FLOWSMO: SCALABLE END-TO-END SPARSE MATRIX OPERATORS VIA LATENT FLOW MATCHING

Anonymous authors
Paper under double-blind review

ABSTRACT

Large-scale scientific computing often requires repeatedly applying matrix operators $f(A)$ associated with sparse matrices A, including inverse and inverse-square-root operators. Classical algebraic methods, including incomplete factorizations, Krylov subspace methods, and polynomial or rational approximations, remain effective but are difficult to map efficiently to modern GPUs due to problem-specific iterations, triangular solves, and global synchronization. Learning-based methods offer a promising alternative, yet existing approaches are largely confined to inverse operators for linear system solving and often require CPU-side refinement or non-GPU-resident post-processing, breaking end-to-end GPU execution. We propose FlowSMO (Flow-based Sparse Matrix Operator), a size-generalizable GPU-resident generative framework for approximating sparse matrix operators across varying matrix dimensions. For a given operator class, FlowSMO first encodes the input sparse matrix A into a dimension-agnostic conditioning representation. Conditioned on this representation, a flow-matching model samples latent operator variables with a cost independent of the matrix size. To recover an operator at the original scale, a sparse decoder predicts a local sparse component over candidate edges induced by the input sparsity pattern, while a low-rank component captures global correlations and fill-in effects. The resulting operator is represented implicitly as $M = M_{\text{sparse}} + U\text{diag}(\sigma)U^\top$, enabling matrix-free application without explicitly assembling dense matrices. All stages, including generation and refinement, are executed on GPU, with computational cost scaling as $O(\text{nnz}(A)+Nr)$ for rank r. We validate FlowSMO across three complementary operator settings: Helmholtz inverse operators for preconditioner generation under moderate spectral difficulty, inverse-square-root operators on sparse SPD matrices, and Schur complements as block-structured operators beyond standard spectral matrix functions. Experiments show that FlowSMO improves approximation quality while enabling GPU-resident near-linear inference and cross-scale generalization.

1 INTRODUCTION

Structured algebraic operators associated with sparse matrices are fundamental computational primitives in modern scientific computing (Saad, 2003; Xu & Zikatanov, 2017), GPU-accelerated simulation (Liu et al., 2016), and AI-for-Science (Chen, 2025; Rudikov et al., 2024). Many discretized problems give rise to large sparse matrices A, while the actual computation often depends on operators derived from A, such as inverse operators A^{-1}, inverse-square-root operators $A^{-1/2}$, and block-structured Schur complements (Higham, 2008; Benzi & Golub, 1999). These operators support core tasks including linear system solving (Saad, 2003), preconditioning (Benzi et al., 1996; Xu & Zikatanov, 2017), Gaussian sampling (Rasmussen & Williams, 2006), spectral graph learning (Trifonov et al., 2024), and PDE simulation (Brenner & Scott, 2008). As problem sizes grow and such operators must be constructed or applied repeatedly across matrix instances, efficient and scalable approximation of sparse matrix operators becomes a central challenge in scientific machine learning and numerical computing.

GPUs are well suited to the parallel matrix–matrix and matrix–vector products used in repeated operator applications (Bell & Garland, 2009; NVIDIA, 2020). Modern sparse libraries provide optimized kernels, and solvers such as cuDSS (NVIDIA, n.d.) and AmgX (Naumov et al., 2015) support GPU-accelerated direct and iterative solution pipelines. The cost of constructing and applying

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.

an operator nevertheless depends on the underlying algorithm. Incomplete factorizations involve data-dependent fill-in and triangular dependencies (Häusner et al., 2024; Trifonov et al., 2026); Krylov methods require sequential iterations with global reductions (Saad, 2003); and polynomial or rational approximations rely on sequential recurrences, repeated sparse operator applications, or shifted linear solves (Higham, 2008; Benzi & Razouk, 2007). These computations can incur irregular memory access, synchronization, and problem-specific preprocessing (Bell & Garland, 2009; Liu et al., 2016). For repeated use, an effective representation should therefore support both end-to-end GPU construction and inexpensive application.

