
        ctx.save_for_backward(gsigma, gpi)
        return torch.trace(lam), eigvals, chol, gsigma, gpi

    @staticmethod
    @torch.cuda.amp.custom_bwd
    def backward(ctx, *grad_outputs):
        return ctx.saved_tensors


def cp_1d_components(model, dim_samples, use_vmap=True):
    # Evaluate frozen 1D CP factors and compute their
    # per-parameter Jacobians.  Since every parameter belongs to exactly one
    # 1D sub-network, the expensive N-D grid Jacobian is never constructed.
    params, jac = dict(model.named_parameters()), {}
    with torch.no_grad():
        phi_1d = model(None, dim_samples)  # Shape: (L, R, D, B_1D).

    for d, net in enumerate(model.net_list):
        prefix = f"net_list.{d}."
        local = {k.split(prefix)[1]: v for k, v in params.items()
                 if prefix in k}
        local_to_global = {k.split(prefix)[1]: k for k in params
                           if prefix in k}
        if not local:
            continue

        def call(p, inp):
            # Stateless call lets jacrev differentiate only this 1D network.
            vmapped = inp.dim() == 1
            out = functional_call(
                net, p, (inp.unsqueeze(0) if vmapped else inp,)
            )
            return out.squeeze(0) if vmapped else out

        inp = model._ensure_2d(dim_samples[d])
        if use_vmap:
            jacs = vmap(jacrev(call, argnums=0),
                        in_dims=(None, 0))(local, inp)
        else:
            jacs = jacrev(call, argnums=0)(local, inp)

        for name, value in jacs.items():
            jac[local_to_global[name]] = (d, value)
```

```
    return phi_1d, jac


class SpIN(nn.Module):
    def compute_loss_operator(self, operator, x, dim_samples, importance,
                              NN_type="CP_network", use_tensor_integral=True,
                              *args, **kwargs):
        # operator(...) assembles Sigma and Pi by TPQ from 1D quadratures.
        # Pi keeps its computational graph, so its masked gradient flows
        # through normal autograd after loss.backward().
        _, phi, mass, ham = operator(
            self.model, x, dim_samples, importance, use_tensor_integral
        )

        # Build the ingredients for the explicit separable Sigma Jacobian.
        phi_1d, jac_1d = cp_1d_components(
            self.model, dim_samples, self.use_vmap
        )
        self.sigma_avg.data = self.moving_average(
            self.sigma_avg.data, mass
        )
        loss, eigvals, self.chol.data, gsigma, _ = SpINFunction.apply(
            self.sigma_avg.data, ham
        )

        L, R, D, B = phi_1d.shape

        # overlaps[k] stores all 1D mass factors
        # M^{lm,k}_{rs}=sum_b phi_{k,l,r}(x_b) phi_{k,m,s}(x_b).
        overlaps = [
            (phi_1d[:, :, k].reshape(L * R, B)
             @ phi_1d[:, :, k].reshape(L * R, B).T)
            .reshape(L, R, L, R)
            for k in range(D)
        ]

        # contexts[d] is the product over all dimensions except d, multiplied
        # by the remaining 1D factor from dimension d.  It implements the
        # tensor-product coefficient in the one-sided Jacobian formula.
        contexts = {}
        for d in range(D):
            ctx = torch.ones(L, R, L, R, device=phi_1d.device)
            for k in range(D):
                if k != d:
                    ctx = ctx * overlaps[k]
            contexts[d] = torch.einsum(
                "lrms,msb->lrmb", ctx, phi_1d[:, :, d]
            ).permute(0, 2, 1, 3).reshape(L, L, R * B)

        for name, p in self.model.named_parameters():
            if p.requires_grad:
                d, j = jac_1d[name]
                p_shape = j.shape[3:]
                j_perm = j.permute(
                    1, 2, 0, *range(3, j.ndim)
                ).reshape(L, R * B, math_prod(p_shape) if p_shape else 1)

                # Separable approximation of dSigma/dtheta_{d,l}.
                # The factor 2 is the symmetric one-sided estimator used in
                # the doubled one-sided separable Jacobian estimator;
                # division by B**D averages the
                # implicit tensor-product grid.
                j_new = 2 * torch.bmm(
                    contexts[d], j_perm
                ).transpose(0, 1).reshape(L, L, *p_shape) / B**D

                key = name.replace(".", "_")
                self.j_avg[key].data = self.moving_average(
                    self.j_avg[key].data, j_new
                )

                # Apply the SpIN Sigma mask manually:
                # grad(theta)=sum_{l,m} gsigma_{lm} dSigma_{lm}/dtheta.
                p.grad = torch.einsum(
                    "lm,lm...->...", gsigma, self.j_avg[key].data
                )

        dummy = torch.zeros(1, self.neigs, device=x.device)
        return loss, dict(f=dummy, Tf=dummy, eigvals=eigvals)
```

J.2 PyTorch Code of the TPQ Evaluation

Listing 2 shows the proposed TPQ integral assembly. In `CP_network.forward`, the network is evaluated only on the marginal 1D grids and returns the CP factors in the ansatz equation 4. The class `SeparableMatrixAssembler` then implements the tensor-product quadrature: it builds the 1D mass factors used in equation 6, the finite-difference kinetic factors for the additively separable operator in equation 8, the full mass matrix Σ, and the diagonal kinetic term without using a full B_{1D}^{D} grid. Finally, `Operator` combines these TPQ mass/kinetic terms with the separable Gaussian mixture potential in equation 12, whose tensor-product contraction follows equation 10, to form the diagonal operator matrix Π. The Gaussian mixture coefficients `weights` and `alphas` are pre-trained by `fit_potential` as in Listing 3.

Listing 2: TPQ assembly of mass, kinetic, and potential integrals in NeuralSVD-TPQ.

```python
import opt_einsum as oe


class CP_network(nn.Module):
    # CP tensor-function representation used by TPQ:
    def forward(self, x=None, dim_samples=None):
        if dim_samples is not None:
            # TPQ path: evaluate only D one-dimensional marginal grids.
            # Output shape is (L, R, D, B_1D), avoiding a B_1D^D grid.
            outs = []
            for d, net in enumerate(self.net_list):
                out = net(self._ensure_2d(dim_samples[d]))
                # Move the eigenfunction/rank axes (L,R) to the front.
                outs.append(
                    out.permute(*(list(range(out.dim()))[-2:]
                                  + list(range(out.dim()))[:-2]))
                )
            return torch.stack(outs, dim=2)[:self.neigs]

        # Dense fallback: evaluate explicit points by multiplying all
        # coordinate factors and summing over the CP rank.
        xs = [x[..., d:d + 1] for d in range(self.ndim)]
        y = self.net_list[0](self._ensure_2d(xs[0]))
        for d in range(1, self.ndim):
            y = y * self.net_list[d](self._ensure_2d(xs[d]))
        return y.sum(dim=-1)[..., :self.neigs]


class SeparableMatrixAssembler:
    # TPQ assembler for the diagonal Hamiltonian terms required by the
    # self-adjoint NeuralSVD experiment. It builds 1D mass/stiffness factors
    # without backpropagating through a full B_1D^3 graph.
    def __init__(self, eps):
        self.eps = eps

    def __call__(self, f, xs, dim_samples, importance):
        def eval_g(ds):
            # CP factors phi_{l,r,d,b}, shape (L,R,D,B_1D).
            phi = f(None, ds)
            # sqrt of the factorized proposal density rho_d(x_d).
            sw = importance(None, ds).sqrt()
            # Weighted factors are used for stable finite differences.
            return phi * sw, phi, sw

        g, phi, sw = eval_g(dim_samples)
        gp, _, _ = eval_g([x + self.eps for x in dim_samples])
        gm, _, _ = eval_g([x - self.eps for x in dim_samples])

        # Central finite-difference approximation of d^2 phi / dx_d^2.
        d2_phi = (gp + gm - 2 * g) / self.eps ** 2
        d2_phi = d2_phi / torch.clamp(sw, min=1e-5)

        L, R, D, B = phi.shape

        # Same-mode 1D factors:
        # M^d_{lrs}=<phi_{d,l,r}, phi_{d,l,s}>,
        # K^d_{lrs}=<phi_{d,l,r}, -d^2 phi_{d,l,s}>.
        m_cross = torch.einsum("lrdb,lsdb->lrsd", phi, phi.detach()) / B
        k_cross = torch.einsum(
            "lrdb,lsdb->lrsd", phi, -d2_phi.detach()
        ) / B

        # Full mass matrix:
```

```
        # Sigma_{lk}=sum_{r,s} prod_d <phi_{d,l,r}, phi_{d,k,s}>.
        mass = torch.prod(
            torch.einsum("lrdb,ksdb->lrksd", phi, phi) / B,
            dim=-1
        ).sum(dim=(1, 3))

        # Additive Laplacian/kinetic assembly:
        # sum_k K^k_{lrs} prod_{d != k} M^d_{lrs}, then sum over r,s.
        mask = ~torch.eye(D, dtype=torch.bool, device=phi.device)
        prod_others = torch.prod(
            torch.where(
                mask, m_cross.unsqueeze(-2),
                torch.ones_like(m_cross.unsqueeze(-2))
            ),
            dim=-1
        )
        stiff = torch.sum(k_cross * prod_others, dim=-1).sum(dim=(1, 2))
        return 0, phi, mass, stiff


class Operator:
    def __init__(self, a):
        self.sep = SeparableMatrixAssembler(a.laplacian_eps)
        self.scale = a.operator_scale
        self.scale_kinetic = 1.0
        self.weights = None
        self.alphas = None

    def __call__(self, model, x, dim_samples=None, importance=None,
                 use_tensor_integral=True):
        if use_tensor_integral:
            # TPQ mass and kinetic diagonal.
            fs, phi, mass, stiff = self.sep(
                model, x, dim_samples, importance
            )

            # Potential TPQ assembly for
            # V(x)=sum_k omega_k prod_d exp(-alpha_k x_d^2):
            # sum_k omega_k sum_{r,s} prod_d
            # <phi_{d,l,r} phi_{d,l,s} exp(-alpha_k x_d^2)>.
            x_grid = torch.stack(dim_samples, dim=0)  # Shape (D,B_1D).
            v_fac = torch.exp(
                -self.alphas.view(-1, 1, 1) * x_grid.unsqueeze(0) ** 2
            )
            v_1d = oe.contract(
                "lrdb,lsdb,kdb->lrskd",
                phi, phi.detach(), v_fac,
                backend="torch"
            ) / phi.shape[-1]

            potential = torch.einsum(
                "lrsk,k->l", torch.prod(v_1d, dim=-1), self.weights
            )

            # Diagonal Pi for T=-H in the hydrogen experiment.
            h_diag = -(self.scale_kinetic * stiff + potential)
            return None, None, mass, self.scale * torch.diag(h_diag)
```

J.3 PyTorch Code of the Separable Potential Function Pre-training

Listing 3 gives the Gaussian mixture pre-training routine used to convert the Coulomb potential into a sum of separable Gaussian factors. This routine is run before the TPQ operator assembly above and returns the coefficients used as `weights` and `alphas`.

Listing 3: Pre-training the separable Gaussian mixture surrogate for the Coulomb potential.

```
def fit_potential(local_potential_ftn, device, K=15,
                  r_min=.01, r_max=60., steps=6001):
    # Fit |V(r)| by sum_k w_k exp(-alpha_k r^2).
    # Returning negative weights gives the attractive Coulomb sign and makes
    # each Gaussian separable:
    # exp(-alpha_k ||x||^2)=prod_d exp(-alpha_k x_d^2).
    import torch.optim as optim

    class Fitter(nn.Module):
        def __init__(self):
            super().__init__()
            self.log_alphas = nn.Parameter(
```

```
                torch.linspace(-2, 4, K, device=device) * 2.302
            )
            self.weights = nn.Parameter(torch.ones(K, device=device) * .1)

        def forward(self, r):
            return (
                torch.exp(
                    -torch.exp(self.log_alphas).unsqueeze(0)
                    * r.unsqueeze(1) ** 2
                ) * self.weights
            ).sum(1)

    model = Fitter().to(device)
    opt = optim.Adam(model.parameters(), lr=.05)

    # Fit on a logarithmic radial grid. Points are embedded as (r,0,0), so
    # the true radius is exactly ||x||=r.
    r = torch.logspace(
        torch.log10(torch.tensor(r_min, device=device)),
        torch.log10(torch.tensor(r_max, device=device)),
        steps,
        device=device
    )
    coords = torch.zeros((r.shape[0], 3), device=device)
    coords[:, 0] = r

    with torch.no_grad():
        v_true = torch.abs(local_potential_ftn(coords).squeeze())

    print("fitting GMM...")
    for epoch in range(steps):
        opt.zero_grad()
        loss = torch.mean(((model(r) - v_true) / v_true) ** 2)
        loss.backward()
        opt.step()
        if epoch % 500 == 0:
            print(f"Epoch {epoch}, Loss (Relative MSE): {loss.item():.6f}")

    # Negative weights approximate the attractive potential V(r)<0.
    return -model.weights.detach(), torch.exp(model.log_alphas).detach()
```

J.4 PyTorch Code of the Optimization Objectives in TPQ

Listings 4, 5, and 6 summarize the objective implementations used by the SpIN-TPQ, NeuralEF-TPQ, and NeuralSVD-TPQ experiments, respectively. In all cases, the matrices `mass`/`M` and `pi`/`H`/`Pi` are assembled by TPQ from 1D factors as described in Secs. 2.1–2.3.

Listing 4: SpIN-TPQ Rayleigh-trace loss with Cholesky gradient masks.

```
def spin_step(sigma, pi):
    # Cholesky-normalize the mass matrix Sigma = L L^T.
    chol = torch.linalg.cholesky(
        sigma + 1e-3 * torch.eye(sigma.shape[0], device=sigma.device)
    )
    chol_inv = torch.linalg.inv(chol)

    # Lambda = L^{-1} Pi L^{-T}; its trace is the SpIN objective.
    lam = chol_inv @ pi @ chol_inv.T
    return chol, chol_inv, lam, torch.diagonal(lam)


class SpINFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, sigma, pi):
        chol, chol_inv, lam, eigvals = spin_step(sigma, pi)

        # Multiplier masks used by the SpIN ordered-gradient rule.
        diag = torch.diag(torch.diagonal(chol_inv))
        gsigma = chol_inv.T @ torch.triu(lam @ diag)
        gpi = -chol_inv.T @ diag
        ctx.save_for_backward(gsigma, gpi)

        return torch.trace(lam), eigvals, chol, gsigma, gpi

    @staticmethod
    def backward(ctx, *grad_outputs):
        return ctx.saved_tensors
```

```

def compute_spin_tpq_loss(model, operator, x, dim_samples, importance):
    # TPQ returns mass = Sigma and ham = Pi.
    _, phi, mass, ham = operator(
        model, x, dim_samples, importance, use_tensor_integral=True
    )

    # The returned loss is Tr(Sigma^{-1} Pi) in the Cholesky basis.
    loss, eigvals, chol, gsigma, gpi = SpINFunction.apply(mass, ham)
    return loss, eigvals, chol, gsigma, gpi
```

Listing 5: NeuralEF-TPQ Rayleigh quotient with sequential soft penalties.

```
class NeuralEigenfunctions(nn.Module):
    def compute_loss_operator(
        self, operator, x, dim_samples, importance, use_tensor_integral=True
    ):
        # TPQ path directly returns Pi assembled from 1D quadratures.
        Tphi, phi, _, Pi = operator(
            self.model, x, dim_samples, importance, use_tensor_integral
        )

        # Dense fallback: estimate Pi_lm = <u_l, T u_m> by samples.
        if not (use_tensor_integral and Pi is not None):
            Pi = torch.einsum("bl,bm->lm", phi, Tphi.detach()) / phi.shape[0]

        # NeuralEF maximizes Pi_ll - sum_{m<l} Pi_lm^2 / Pi_mm.
        diag = torch.diag(Pi)
        loss = 0.0
        for l in range(Pi.shape[0]):
            if l:
                penalty = torch.sum(
                    Pi[l, :l] ** 2 / (torch.abs(diag[:l].detach()) + 1e-5)
                )
            else:
                penalty = 0.0
            loss -= 2 * diag[l] - penalty
        return loss
```

Listing 6: NeuralSVD-TPQ nested low-rank objective.