Learning-based methods offer a promising way to amortize algebraic operator construction across matrix instances (Li et al., 2023; Trifonov et al., 2024). Neural solvers and learned preconditioners exploit recurring structural patterns in sparse matrices (Häusner et al., 2025; Chen, 2025), while neural operator models demonstrate the potential of learning reusable solution operators across related problem instances (Rudikov et al., 2024; Kopaničáková & Karniadakis, 2025). Nevertheless, existing learning-based approaches for sparse algebraic computation remain largely specialized to linear system solving or preconditioner prediction (Häusner et al., 2024; Lan et al., 2024), effectively targeting A^{-1} rather than broader sparse matrix operators such as $A^{-1/2}$ or Schur complements. Methods that predict factorization factors retain triangular solves during application, while methods that use the network as a preconditioner require a forward pass at each iteration (Häusner et al., 2024; Chen, 2025). This motivates an end-to-end GPU framework that generates reusable operator approximations across matrix sizes and keeps their refinement and application on the device.

To address these limitations, we propose Flow-based Sparse Matrix Operator (FlowSMO), a size-generalizable, GPU-resident framework for approximating matrix operators induced by sparse matrices. Unlike many neural operators that predict solutions, corrections, or preconditioners for individual linear systems, FlowSMO constructs an implicit, reusable approximation to the operator itself. Given a sparse matrix A, FlowSMO encodes its structural and numerical information into a compact, dimension-agnostic latent representation. Conditioned on this representation, a flow-matching model (Lipman et al., 2023) samples operator-level latent variables, which are decoded into a sparse and low-rank operator representation. The sparse component models local graph interactions, while the low-rank component captures long-range correlations and fill-in effects (Demko et al., 1984; Benzi & Razouk, 2007) commonly induced by inverse, inverse-square-root, and Schur-complement operators. Matrix encoding, latent generation, decoding, refinement, and operator application all remain on the GPU. The implicit representation supports repeated matrix–vector products without dense assembly or triangular solves. Together, these components enable matrix-free GPU execution with computational cost scaling as $O(\mathrm{nnz}(A) + Nr)$ for rank r, while allowing the same model to generalize across different matrix sizes. In summary, we contribute:

1) Sparse matrix operator learning beyond linear solves. We study the problem of learning reusable approximations to matrix operators induced by sparse matrices, including inverse, inverse-square-root, and Schur-complement operators. This formulation goes beyond neural linear solvers that predict solutions or preconditioners for individual linear systems.

2) A size-generalizable flow-based operator generator. We propose FlowSMO, a GPU-resident generative framework that maps sparse matrices to operator-level latent variables through conditional flow matching. By performing generation in a compact dimension-agnostic latent space, FlowSMO supports generalization across matrix sizes.

3) An implicit sparse and low-rank representation. FlowSMO decodes latent variables into $M = M_{\mathrm{sparse}} + U\mathrm{diag}(\sigma)U^\top$, where the sparse component captures local graph interactions and the low-rank component models global correlations and fill-in effects. The operator is stored and applied in matrix-free form, avoiding explicit dense assembly.

4) GPU-resident inference and broad empirical validation. FlowSMO avoids CPU-side refinement during inference and achieves matrix-free GPU execution with cost $O(\mathrm{nnz}(A) + Nr)$ for rank r. We validate FlowSMO on Helmholtz inverse operators, inverse-square-root operators for sparse SPD matrices, and Schur complements, demonstrating strong approximation quality, near-linear inference, and cross-size generalization.

2 Related Work

Classical algebraic methods. Classical methods for sparse matrix operator approximation differ in their computational dependencies and target-specific constructions. First, efficient GPU execution must account for algorithm-specific dependencies: ILU/IC use sparse triangular solves (Saad, 2003; Liu et al., 2016), AMG coordinates work across levels (Xu & Zikatanov, 2017; Naumov et al., 2015), and classical SPAI construction solves local optimization problems for individual columns (Benzi et al., 1996; Kolotilina & Yeremin, 1993; Bridson & Tang, 1999). Second, common constructions depend on the target operator: incomplete factorizations for A^{-1}, rational Krylov iterations for matrix functions such as $A^{-1/2}$ (Saad, 2003), and block elimination for Schur complements.