```
def masks(neigs, step):
    # NeuralSVD nesting weights. vector_mask is the coefficient a_l for the
    # diagonal operator term, and matrix_mask is Q_lm=min(a_l,a_m) for LoRA.
    ends = list(range(step, neigs + 1, step))  # End indices of nested subproblems.
    if neigs not in ends:
        ends.append(neigs)  # Always include the full rank-L objective.
    w = np.zeros(neigs)  # Raw nesting weights w_l, mostly zero for stepped nesting.
    w[np.array(ends) - 1] = 1.  # Activate one objective at each nested endpoint.
    v = torch.tensor(np.cumsum((w / w.sum())[::-1])[::-1].copy()).float()  # a_l=sum_{m>=l}
        w_m.
    return v, torch.minimum(v[:, None], v[None, :]).float()  # Q_lm=min(a_l,a_m).


class NestedLoRA(nn.Module):
    # NeuralSVD nested low-rank objective specialized to the self-adjoint
    # hydrogen EVD case. The left/right singular families are represented by
    # the same wavefunction model, so the mass regularizer becomes M * M.
    name = 'nestedlora'
    def __init__(self, model, neigs, step):
        super().__init__()
        self.model = model  # Shared self-adjoint branch f_l for both sides.
        self.vector_mask, self.matrix_mask = masks(neigs, step)  # Precompute a_l and Q_lm.
    def forward(self, *args):
        return self.model(*args)  # Forward to CP wavefunction model.
    def compute_loss_operator(self, operator, x, dim_samples, importance, use_tensor_integral=
        True):
        Tf, f, M, H = operator(self, x, dim_samples=dim_samples, importance=importance,
            use_tensor_integral=use_tensor_integral)  # Get Sigma/Pi or dense Tf/f.
        dev = M.device if M is not None else f.device  # Device for masks.
        v, mm = self.vector_mask.to(dev), self.matrix_mask.to(dev)  # Move a_l,Q_lm to GPU/CPU
            .
        if M is None or H is None:
            M = torch.einsum('bi...,bj...->ij', f, f) / f.shape[0]  # Dense fallback Sigma_ij
                =<f_i,f_j>.
            H = torch.einsum('bi...,bj...->ij', f, Tf.detach()) / f.shape[0]  # Dense fallback
                Pi_ij=<f_i,Tf_j>.
        # Minimize Q_lm Sigma_lm^2 - 4 a_l Pi_ll.
        # The factor 4 corresponds to the self-adjoint u=v reduction of the
```

```
        # NeuralSVD LoRA term -2 * <v_l, T u_l> with tied branches.
        return (mm * M * M).sum() - 4 * (v * H.diag()).sum(), dict(f=f, Tf=Tf)
```

J.5 PyTorch Code of a NeuralSVD-TPQ Example

Here we give a full, standalone code example of the NeuralSVD-TPQ for the 3D Hydrogen atom problem in Listing 7.

Listing 7: Standalone PyTorch implementation of NeuralSVD-TPQ for the 3D hydrogen atom problem.

```
import argparse
import csv, math, os, random, time
from contextlib import suppress
from functools import partial

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import opt_einsum as oe
import termplotlib as tpl
import torch
import torch.nn as nn
from mpl_toolkits.axes_grid1 import make_axes_locatable
from scipy.special import gamma, gammaln, genlaguerre, hyp2f1, lpmv
from torch.distributions.multivariate_normal import MultivariateNormal
from torch_ema import ExponentialMovingAverage
from tqdm import tqdm, trange

plt.style.use('ggplot')
ABC = "abcdefghijklmnopqrstuvwxyz"


# Standalone NeuralSVD-TPQ script for the 3D hydrogen Hamiltonian experiment.
# The physical Hamiltonian is H = -Delta + V, while the optimized operator is
# T = -H = Delta - V so that low-energy states become leading eigenvalues.
DEFAULT_CONFIG = dict(seed=4, parallel=True, ndim=3, val_eps=1, neigs=14, eval_freq=1000, rank=20,
                      laplacian_eps=0.1, batch_size=27000000, mlp_hidden_dims='64', log_dir='./log',
                      overwrite=False, print_freq=1000, num_iters=20000, lr=1e-4, rmsprop_decay=0.995,
                      momentum=0., ema_decay=0.99, hard_mul_const=1., sampling_scale=16, charge=1.0,
                      lim=50., operator_scale=100, operator_shift=0., fourier_mapping_size=1024,
                      fourier_scale=0.1, fourier_append_raw=False, loss=dict(neuralsvd=dict(step=1)))


class NS:
    # Minimal recursive namespace so dict configs can be accessed as a.b.c.
    def __init__(self, d):
        self.__dict__.update({k: NS(v) if isinstance(v, dict) else v for k, v in d.items()})


def dims(s): return [] if s == '' else list(map(int, s.split(',')))


def set_seed(seed):
    # Fix random sources for comparable TPQ runs and deterministic CuDNN kernels.
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)


def get_device(device_name=None):
    if device_name is not None:
        device = torch.device(device_name)
        if device.type == 'cuda' and not torch.cuda.is_available():
            raise RuntimeError("CUDA was requested but is not available.")
        return device
```

```
    if torch.cuda.is_available():
        print(f'There are {torch.cuda.device_count()} GPU(s) available.')
        print(f'Device name: {torch.cuda.get_device_name(0)}')
        return torch.device('cuda')
    print('No GPU available, using the CPU instead.')
    return torch.device('cpu')


def build_config(**overrides):
    # Merge command-line overrides into the default experiment configuration.
    cfg = dict(DEFAULT_CONFIG)
    cfg.update(overrides)
    if 'loss' in overrides and isinstance(overrides['loss'], dict):
        cfg['loss'] = overrides['loss']
    return NS(cfg)


def log_file(log_dir):
    f = open(os.path.join(log_dir, 'test.csv'), 'w', newline='')
    w = csv.DictWriter(f, fieldnames=['iter', 'train_loss', 'avg_train_loss', 'time'])
    w.writeheader()
    return f, w


class Fourier(nn.Module):
    # Gaussian Fourier feature map used by the 1D factor networks.
    # Raw coordinates are normally not appended in the reported hydrogen run.
    def __init__(self, input_dim, mapping_size, scale, append_raw=False):
        super().__init__()
        self.input_dim, self.append_raw = input_dim, append_raw
        self._B = nn.Parameter(2 * torch.pi * scale * torch.randn(input_dim, mapping_size).
             float(), requires_grad=False)
        self.feature_dim = 2 * mapping_size + (input_dim if append_raw else 0)

    def forward(self, x):
        x = x.reshape(x.shape[0], -1) if x.dim() == 3 else x
        y = x @ self._B.to(x.device)
        y = torch.cat([torch.sin(y), torch.cos(y)], 1)
        return torch.cat([y, x], 1) if self.append_raw else y


class ParallelMLP(nn.Module):
    # Batched collection of L independent 1D MLPs.
    # Parameters are shaped (L, out, in), so each target mode owns its own
    # factor network while the forward pass is vectorized over L.
    def __init__(self, input_dim, hidden, out_dim, neigs, feature_map):
        super().__init__()
        self.feature_map, self.activation = feature_map, nn.Softplus
        self.ws, self.bs = nn.ParameterList(), nn.ParameterList()
        prev = feature_map.feature_dim if feature_map else input_dim
        for h in hidden + [out_dim]:
            self.ws.append(nn.Parameter(math.sqrt(2. / prev) * torch.randn(neigs, h, prev)))
            self.bs.append(nn.Parameter(torch.zeros(neigs, h, 1)))
            prev = h

    def forward(self, x):
        # Returns factor values with axes arranged so that the last dimension
        # is the CP rank R and the middle dimension indexes eigenmodes L.
        x = self.feature_map(x) if self.feature_map is not None else x
        h = self.activation()(torch.einsum('lhd,bd->lhb', self.ws[0], x) + self.bs[0])
        for i in range(1, len(self.ws)):
            h = torch.einsum('lhp,lpb->lhb', self.ws[i], h) + self.bs[i]
            if i < len(self.ws) - 1:
                h = self.activation()(h)
        return h.permute(2, 0, 1).squeeze()