Learning-based preconditioning. Recent work has applied neural networks to accelerate preconditioner construction. A series of methods use GNNs to predict incomplete factorization factors (Li et al. (2023), NeuralIF (Häusner et al., 2024; 2025), Trifonov et al. (2024)), which require triangular solves when applied, and the elimination-tree structure introduces long-range dependencies that conflict with GNNs' local message passing (Trifonov et al., 2026). Another line of work uses the neural network itself as the preconditioner (Lan et al. (2024), GNP (Chen, 2025), FCG-NO (Rudikov et al., 2024), Kopaničáková & Karniadakis (2025)), avoiding triangular solves but requiring a full network forward pass for each preconditioner application. The graph-conditioned variational autoencoder (GCVAE) proposed by Li et al. (2024) and the method of Yang et al. (2025) construct sparse approximate inverse preconditioners with purely sparse representations. These approaches primarily address inverse approximation for linear system solving.

Flow matching for structured generation. Flow matching (Lipman et al., 2023; Albergo & Vanden-Eijnden, 2023; Liu et al., 2023) has emerged as a highly efficient framework for training continuous normalizing flows, learning a velocity field that transports a tractable prior to a target data distribution via a deterministic probability flow ODE. Generative models have been applied to molecular graphs (Jo et al., 2022; Vignac et al., 2023) and point clouds (Zhou et al., 2021), while multi-physics pretraining extends learned surrogates across physical systems (McCabe et al., 2024). Conditional flow generation for large-scale algebraic operators remains comparatively unexplored. Existing generative models for relational or graphical data typically operate directly in the ambient space. The dimension of the evolving state therefore grows with the problem size N, increasing computation and memory costs when the models are applied to larger instances. FlowSMO formulates sparse matrix operator approximation as a conditional latent flow matching problem. By projecting the operator's spectral properties into a compact, fixed-dimensional latent space, FlowSMO fully decouples the ODE integration from the matrix dimension N, circumventing the dimensionality bottleneck and enabling zero-shot generalization across varying matrix scales.

3 FlowSMO: conditional generation of matrix operators

3.1 Problem setup and overview

Given a sparse matrix $A \in \mathbb{R}^{N\times N}$ and an operator f, we seek an approximation $M \approx f(A)$ that supports efficient matrix–vector products on GPU. We consider inverse, inverse-square-root, and Schur-complement operators. For each operator type, the learned parameters are shared across matrix sizes.

FlowSMO constructs this approximation by encoding the matrix, generating an operator code, and decoding it at the required size (Figure 1). The matrix encoder provides node features and a global condition; a conditional flow uses the latter to sample a fixed-dimensional code. The decoder combines the code with the node features to produce a sparse-plus-low-rank operator. We first initialize the model on small matrices using target-operator responses for supervision. We then freeze the encoders and jointly optimize the flow and decoder on larger matrices through operator residuals. This second stage, like inference, requires only A.

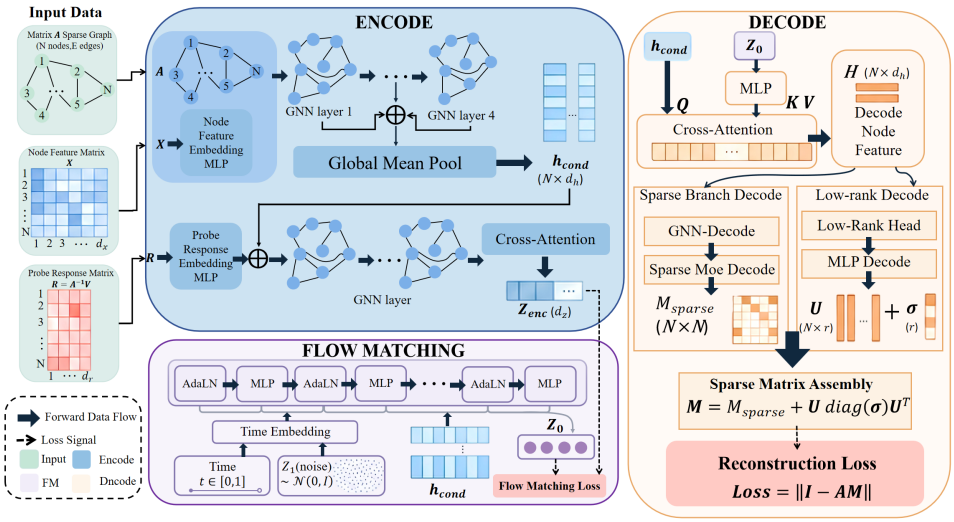


Figure 1: Overview of FlowSMO. The probe path supplies target latents during supervised initialization. Self-supervised training and inference generate latents from A alone, using the condition encoder, flow, and decoder.