class CP_network(nn.Module):
    # This is the CP/variable-separable ansatz from the paper.
    def __init__(self, a):
        super().__init__()
        self.rank, self.neigs, self.ndim = a.rank, a.neigs, a.ndim
        self.net_list = nn.ModuleList([
            ParallelMLP(a.n_particles, dims(a.mlp_hidden_dims), a.rank, a.neigs,
                        Fourier(a.n_particles, a.fourier_mapping_size, a.fourier_scale, a.
                            fourier_append_raw))
            for _ in range(a.ndim)
        ])

    def _ensure_2d(self, x):
```

```
        return x.unsqueeze(0).unsqueeze(-1) if x.dim() == 0 else x if x.dim() > 1 else x.unsqueeze(-1)

    def forward(self, x=None, dim_samples=None):
        if dim_samples is not None:
            # TPQ path: evaluate only the D one-dimensional marginal grids.
            # Output shape is (L, R, D, B_1D).
            outs = []  # Stores one factor tensor phi_d for each coordinate direction.
            for d, net in enumerate(self.net_list):  # Loop over dimensions d=0,1,2.
                out = net(self._ensure_2d(dim_samples[d]))  # Evaluate the d-th 1D MLP on B_1D samples.
                outs.append(out.permute(*(list(range(out.dim()))[-2:] + list(range(out.dim()))[:-2])))  # Move (L,R) to front.
            return torch.stack(outs, dim=2)[:self.neigs]  # Combine factors into (L,R,D,B_1D).
        # Dense evaluation path: reconstruct f_l(x) on explicit 3D points by
        # multiplying factors across dimensions and summing over rank.
        xs = [x[..., d:d + 1] for d in range(self.ndim)] if isinstance(x, torch.Tensor) else x  # Split x into 1D coordinates.
        y = self.net_list[0](self._ensure_2d(xs[0]))  # Start product with the first coordinate factor.
        for d in range(1, self.ndim):  # Multiply remaining coordinate factors.
            y = y * self.net_list[d](self._ensure_2d(xs[d]))  # Pointwise product over CP factors.
        return y.sum(dim=-1)[..., :self.neigs]  # Sum over rank R to obtain L function values.


class WaveFunctions(nn.Module):
    # Lightweight wrapper for optional global scaling of all learned functions.
    def __init__(self, base, hard_mul_const=1.):
        super().__init__()
        self.base, self.hard_mul_const = base, hard_mul_const

    def forward(self, x, dim_samples=None):
        return self.hard_mul_const * self.base(x, dim_samples=dim_samples)


def cp_to_tensor(factors):
    # Materialize CP factors on the full Cartesian grid when validation or a
    # dense fallback needs explicit values: (L,R,D,B_1D) -> (B_1D^D,L).
    if not hasattr(factors, 'dim') or factors.dim() != 4:
        return factors
    L, R, D, B = factors.shape
    return torch.einsum(f"{','.join(f'lr{ABC[i]}' for i in range(D))}->l{ABC[:D]}", *factors.unbind(2)).reshape(L, -1).t()


def hydrogen_potential(x, charge=1.):
    # Coulomb potential V(x) = -Z / ||x|| with a small clamp at the origin.
    x = x.reshape(x.shape[0], -1)
    r = torch.clamp(x.norm(dim=1, p=2), min=1e-12)
    return -(charge / r).reshape(-1, 1)


def legendre_function(mu, lamb, z):
    return 1 / gamma(1 - mu) * ((1 + z) / (1 - z)) ** (mu / 2) * hyp2f1(-lamb, lamb + 1, 1 - mu, (1 - z) / 2)


def sph_harm(ells, ths):
    def unit(j, m, l, th):
        if j == 2:
            return np.sqrt((2 * l + 1) / 2 * gamma(l + m + 1) / gamma(l - m + 1)) * lpmv(-m, l, np.cos(th))
        return np.sqrt((2 * l + j - 1) / 2 * gamma(l + m + j - 1) / gamma(l - m + 1)) * \
            (np.sin(th) ** ((2 - j) / 2)) * legendre_function(-(m + (j - 2) / 2), l + (j - 2) / 2, np.cos(th))
    temp = np.zeros((len(ells), ths.shape[1]), dtype=complex)
    temp[0] = np.exp(1j * ells[0] * ths[0])
    for j in range(1, len(ells)):
        temp[j] = unit(j + 1, ells[j - 1], ells[j], ths[j])
    return temp.prod(axis=0) / np.sqrt(2 * np.pi)


def real_sph_harm(ells, ths):
    ells = np.array(ells).copy()
    positive = ells[0] > 0
    ells[0] = -abs(ells[0])
    y = sph_harm(ells, ths)
    if ells[0] == 0:
        return y.real
    return np.sqrt(2) * (1 if ells[0] % 2 == 0 else -1) * (y.imag if positive else y.real)
```

```python

class Hydrogen3D:
    # Analytical hydrogen spectrum/eigenfunctions used only for validation,
    # plotting, and degenerate-eigenspace alignment.
    def __init__(self, charge=1.):
        self.charge = charge

    def eigvals(self, neigs):
        q = np.array([n for n in range(1, int(np.ceil(neigs ** (1. / 3))) + 1) for _ in range(
            n * n)])
        return -self.charge ** 2 / (4 * q[:neigs] ** 2)

    def eigfunc(self, n, l, m, r, th, phi):
        a0, rho = 2 / self.charge, None
        rho = 2 * r / (n * a0)
        radial = np.sqrt((2 / (n * a0)) ** 3 / (2 * n)) * rho ** l * \
            np.exp(.5 * (-rho + gammaln(n - l) - gammaln(n + l + 1))) * genlaguerre(n - l - 1,
                2 * l + 1)(rho)
        return radial * real_sph_harm(np.array([m, l]), np.stack([phi, th]))


class VectorizedLaplacian:
    # Dense finite-difference Laplacian for validation/fallback paths.
    # Importance weights are folded in and divided back out so the same
    # weighted quadrature convention is used as in TPQ.
    def __init__(self, eps):
        self.eps = eps

    def __call__(self, f, xs, importance):
        def g(x):
            return importance(x).sqrt() * f(x)
        fs, D = g(xs), xs.shape[1]
        lap = -2 * D * fs
        for i in range(D):
            e = torch.zeros(1, D, device=xs.device)
            e[0, i] = self.eps
            lap += g(xs + e) + g(xs - e)
        sw = torch.clamp(importance(xs).sqrt(), min=1e-5)
        return lap / self.eps ** 2 / sw, fs / sw


class SeparableMatrixAssembler:
    # TPQ assembler for the diagonal Hamiltonian terms required by this
    # self-adjoint NeuralSVD experiment. It builds 1D mass/stiffness factors
    # without backpropagating through a full B_1D^3 graph.
    def __init__(self, eps):
        self.eps = eps

    def __call__(self, f, xs, dim_samples, importance):
        def eval_g(ds):
            phi = f(None, ds)  # CP factors phi_{l,r,d,b}, shape (L,R,D,B_1D).
            sw = importance(None, ds).sqrt()  # sqrt of factorized proposal density rho_d.
            return phi * sw, phi, sw  # Weighted factors are used for stable finite
                differences.
        g, phi, sw = eval_g(dim_samples)  # Evaluate original 1D factors.
        gp, _, _ = eval_g([x + self.eps for x in dim_samples])  # Forward-shift factors for
            second derivative.
        gm, _, _ = eval_g([x - self.eps for x in dim_samples])  # Backward-shift factors for
            second derivative.
        d2_phi = (gp + gm - 2 * g) / self.eps ** 2 / torch.clamp(sw, min=1e-5)  # Central
            finite difference d^2 phi/dx_d^2.
        L, R, D, B = phi.shape  # L modes, R CP rank, D dimensions, B=B_1D samples.
        # m_cross and k_cross are per-mode rank-pair 1D factors:
        # M^d_{lrs}=<phi_{d,l,r},phi_{d,l,s}> and
        # K^d_{lrs}=<phi_{d,l,r},-d2 phi_{d,l,s}>.
        m_cross = torch.einsum('lrdb,lsdb->lrsd', phi, phi.detach()) / B  # Same-mode 1D mass
            factors.
        k_cross = torch.einsum('lrdb,lsdb->lrsd', phi, -d2_phi.detach()) / B  # Same-mode
            kinetic factors.
        # Full mass matrix Sigma_{lm} assembled as sum_{r,s} prod_d M^{lm,d}_{rs}.
        mass = torch.prod(torch.einsum('lrdb,ksdb->lrksd', phi, phi) / B, dim=-1).sum(dim=(1,
            3))  # Sigma_{lk}.
        mask = ~torch.eye(D, dtype=torch.bool, device=phi.device)  # Select dimensions d != k
            in products.
        # Additive Laplacian assembly:
        # sum_k K^k_{lrs} prod_{d != k} M^d_{lrs}, then sum over rank pairs.
        prod_others = torch.prod(torch.where(mask, m_cross.unsqueeze(-2), torch.ones_like(
            m_cross.unsqueeze(-2))), dim=-1)  # Product over non-active dimensions.
        stiff = torch.sum(k_cross * prod_others, dim=-1).sum(dim=(1, 2))  # Diagonal kinetic
            Pi_ll before potential.
```