3.2 Matrix Conditioning and Operator Encoding

Supervised initialization uses two complementary representations: a condition extracted from A and a latent code describing $f(A)$. The first guides generation; the second provides the target that the flow learns to generate.

Matrix condition. We represent A as a weighted graph, with one node per matrix index and one edge per nonzero entry. A GNN produces node features $h_{\text{cond}} \in \mathbb{R}^{N\times D}$ and a global summary $g \in \mathbb{R}^{d_g}$. Each layer aggregates neighboring features with learned edge gates:

$$m_i^{(l)} = \sum_{j\in\mathcal{N}(i)} \phi_m(h_j^{(l)}) \odot \sigma(\phi_e(A_{ij})), \qquad h_i^{(l+1)} = h_i^{(l)} + \phi_u(h_i^{(l)}, m_i^{(l)}), \tag{1}$$

where ϕ_m, ϕ_e, and ϕ_u are MLPs. Periodic mean pooling shares information across the graph. The node features retain local matrix structure, while g provides a condition whose dimension is independent of N.

Target-operator code. To encode $f(A)$, we evaluate its responses $R = f(A)V_{\text{in}}$ to $P \ll N$ input probes. Entries of $V_{\text{in}} \in \mathbb{R}^{N\times P}$ are drawn independently from $\mathcal{N}(0, 1/N)$ and held fixed, avoiding probe-resampling noise in the latent targets during initialization. The probe encoder combines R with h_{cond} and pools the features through K_{enc} learnable query tokens to obtain z. Thus the target code summarizes operator actions without taking a full $N \times N$ matrix as input. This path is used only for supervised initialization; separate, resampled probes V_{loss} evaluate generated-operator residuals during training (Appendix H).

3.3 Conditional Latent Generation

The flow learns to generate operator codes conditioned on the matrix summary g. During initialization, the encoders supply target pairs (z_0, g). We draw $z_1 \sim \mathcal{N}(0, I)$ and $t \sim \mathcal{U}[0, 1]$, form $z_t = (1-t)z_0 + tz_1$, and fit the velocity along this interpolation:

$$\mathcal{L}_{\text{flow}} = \mathbb{E}_{t,z_1}\left[\|v_\theta(z_t, t, g) - (z_1 - z_0)\|_2^2\right]. \tag{2}$$

We parameterize the velocity by an MLP with AdaLN conditioning on (t, g). Integrating backward from $t = 1$ to $t = 0$ transforms Gaussian noise into an operator code conditioned on A. The

dimensions of z and g do not depend on N, allowing the same flow to generate codes across matrix sizes.

At inference, we sample an operator code using 12 reverse Heun integration steps:

$$\hat{z}_0 = \mathrm{HeunODE}(v_\theta, z_1, g, 12;\, 1 \to 0). \tag{3}$$

Section 4.4 compares alternative step counts. After supervised initialization, training follows this sampling path rather than target codes: the flow generates a code, the decoder constructs an operator, and residual gradients update both jointly. Late in this stage, a power-iteration loss is added to the residual loss. Appendix H specifies the objectives and updates.

3.4 Sparse and low-rank decoding

Decoding restores the matrix dimension by combining the generated code with features of individual nodes. We project z into K_{dec} tokens and attend to them using h_{cond} as queries, producing $h_{\mathrm{dec}} \in \mathbb{R}^{N \times D}$. Each node representation therefore contains both local matrix information and information about the sampled operator.

The sparse branch predicts entries on the k-hop graph pattern $\mathcal{S}_k$. A FiLM-conditioned GNN refines the node representations, and each candidate edge is described by

$$\psi_{ij} = [h_i + h_j,\ h_i \odot h_j,\ e_{\mathrm{hop}}(i,j),\ A_{ij},\ g], \tag{4}$$

where e_{hop} encodes graph distance and $A_{ij} = 0$ for edges absent from the input pattern. A mixture-of-experts head maps ψ_{ij} to an entry of M_{sparse}.