```
        return 0, phi, mass, stiff  # First return slot is unused in TPQ mode.


def fit_potential(local_potential_ftn, device, K=15, r_min=.01, r_max=60., steps=6001):
    # Fit |V(r)| by sum_k w_k exp(-alpha_k r^2). Returning negative weights
    # gives the attractive Coulomb sign and makes each Gaussian separable:
    # exp(-alpha_k ||x||^2) = prod_d exp(-alpha_k x_d^2).
    import torch.optim as optim
    class Fitter(nn.Module):
        def __init__(self):
            super().__init__()
            self.log_alphas = nn.Parameter(torch.linspace(-2, 4, K, device=device) * 2.302)  # Log-widths alpha_k.
            self.weights = nn.Parameter(torch.ones(K, device=device) * .1)  # Mixture amplitudes w_k.
        def forward(self, r):
            return (torch.exp(-torch.exp(self.log_alphas).unsqueeze(0) * r.unsqueeze(1) ** 2) * self.weights).sum(1)  # Sum_k w_k exp(-alpha_k r^2).
    model, opt = Fitter().to(device), None  # Small standalone surrogate model.
    opt = optim.Adam(model.parameters(), lr=.05)  # Fast offline fit for GMM coefficients.
    r = torch.logspace(torch.log10(torch.tensor(r_min, device=device)), torch.log10(torch.tensor(r_max, device=device)), steps, device=device)  # Radial grid.
    coords = torch.zeros((r.shape[0], 3), device=device)  # Embed radii on x-axis for V(||x||).
    coords[:, 0] = r  # Points are (r,0,0), so ||x||=r.
    with torch.no_grad():
        v_true = torch.abs(local_potential_ftn(coords).squeeze())  # Fit magnitude; sign is restored at return.
    print("fitting GMM...")
    for epoch in range(steps):  # Optimize Gaussian widths and weights.
        opt.zero_grad()  # Clear GMM gradients.
        loss = torch.mean(((model(r) - v_true) / v_true) ** 2)  # Relative MSE over radii.
        loss.backward()  # Differentiate surrogate fitting loss.
        opt.step()  # Update alpha_k and w_k.
        if epoch % 500 == 0:
            print(f"Epoch {epoch}, Loss (Relative MSE): {loss.item():.6f}")
    return -model.weights.detach(), torch.exp(model.log_alphas).detach()  # Negative weights approximate V(r)<0.


class Operator:
    # Applies T = Delta - V and the final affine spectral scaling. In TPQ mode,
    # it returns the mass matrix and a diagonal operator matrix H/Pi assembled
    # from 1D factors plus the Gaussian mixture potential.
    def __init__(self, a):
        self.lap, self.sep = VectorizedLaplacian(a.laplacian_eps), SeparableMatrixAssembler(a.laplacian_eps)
        self.local_potential_ftn = partial(hydrogen_potential, charge=a.charge)
        self.scale, self.shift, self.scale_kinetic, self.n_particles = a.operator_scale, a.operator_shift, 1., a.n_particles

    def __call__(self, model, x, dim_samples=None, importance=None, use_tensor_integral=True):
        if use_tensor_integral:
            # Fit the separable potential surrogate once, then reuse its
            # coefficients throughout training.
            if not hasattr(self, 'weights'):
                print("Starting GMM pre-training for Potential Function...")
                self.weights, self.alphas = fit_potential(self.local_potential_ftn, x.device)
                print("GMM pre-training completed!")
            fs, phi, mass, stiff = self.sep(model, x, dim_samples, importance)  # TPQ mass and kinetic diagonal.
            x_grid = torch.stack(dim_samples, dim=0)  # Shape (D,B_1D).
            # Potential TPQ assembly:
            # sum_k omega_k sum_{r,s} prod_d <phi_{l,r} phi_{l,s}
            # exp(-alpha_k x_d^2)>.
            v_fac = torch.exp(-self.alphas.view(-1, 1, 1) * x_grid.unsqueeze(0) ** 2)  # Gaussian factors per k,d,b.
            v_1d = oe.contract('lrdb,lsdb,kdb->lrskd', phi, phi.detach(), v_fac, backend='torch') / phi.shape[-1]  # 1D potential integrals.
            h_diag = -(self.scale_kinetic * stiff + torch.einsum('lrsk,k->l', torch.prod(v_1d, dim=-1), self.weights))  # Pi_ll for T=-H.
            return None, None, mass, self.scale * torch.diag(h_diag)  # Return Sigma and diagonal Pi.
        # Dense fallback: evaluate T f pointwise and let the caller assemble
        # matrices by empirical inner products.
        lap, f = self.lap(model, x, importance)
        pot = self.local_potential_ftn(x.reshape(x.shape[0], self.n_particles, -1)) * f
        return self.scale * (lap - pot) + self.shift * f, f, None, None


def masks(neigs, step):
```

```
    # NeuralSVD nesting weights. vector_mask is the coefficient a_l for the
    # diagonal operator term, and matrix_mask is Q_lm=min(a_l,a_m) for LoRA.
    ends = list(range(step, neigs + 1, step))  # End indices of nested subproblems.
    if neigs not in ends:
        ends.append(neigs)  # Always include the full rank-L objective.
    w = np.zeros(neigs)  # Raw nesting weights w_l, mostly zero for stepped nesting.
    w[np.array(ends) - 1] = 1.  # Activate one objective at each nested endpoint.
    v = torch.tensor(np.cumsum((w / w.sum())[::-1])[::-1].copy()).float()  # a_l=sum_{m>=l} w_m.
    return v, torch.minimum(v[:, None], v[None, :]).float()  # Q_lm=min(a_l,a_m).


class NestedLoRA(nn.Module):
    # NeuralSVD nested low-rank objective specialized to the self-adjoint
    # hydrogen EVD case. The left/right singular families are represented by
    # the same wavefunction model, so the mass regularizer becomes M * M.
    name = 'nestedlora'
    def __init__(self, model, neigs, step):
        super().__init__()
        self.model = model  # Shared self-adjoint branch f_l for both sides.
        self.vector_mask, self.matrix_mask = masks(neigs, step)  # Precompute a_l and Q_lm.
    def forward(self, *args):
        return self.model(*args)  # Forward to CP wavefunction model.
    def compute_loss_operator(self, operator, x, dim_samples, importance, use_tensor_integral=True):
        Tf, f, M, H = operator(self, x, dim_samples=dim_samples, importance=importance, use_tensor_integral=use_tensor_integral)  # Get Sigma/Pi or dense Tf/f.
        dev = M.device if M is not None else f.device  # Device for masks.
        v, mm = self.vector_mask.to(dev), self.matrix_mask.to(dev)
        if M is None or H is None:
            M = torch.einsum('bi...,bj...->ij', f, f) / f.shape[0]  # Dense fallback Sigma_ij=<f_i,f_j>.
            H = torch.einsum('bi...,bj...->ij', f, Tf.detach()) / f.shape[0]  # Dense fallback Pi_ij=<f_i,Tf_j>.
        # Minimize Q_lm Sigma_lm^2 - 4 a_l Pi_ll.
        # The factor 4 corresponds to the self-adjoint u=v reduction of the
        # NeuralSVD LoRA term -2 * <v_l, T u_l> with tied branches.
        return (mm * M * M).sum() - 4 * (v * H.diag()).sum(), dict(f=f, Tf=Tf)


def align(preds, targets, degeneracies):
    # Align each degenerate hydrogen eigenspace by an orthogonal Procrustes
    # rotation before comparing learned functions to analytical references.
    preds, targets = torch.as_tensor(preds).float(), torch.as_tensor(targets).float()  # Convert arrays to tensors.
    parts, i = [], 0  # Accumulate aligned blocks.
    for k in degeneracies:
        a, b = preds[:, i:i + k], targets[:, i:i + k].to(preds.device)  # One degenerate subspace.
        u, _, vh = torch.linalg.svd(a.T @ b)  # Orthogonal Procrustes rotation.
        parts.append(a @ (u @ vh))  # Rotate learned basis into target basis.
        i += k  # Advance to next energy level.
    return torch.cat(parts, 1)  # Concatenate aligned eigenspaces.