The low-rank branch complements these local entries with nonlocal interactions. It projects z into r direction tokens, which attend to h_{dec} to form $U \in \mathbb{R}^{N \times r}$. An MLP maps (g, z) to weights $\sigma \in \mathbb{R}^r$, giving the combined operator

$$M = M_{\mathrm{sparse}} + U\,\mathrm{diag}(\sigma)U^\top. \tag{5}$$

The operator is applied as $Mx = M_{\mathrm{sparse}}x + U\,\mathrm{diag}(\sigma)(U^\top x)$, so no dense M is assembled. For fixed network widths, decoding and application cost $O(|\mathcal{S}_k| + Nr)$. This is linear in N for a graph of bounded degree and fixed k and r (Appendix B).

Approximation properties. The two branches also complement each other in approximation error: a sparse correction can capture part of the residual left by a low-rank approximation. The following construction quantifies this gain for inverse powers of SPD matrices. Let $\mathcal{S}_k$ include the diagonal, and let $[X]_{\mathcal{S}_k}$ retain the entries on this pattern and set all others to zero.

Theorem 1 (Sparse correction of a spectral truncation)**.** *Let A be SPD with orthonormal eigenpairs (λ_m, v_m) ordered by $0 < \lambda_1 \le \cdots \le \lambda_N$, and let $f(\lambda) = \lambda^{-\alpha}$, $\alpha > 0$. For $0 \le r < N$, define $L_r = \sum_{m=1}^{r} \lambda_m^{-\alpha} v_m v_m^\top$ and $H_r = f(A) - L_r$. Then the sparse-plus-low-rank approximation $M^\star = L_r + [H_r]_{\mathcal{S}_k}$ satisfies*

$$\|f(A) - M^\star\|_F^2 = \sum_{m>r} \lambda_m^{-2\alpha} - \|[H_r]_{\mathcal{S}_k}\|_F^2. \tag{6}$$

For the fixed L_r, $[H_r]_{\mathcal{S}_k}$ is the unique correction supported on $\mathcal{S}_k$ minimizing the Frobenius error.

The construction guarantees a squared Frobenius error equal to the spectral tail minus the residual energy recovered by the sparse correction. It therefore bounds the best error attainable in the combined representation. Because the pattern includes the diagonal, the correction strictly improves on spectral truncation alone. The proof and extensions to polynomial corrections and symmetric indefinite inverses are given in Appendix F.

Empirical validation. The combined representation attains the lowest Frobenius error across four matrix families (Table 9, Appendix E.1), consistent with the gain from combining the two components. Residual-based training learns better preconditioners within this class: on Helmholtz at $N = 4096$, the trained model achieves $\rho(I - AM) = 0.836$, compared with 5.54 for the Frobenius approximation. Removing the low-rank or sparse branch raises ρ to 5.81 or 1.0000, respectively (Table 10). Thus the representation provides the approximation guarantee, while training exploits it for preconditioning. Beyond these component comparisons, a joint Poisson–Helmholtz model retains convergence at coefficient contrast 660 and adapts to unseen anisotropy ratios after one epoch of fine-tuning (Appendix E.2).

4 EXPERIMENTS

We evaluate whether FlowSMO produces accurate operators at larger matrix sizes and whether their construction cost translates into faster linear solves. Inverse-square-root and Schur-complement tasks test spectral and block-structured operators; Helmholtz systems test preconditioning performance. Each operator type uses a separate model with the same architecture and two-stage training procedure. All experiments run on one NVIDIA RTX 5090 GPU (32 GB). Model settings and training objectives appear in Appendices A and H. Unless an integration-step ablation explicitly specifies otherwise, all FlowSMO quality and timing results use 12 Heun steps. Tables 1–4 use one draw per matrix, without candidate selection.

4.1 INVERSE-SQUARE-ROOT APPROXIMATION

Setup. For SPD A, an approximation $M \approx A^{-1/2}$ gives a symmetric transformation $MAM \approx I$. This experiment tests a spectral matrix function beyond inversion and transfer to substantially larger matrices. We generate diagonally dominant sparse SPD matrices with average degree about 15 and train on $N \in \{1\text{k}, 3\text{k}, 5\text{k}, 7\text{k}, 10\text{k}\}$, with 200–300 samples per size. The low-rank head is disabled to evaluate the sparse decoder. We measure $\rho(I - MAM)$ and compare total time with exact eigendecomposition on GPU and CPU; implementations and training details are in Appendix D.

Table 1: Inverse-square-root approximation across matrix sizes. FlowSMO uses 12 flow steps and reports total time; exact eigendecomposition provides reference timings. OOM denotes insufficient memory.

N	FlowSMO Time (ms)	$\rho(I-MAM)$	Exact GPU	Exact CPU
1,000	31	0.219	7 ms	123 ms
5,000	175	0.215	151 ms	4.2 s
10,000	359	0.279	1.1 s	25.1 s
20,000	802	0.284	8.8 s	169 s
50,000	1790	0.221	OOM	OOM
100,000	3591	0.278	OOM	OOM
500,000	11531	0.378	OOM	OOM
1,000,000	23084	0.424	OOM	OOM

Results. A model trained only up to $N = 10{,}000$ transfers to matrices $100\times$ larger, with reported $\rho(I - MAM) = 0.424$ at $N = 1$M and below one at every tested size (Table 1). The sparse representation makes this transfer practical: it supports operator application without storing a dense inverse square root. FlowSMO takes 23.1 s at $N = 1$M, whereas exact eigendecomposition runs out of memory on both GPU and CPU from $N = 50{,}000$. These results demonstrate that the learned spectral approximation remains useful well beyond the training dimensions.

4.2 SCHUR-COMPLEMENT APPROXIMATION

Setup. The Schur complement $S = A_{SS} - A_{SI}A_{II}^{-1}A_{IS}$ depends on a partition into interior (I) and separator (S) nodes. It therefore tests whether the same framework can learn a block-structured operator. We construct random geometric graph Laplacians on $[0, 1]^2$ with average degree about 10, partition them by Fiedler-vector bisection, and provide the partition as binary node labels. Training uses 11 sizes, $N \in \{100, \ldots, 4400\}$, with 150–500 samples per size. Sparse LU provides small-matrix targets for initialization.

We disable the low-rank head and predict Schur entries on the two-hop pattern. Quality is measured on this pattern by $\|M_{\mathcal{S}_k} - S_{\mathcal{S}_k}\|_F / \|S_{\mathcal{S}_k}\|_F$, with $k = 2$. Table 2 includes GPU Neumann approximations with 1, 2, 5, and 10 terms, and sparse direct construction with cuDSS. Appendix D specifies the test matrices and timing scope. All methods are evaluated on the original test matrices with the same graph partitions.

Results. FlowSMO extends to block-structured operators with stable accuracy beyond its training sizes (Table 2). At $N = 32{,}000$, $7.3\times$ the largest training dimension, it achieves relative error 0.057 on the two-hop pattern in 25.1 ms. Construction is $7.8\times$ faster than near-exact cuDSS construction (196.59 ms). Neumann expansions also perform well on this graph family: five and ten terms are both faster and more accurate than FlowSMO. The same architecture thus extends from spectral matrix functions to partition-dependent block operators, retaining accuracy at sizes well beyond training.

Table 2: Schur-complement approximation. Mean relative Frobenius error on the two-hop pattern and GPU construction time over ten matrices per size.

N	FlowSMO		Neumann-1		Neumann-2		Neumann-5		Neumann-10		cuDSS	
	Error	ms	Error	ms	Error	ms	Error	ms	Error	ms	Error	ms
500	0.155	2.65	0.0835	0.40	0.0574	0.42	0.0249	0.50	0.0106	0.64	$< 10^{-12}$	11.19
2,000	0.082	2.69	0.0879	0.71	0.0629	0.65	0.0295	0.74	0.0128	0.89	$< 10^{-12}$	20.85
8,000	0.057	6.12	0.0786	2.57	0.0552	1.65	0.0253	1.83	0.0109	2.13	$< 10^{-12}$	56.90
16,000	0.066	12.6	0.0656	5.50	0.0442	3.67	0.0177	3.85	0.0061	4.42	$< 10^{-12}$	100.55
32,000	0.057	25.1	0.0836	12.67	0.0589	12.65	0.0271	13.41	0.0116	15.31	$< 10^{-12}$	196.59