def spectrum(model, dataloader, operator, imp_train, imp_val, device):
    # Validation-time spectrum estimation on a uniform grid. The training
    # Gaussian importance density is converted to the validation density before
    # forming covariance and operator matrices.
    t0, n, cov, quad, eigfuncs = time.time(), 0, 0., 0., []  # Running quadrature accumulators.
    for x, true_eigs, _ in tqdm(dataloader):
        x = x.to(device)  # Validation batch on target device.
        sw_train, sw_val = imp_train(x).sqrt(), imp_val(x).sqrt()  # Convert Gaussian-weighted values to uniform measure.
        Tphi, phi, _, _ = operator(model, x, importance=imp_train, use_tensor_integral=False)  # Dense Tphi for validation.
        Tphi, phi = Tphi.detach(), phi.detach()  # Validation only; no gradients.
        eigfuncs.append(sw_train * phi)  # Store train-measure weighted functions for visualization.
        phi, Tphi = torch.nan_to_num(sw_train / sw_val * phi), torch.nan_to_num(sw_train / sw_val * Tphi)  # Importance correction.
        Tphi[torch.where(torch.all(torch.isclose(x, torch.zeros_like(x[0])), dim=1))] *= 0.  # Suppress Coulomb-origin artifact.
        cov, quad, n = cov + phi.T @ phi, quad + phi.T @ Tphi, n + len(x)  # Accumulate Sigma and Pi.
    cov, quad = cov / n, quad / n  # Average over validation points.
    print(f"Took {time.time() - t0}s to compute spectrum with data of size {n}")
    eigfuncs, true = torch.cat(eigfuncs, 0).cpu().numpy(), torch.cat(true_eigs, 1).cpu().numpy()  # Gather full grids.
    deg, used, k = [], 0, 1  # Degeneracy bookkeeping.
```

```
    while used + k * k <= eigfuncs.shape[1]:
        # 3D hydrogen has n^2-fold degeneracy for the n-th energy level.
        deg.append(k * k)  # Append current energy-level multiplicity.
        used += k * k  # Count modes covered so far.
        k += 1  # Move to next principal quantum number.
    if used < eigfuncs.shape[1]:
        deg.append(eigfuncs.shape[1] - used)  # Partial final degenerate block if needed.
        eigfuncs = eigfuncs[:, :used]  # Keep complete blocks for alignment.
    cov, quad = cov.cpu().numpy(), quad.cpu().numpy()  # Move matrices to NumPy for diagnostics.
    vals, norms = np.diag(quad) / np.diag(cov), np.diag(cov)  # Rayleigh quotient and squared norms.
    return dict(eigvals=vals, norms=norms, quad=quad, cov=cov / (np.sqrt(norms[:, None]) @ np.sqrt(norms[:, None]).T),
                eigfuncs=eigfuncs / np.sqrt(norms).reshape(1, -1), eigfuncs_true=true,
                aligned_eigfuncs=align(eigfuncs, true, deg))


def plot_3d_scatter_eigfuncs(eigfuncs, log_dir, tag="", threshold=1e-3, alpha=0.4):
    # Quick sparse 3D scatter visualization of learned eigenfunction support.
    if hasattr(eigfuncs, 'detach'):
        eigfuncs = eigfuncs.detach().cpu().numpy()
    L, vmax = eigfuncs.shape[1], 0.01
    max_pts_per = 5000
    fig = plt.figure(figsize=(L * 4 + 1, 4))
    for i in range(L):
        n = int(np.round(eigfuncs[:, i].shape[0] ** (1 / 3)))
        vals = eigfuncs[:, i].reshape(n, n, n)
        mask = np.abs(vals) > threshold
        ii, jj, kk = np.nonzero(mask)
        vv = vals[mask]
        if len(vv) > max_pts_per:
            idx = np.random.choice(len(vv), max_pts_per, replace=False)
            ii, jj, kk, vv = ii[idx], jj[idx], kk[idx], vv[idx]
        ax = fig.add_subplot(1, L, i + 1, projection="3d")
        sc = ax.scatter(ii, jj, kk, c=vv, cmap="seismic", vmin=-vmax, vmax=vmax, alpha=alpha,
                        marker="o", s=15)
        ax.axis("off")
        ax.set_title(f"Eigfunc {i}")
    fig.suptitle(tag)
    fig.colorbar(sc, shrink=0.7, pad=0.05, label="Eigenfunction Value")
    fig.tight_layout()
    fig.savefig(f"{log_dir}/eigfuncs_3d_{tag}.png", dpi=100, pil_kwargs={"compress_level": 1})
    plt.close(fig)


def plot_orth(ax, data):
    mesh = ax.pcolormesh(data[::-1, :], edgecolors='black', linewidth=0.005, cmap='gray', vmin=0, vmax=1)
    ax.set_aspect('equal')
    ax.set_xticks([])
    ax.set_yticks([])
    ax.tick_params(labelbottom=False, labelleft=False)
    return mesh


def plot_spectrum(out, log_dir, tag, truth):
    # Save eigenvalue/norm curves and the normalized covariance matrix, which
    # diagnoses whether the learned modes are close to orthogonal.
    try:
        fig = tpl.figure()
        for y in (out['eigvals'], out['norms']):
            fig.plot(np.arange(1, len(y) + 1), y, label=f'sum={y.sum():.2f}', width=80, height=20)
        fig.show()
    except Exception:
        print("Warning: Something went wrong with termplot")
    fig, axes = plt.subplots(1, 2, figsize=(10, 4))
    for key, y in {'RQ': out['eigvals'], 'Norms^2': out['norms']}.items():
        axes[0].plot(y, marker='o', label=f'{key}(sum={y.sum():.2f})')
        err = np.abs(y - truth) / np.abs(truth)
        print(err)
        print(f"{key} error: {np.mean(err):.4f}")
    axes[0].plot(truth, marker='x', label='ground truth')
    axes[0].legend()
    axes[0].set_title('Spectrum')
    axes[0].set_xlim([0, len(out['eigvals']) - 1])
    axes[0].set_ylim([np.min(truth) - 5, np.max(truth) + 5])
    axes[0].grid('on')
    mesh = plot_orth(axes[1], np.abs(out['cov']))
    axes[1].set_title(r'Orthogonality ($f(x)$)')
```

```
    axes[1].grid('off')
    fig.colorbar(mesh, cax=make_axes_locatable(axes[-1]).append_axes('right', size='5%', pad=0.05), orientation='vertical')
    fig.suptitle(tag)
    fig.tight_layout()
    plt.savefig(f'{log_dir}/spectrum_{tag}.png')
    plt.close()