4.3 Helmholtz preconditioning

Setup. We now evaluate whether the generated inverse operators reduce the cost of linear solves. The training matrices discretize $-\nabla \cdot (\kappa(x)\nabla u) + \sigma(x)u = f$ on $n \times n$ grids with a five-point stencil, giving $N = n^2$. Diffusion coefficients are smooth fields built from 3–10 Fourier modes with $\kappa_0 \in [0.5, 3.0]$. Reaction coefficients are positive in 60% of samples and mildly negative in 40%. Training uses 10 sizes, $N \in \{100, \ldots, 4096\}$, with 300–500 samples per size. Testing uses ten smooth patches per grid size from the Marmousi-II velocity model (Martin et al., 2006), mapped to $\kappa \in [0.5, 3.0]$ (Appendix C).

After decoding, three Neumann–Schulz steps refine the operator on GPU. We compare with Jacobi, ILU, GPU and CPU dense inversion, and two learned preconditioners: SPAI-GNN (Yang et al., 2025) and NeuralIF (Häusner et al., 2024), retrained on the same data. Iterative methods use GMRES(10), relative residual tolerance 10^{-8}, and a 500-iteration cap. For this Marmousi-II comparison, preconditioner application and Krylov iterations run on GPU. We report $\rho(I - AM)$, GMRES iterations, construction time, and total solve time. NeuralIF construction takes about 3 ms and is not listed separately in Table 3. Appendix D specifies the implementations and refinement.

Results. FlowSMO achieves the fewest GMRES iterations and the lowest total solve time among the learned methods at every tested size (Table 3). Its more expensive construction is offset by faster convergence. At $N = 8100$, for example, FlowSMO requires 7 iterations and 53 ms, compared with 104 iterations and 305 ms for SPAI-GNN, and 230 iterations and 1342 ms for NeuralIF. Across all sizes, FlowSMO is 1.6–$7.7\times$ faster than SPAI-GNN in mean total time. Even at $N = 22{,}500$, more than five times the largest training dimension, it solves the systems in 32 iterations and about 151 ms.

The iteration counts also show why spectral radius alone is insufficient to assess GMRES performance: at larger sizes, FlowSMO converges faster even when NeuralIF has a smaller ρ. Whereas $\rho(I - AM)$ governs stationary iteration, restarted GMRES depends on more than this scalar summary. NeuralIF applies its predicted factors through triangular solves, whose application cost in our benchmark exceeds that of FlowSMO's sparse matrix–vector products and low-rank operations. Together with the larger iteration counts, this leads to total solve times up to $77\times$ longer.

The reduction in iterative work also yields gains over Jacobi and ILU as the Marmousi-II systems grow (Table 4). At $N = 22{,}500$, FlowSMO reduces ILU's 173 iterations to 32 and total time from 635.3 to 151.3 ms, a $4.2\times$ speedup. Jacobi does not reach tolerance within 500 iterations for $N \geq 8100$. AMG-SA and cuDSS remain faster at every tested size. FlowSMO's advantage on this benchmark is therefore a stronger learned preconditioner that reduces solve time relative to the learned baselines and, on larger systems, Jacobi and ILU.

Table 3: Helmholtz solves with learned preconditioners. Mean ± standard deviation over ten Marmousi-II matrices per size. Model and Total denote construction and total solve time in milliseconds.