def make_data(a, device):
    # Gaussian-grid TPQ training data. B_1D samples are drawn per coordinate;
    # their Cartesian product is materialized for interfaces, while model
    # evaluation in the TPQ loss uses only the factor lists ds.
    B = int(a.batch_size ** (1 / a.ndim))  # Convert effective batch B_1D^D to marginal size B_1D.
    def train_batch():
        ds = [a.sampling_scale * torch.randn(B, device=device) for _ in range(a.ndim)]  # Independent 1D Gaussian samples.
        return torch.stack(torch.meshgrid(ds, indexing='ij'), dim=-1).reshape(-1, a.ndim).unsqueeze(1), ds  # Product grid plus factors.
    mvn = MultivariateNormal(torch.zeros(a.n_particles * a.ndim, device=device),
                             a.sampling_scale ** 2 * torch.eye(a.n_particles * a.ndim, device=device))  # Full Gaussian density for dense path.
    def imp_train(x, dim_samples=None):
        if dim_samples is None:
            return mvn.log_prob(x.view(x.shape[0], -1)).exp().view(-1, 1)  # rho(x) for explicit 3D points.
        # Factorized Gaussian density rho_d(x_d), broadcastable to (L,R,D,B_1D).
        scale = 1 / (a.sampling_scale * (2 * np.pi) ** .5)  # 1D Gaussian normalizer.
        vals = [scale * torch.exp(-.5 / a.sampling_scale ** 2 * s ** 2) for s in dim_samples]  # rho_d samples.
        return torch.stack(vals, dim=-1).T.unsqueeze(0).unsqueeze(0)  # Shape (1,1,D,B_1D).
    # Uniform validation grid and analytical eigenfunctions for diagnostics.
    grid = np.arange(-a.lim, a.lim, max(a.val_eps, .5))  # 1D validation coordinates.
    val = torch.tensor(np.array(list(zip(*[xx.flatten() for xx in np.meshgrid(grid, grid, grid, indexing='ij')])))), device=device).float()  # Full 3D grid.
    xyz = val.cpu().numpy()  # NumPy copy for analytical hydrogen functions.
    r = np.sqrt((xyz ** 2).sum(1))  # Radius.
    th, phi = np.arccos(xyz[:, 2] / np.where(r == 0, 1e-10, r)), np.arctan2(xyz[:, 1], xyz[:, 0])  # Spherical angles.
    h3d = Hydrogen3D(a.charge)  # Analytical reference object.
    qnums = [(n, l, m) for n in range(1, int((3 * a.neigs) ** (1. / 3.)) + 1) for l in range(n) for m in range(-l, l + 1)]  # Quantum numbers.
    eigs = [torch.from_numpy(h3d.eigfunc(n, l, m, r, th, phi)).unsqueeze(1) for n, l, m in qnums]  # True eigenfunctions.
    def val_batches():
        for i in range(int(np.ceil(len(val) / 4096))):
            yield val[i * 4096:min((i + 1) * 4096, len(val))], eigs, 0.  # Chunk validation to limit memory.
    def imp_val(x):
        return (1 / (2 * a.lim) ** a.ndim * torch.ones(x.shape[0], 1, device=x.device)).float()  # Uniform validation density.
    return train_batch, val_batches, imp_train, imp_val  # Return closures used by train/eval.


def train(a, method, operator, train_batch, val_batches, lw, lf, device, imp_train, imp_val, truth):
    # Main RMSProp loop. The training loss uses TPQ matrix assembly, and EMA
    # parameters are used for periodic spectrum evaluation.
    opt = torch.optim.RMSprop(method.parameters(), lr=a.lr, alpha=a.rmsprop_decay, eps=1e-10, weight_decay=0, momentum=a.momentum)  # Optimizer from paper settings.
    ema, all_vals, all_norms, total, start = ExponentialMovingAverage(method.parameters(), decay=a.ema_decay), [], [], 0., time.time()  # EMA and logs.
    step_times = []  # Per-iteration wall-clock times for speed reporting.
    for it in trange(a.num_iters, desc='training'):
        step_start = time.perf_counter()  # Timer for this optimization step.
        method.train()  # Enable training mode.
        opt.zero_grad()  # Clear gradients.
        x, ds = train_batch()  # x is product grid; ds are 1D TPQ samples.
        with suppress():
            loss, _ = method.compute_loss_operator(operator, x.reshape(x.shape[0], -1).to(device), ds, imp_train, True)  # TPQ LoRA loss.
        loss.backward()  # Backprop through 1D factors and TPQ contractions.
        opt.step()  # Update factor-network parameters.
        ema.update()  # Track smoothed parameters for evaluation.
        if device.type == 'cuda':
            torch.cuda.synchronize(device)  # Make GPU timing accurate.
        step_times.append(time.perf_counter() - step_start)  # Save elapsed seconds.
        total += loss.detach().item()  # Accumulate scalar loss.
        if (it + 1) % a.print_freq == 0:
```

```
            row = {'iter': it + 1, 'train_loss': loss.item(), 'avg_train_loss': total / (it + 1), 'time': time.time() - start}  # CSV row.
            print(row)  # Console progress.
            print(f'learning rate is {[a.lr]}')
            lw.writerow(row)  # Persist training log.
            lf.flush()  # Flush so interrupted runs keep logs.
        if (it + 1) % a.eval_freq == 0:
            method.eval()  # Switch to evaluation mode.
            with ema.average_parameters():
                # Evaluation switches to the dense finite-difference operator
                # on validation batches to report Rayleigh-quotient spectra.
                out = spectrum(method, val_batches(), operator, imp_train, imp_val, device)
            plot_spectrum(out, a.log_dir, f'it{it + 1}', truth)  # Save spectrum and orthogonality plot.
            print(f"it{it + 1} eigvals: {out['eigvals']}")
            print(f"it{it + 1} norms: {out['norms']}")
            all_vals.append(out['eigvals'])  # Store spectrum history.
            all_norms.append(out['norms'])  # Store norm history.
            n_1d = int(2 * a.lim / max(a.val_eps, 0.5))  # Validation grid side length.
            np.savez(f'{a.log_dir}/checkpoint_it{it+1}.npz',
                     eigvals=out['eigvals'], eigvals_norm=out['norms'],
                     norms=out['norms'],
                     eigfuncs=out['eigfuncs'],
                     eigfuncs_grid=out['eigfuncs'].reshape(n_1d, n_1d, n_1d, -1).transpose(3, 0, 1, 2),
                     elapsed=time.time() - start)  # Save numerical arrays for later plotting.
            plot_3d_scatter_eigfuncs(out['eigfuncs'], a.log_dir, f'it{it + 1}')  # Native learned functions.
            plot_3d_scatter_eigfuncs(out['eigfuncs_true'], a.log_dir, f'_true_it{it + 1}')  # Analytical references.
            plot_3d_scatter_eigfuncs(out['aligned_eigfuncs'], a.log_dir, f'_aligned_it{it + 1}')  # Procrustes-aligned functions.
    return all_vals, all_norms, step_times  # Return training history.

def plot_3d_slice_eigfuncs(eigfuncs, log_dir, tag='', slice_axis=2, slice_idx=None):
    """Plot 2D slices of 3D eigenfunctions."""
    L = eigfuncs.shape[1]
    fig, axes = plt.subplots(1, L, figsize=(L * 2, 2.5))
    if L == 1:
        axes = [axes]
    for i in range(L):
        n = int(np.round(eigfuncs[:, i].shape[0] ** (1.0 / 3.0)))
        if slice_idx is None:
            current_slice_idx = n // 2
        else:
            current_slice_idx = slice_idx
        eigfunc_3d = eigfuncs[:, i].reshape(n, n, n)
        if slice_axis == 0:
            eigfunc_slice = eigfunc_3d[current_slice_idx, :, :]
        elif slice_axis == 1:
            eigfunc_slice = eigfunc_3d[:, current_slice_idx, :]
        else:
            eigfunc_slice = eigfunc_3d[:, :, current_slice_idx]
        axes[i].imshow(eigfunc_slice, interpolation='none', cmap='coolwarm')
        axes[i].grid(False)
        axes[i].axis('off')
    fig.suptitle(f"{tag} (axis={slice_axis}, idx={current_slice_idx})")
    fig.tight_layout()
    plt.savefig(f'{log_dir}/eigfuncs_{tag}.png', bbox_inches='tight')
    plt.close(fig)


def finalize_log_dir(a, suffix=None):
    # Include the high-impact TPQ settings in the directory name for later
    # comparison across rank, batch size, and nesting step.
    a.n_particles = 1
    b_1d = int(round(a.batch_size ** (1 / a.ndim)))
    base = (
        f'{time.strftime("%Y%m%d_%H%M%S")}_'
        f'n{a.ndim}_k{a.neigs}_r{a.rank}_'
        f'bld{b_1d}_b{a.batch_size}_s{a.loss.neuralsvd.step}_gmm'
    )
    if suffix:
        base = f'{base}_{suffix}'
    a.log_dir = os.path.join(a.log_dir, base)
    if os.path.exists(a.log_dir) and not a.overwrite:
        raise ValueError(f"{a.log_dir} already exists and overwrite is not permitted")
    os.makedirs(a.log_dir, exist_ok=True)
    print(f'creating file {a.log_dir}')
```

```python
def evaluate_method(method, operator, val_batches, device, imp_train, imp_val):
    # Final dense-grid evaluation helper.
    method.eval()
    with torch.no_grad():
        return spectrum(method, val_batches(), operator, imp_train, imp_val, device)


def summarize_speed(csv_path):
    # Fallback speed summary from the CSV log; train() also records per-step
    # wall-clock timings for a more direct mean.
    rows = []
    with open(csv_path, 'r', newline='') as handle:
        for row in csv.DictReader(handle):
            rows.append(row)
    if not rows:
        return dict(mean_it_s=np.nan, mean_sec_per_iter=np.nan)
    last = rows[-1]
    total_time = float(last['time'])
    total_iter = int(last['iter'])
    return dict(mean_it_s=total_iter / total_time if total_time > 0 else np.nan,
                mean_sec_per_iter=total_time / total_iter if total_iter > 0 else np.nan)


def run_training(a, device=None, log_suffix=None):
    # End-to-end experiment driver: data, model, GMM pretraining, optimization,
    # final evaluation, and metric serialization.
    set_seed(a.seed)
    device = get_device(device) if isinstance(device, str) or device is None else device
    finalize_log_dir(a, suffix=log_suffix)
    lf, lw = log_file(a.log_dir)
    # Negate analytical hydrogen energies because the learned operator is -H.
    truth = a.operator_scale * -Hydrogen3D(a.charge).eigvals(a.neigs) + a.operator_shift
    print('Ground truth eigenvalues:', truth)
    train_batch, val_batches, imp_train, imp_val = make_data(a, device)
    method = NestedLoRA(WaveFunctions(CP_network(a), a.hard_mul_const), a.neigs, a.loss.
        neuralsvd.step).to(device)
    operator = Operator(a)
    if not hasattr(operator, 'weights'):
        print("Starting GMM pre-training for Potential Function...")
        operator.weights, operator.alphas = fit_potential(operator.local_potential_ftn, device
            )
        print("GMM pre-training completed!")
    vals, norms, step_times = train(a, method, operator, train_batch, val_batches, lw, lf,
        device, imp_train, imp_val, truth)
    np.savez(f'{a.log_dir}/stats.npz', all_eigvals=vals, all_norms=norms)
    final_out = evaluate_method(method, operator, val_batches, device, imp_train, imp_val)
    eigval_rel_error = np.abs(final_out['eigvals'] - truth) / np.maximum(np.abs(truth), 1e-12)
    speed = summarize_speed(os.path.join(a.log_dir, 'test.csv'))
    sec = np.asarray(step_times, dtype=np.float64)
    if sec.size:
        speed = dict(mean_it_s=float(np.mean(1.0 / sec)), mean_sec_per_iter=float(np.mean(sec)
            ))
    summary = dict(
        method='GMM',
        batch_size=a.batch_size,
        b_1d=int(round(a.batch_size ** (1 / a.ndim))),
        eigvals=final_out['eigvals'],
        truth=truth,
        eigval_rel_error=eigval_rel_error,
        min_eigval_rel_error=float(np.min(eigval_rel_error)),
        mean_eigval_rel_error=float(np.mean(eigval_rel_error)),
        max_eigval_rel_error=float(np.max(eigval_rel_error)),
        mean_it_s=float(speed['mean_it_s']),
        mean_sec_per_iter=float(speed['mean_sec_per_iter']),
        log_dir=a.log_dir,
    )
    np.savez(f'{a.log_dir}/final_eval.npz', **{k: v for k, v in summary.items() if k != '
        log_dir'})
    lf.close()
    return summary


def parse_args():
    parser = argparse.ArgumentParser(description='Train NeuralSVD-TPQ for 3D hydrogen with GMM
        potential.')
    parser.add_argument('--seed', type=int, default=DEFAULT_CONFIG['seed'])
    parser.add_argument('--ndim', type=int, default=DEFAULT_CONFIG['ndim'])
    parser.add_argument('--val-eps', type=float, default=DEFAULT_CONFIG['val_eps'])
    parser.add_argument('--neigs', type=int, default=DEFAULT_CONFIG['neigs'])
    parser.add_argument('--eval-freq', type=int, default=DEFAULT_CONFIG['eval_freq'])
    parser.add_argument('--rank', type=int, default=DEFAULT_CONFIG['rank'])
```

```
    parser.add_argument('--laplacian-eps', type=float, default=DEFAULT_CONFIG['laplacian_eps'])
    parser.add_argument('--batch-size', type=int, default=DEFAULT_CONFIG['batch_size'])
    parser.add_argument('--mlp-hidden-dims', default=DEFAULT_CONFIG['mlp_hidden_dims'])
    parser.add_argument('--log-dir', default=DEFAULT_CONFIG['log_dir'])
    parser.add_argument('--overwrite', action='store_true')
    parser.add_argument('--print-freq', type=int, default=DEFAULT_CONFIG['print_freq'])
    parser.add_argument('--num-iters', type=int, default=DEFAULT_CONFIG['num_iters'])
    parser.add_argument('--lr', type=float, default=DEFAULT_CONFIG['lr'])
    parser.add_argument('--rmsprop-decay', type=float, default=DEFAULT_CONFIG['rmsprop_decay'])
    parser.add_argument('--momentum', type=float, default=DEFAULT_CONFIG['momentum'])
    parser.add_argument('--ema-decay', type=float, default=DEFAULT_CONFIG['ema_decay'])
    parser.add_argument('--hard-mul-const', type=float, default=DEFAULT_CONFIG['hard_mul_const'])
    parser.add_argument('--sampling-scale', type=float, default=DEFAULT_CONFIG['sampling_scale'])
    parser.add_argument('--charge', type=float, default=DEFAULT_CONFIG['charge'])
    parser.add_argument('--lim', type=float, default=DEFAULT_CONFIG['lim'])
    parser.add_argument('--operator-scale', type=float, default=DEFAULT_CONFIG['operator_scale'])
    parser.add_argument('--operator-shift', type=float, default=DEFAULT_CONFIG['operator_shift'])
    parser.add_argument('--fourier-mapping-size', type=int, default=DEFAULT_CONFIG['fourier_mapping_size'])
    parser.add_argument('--fourier-scale', type=float, default=DEFAULT_CONFIG['fourier_scale'])
    parser.add_argument('--fourier-append-raw', action='store_true')
    parser.add_argument('--step', type=int, default=DEFAULT_CONFIG['loss']['neuralsvd']['step'])
    parser.add_argument('--device', default=None)
    parser.add_argument('--log-suffix', default=None)
    return parser.parse_args()


def args_to_config(args):
    return build_config(
        seed=args.seed, ndim=args.ndim, val_eps=args.val_eps, neigs=args.neigs, eval_freq=args.eval_freq,
        rank=args.rank, laplacian_eps=args.laplacian_eps, batch_size=args.batch_size,
        mlp_hidden_dims=args.mlp_hidden_dims, log_dir=args.log_dir, overwrite=args.overwrite,
        print_freq=args.print_freq, num_iters=args.num_iters, lr=args.lr, rmsprop_decay=args.rmsprop_decay,
        momentum=args.momentum, ema_decay=args.ema_decay, hard_mul_const=args.hard_mul_const,
        sampling_scale=args.sampling_scale, charge=args.charge, lim=args.lim,
        operator_scale=args.operator_scale, operator_shift=args.operator_shift,
        fourier_mapping_size=args.fourier_mapping_size, fourier_scale=args.fourier_scale,
        fourier_append_raw=args.fourier_append_raw, loss=dict(neuralsvd=dict(step=args.step)))


def main():
    args = parse_args()
    summary = run_training(args_to_config(args), device=args.device, log_suffix=args.log_suffix)
    print(f"Final min eigval relative error: {summary['min_eigval_rel_error']:.6e}")
    print(f"Final mean eigval relative error: {summary['mean_eigval_rel_error']:.6e}")
    print(f"Final max eigval relative error: {summary['max_eigval_rel_error']:.6e}")
    print(f"Mean it/s: {summary['mean_it_s']:.6e}")


if __name__ == '__main__':
    main()
```

Acknowledgments on AI use. While the main technical contents and conceptualizations were made by humans, AI was used to polish writing and assist coding. We have reviewed all AI-assisted work. The authors take responsibility for the final content of this work.