N	FlowSMO				SPAI-GNN				NeuralIF		
	ρ	Model (ms)	Total (ms)	it.	ρ	Model (ms)	Total (ms)	it.	ρ	Total (ms)	it.
100	.002±.000	12±0	12±0	2±0	.883±.024	1.2±0.4	19±2	15±1	.507±.015	45±41	21±1
400	.016±.001	13±1	14±1	2±0	.959±.014	1.0±0.0	56±10	27±3	.711±.016	110±13	42±3
1,024	.036±.000	14±0	15±0	3±0	.964±.019	1.0±0.0	103±36	38±8	.816±.031	197±38	71±14
1,600	.076±.003	16±1	18±1	3±0	.966±.023	1.0±0.0	134±56	47±14	.854±.041	280±68	94±23
4,096	.541±.124	19±1	29±3	7±2	.969±.030	0.9±0.0	223±133	80±45	.867±.058	495±286	126±70
6,400	.531±.009	25±2	42±3	6±0	.970±.018	0.9±0.0	198±141	67±47	.904±.050	935±496	185±95
8,100	.542±.014	28±1	53±2	7±1	.972±.019	1.0±0.0	305±218	104±71	.921±.052	1342±612	230±102
12,100	.947±.119	43±2	100±12	18±8	.978±.014	1.3±0.2	411±278	137±90	.866±.091	3310±2816	190±162
16,384	1.24±.18	60±5	117±8	29±2	.964±.033	1.6±0.2	457±396	152±125	.938±.053	9016±5254	392±228
22,500	1.56±.13	77±3	151±11	32±1	.966±.025	2.0±0.0	336±406	110±123	.858±.057	2185±2573	145±170

Table 4: Helmholtz solves with classical baselines. Mean ± standard deviation over ten Marmousi-II matrices per size. Times are in milliseconds unless marked otherwise; it. denotes GMRES iterations.

N	FlowSMO		Jacobi		ILU		Direct GPU	Direct CPU	AMG-SA CPU	cuDSS GPU
	ms	it.	ms	it.	ms	it.	ms	ms	ms	ms
100	12.1±0.3	2±0	2.3±0.1	29±2	0.5±0.0	2±0	0.3±0.1	1.2±0.0	7.7±0.1	5.8±0.1
400	14.0±0.6	2±0	5.7±0.6	68±8	1.2±0.0	2±0	1.0±0.0	15.3±0.1	9.1±1.1	7.3±0.0
1,024	15.4±0.2	3±0	10.8±0.9	117±11	3.1±0.0	2±0	2.5±0.0	62.0±0.5	13.9±1.0	9.7±0.0
1,600	17.6±1.4	3±0	16.6±1.1	172±12	6.6±0.2	12±1	4.3±0.0	107±2	17.4±0.5	13.0±0.6
4,096	28.7±2.6	7±2	44.6±3.4	344±25	31.8±1.5	42±3	14.9±0.0	737±37	27.5±0.7	24.6±8.5
6,400	41.7±2.5	6±0	74.6±6.6	465±37	56.1±2.9	52±4	33.4±0.0	1.7±0.0 s	40.0±1.6	30.6±0.1
8,100	52.9±2.3	7±1	93.9±2.4	>500	92.4±9.3	78±9	53.9±0.1	2.4±0.0 s	48.1±1.9	37.0±0.3
12,100	99.9±12.0	18±8	708.6±40.3	>500	307.8±34	113±14	132±0	5.9±0.1 s	62.6±1.5	52.2±3.3
16,384	117.1±7.7	29±2	601.0±60.1	>500	404.4±51	133±19	250±0	16.9±0.5 s	81.2±3.3	70.5±18.7
22,500	151.3±11	32±1	801.7±37.1	>500	635.3±46	173±7	646±21	28.9±4.0 s	111.3±4.7	84.3±3.1

Iterative timings include setup and solve. Direct GPU/CPU report dense inversion; cuDSS reports analysis, factorization, and solve, excluding host–device transfers.

4.4 Regression and conditional flow

For a fixed A, structurally different operators can all approximate A^{-1} well. Averaging them need not be harmful: the residual norm is convex in M. If $\|I - AM_i\|_2 \le \varepsilon < 1$ for $i = 1, 2$, every convex combination $\bar{M} = \tau M_1 + (1-\tau)M_2$, $\tau \in [0, 1]$, satisfies $\rho(I - A\bar{M}) \le \|I - A\bar{M}\|_2 \le \varepsilon$. Thus averaging operators preserves this contraction guarantee.

FlowSMO predicts a low-dimensional latent code z, however, and nonlinear decoding does not preserve this convexity. Under squared-error latent supervision, deterministic regression targets the conditional mean of the codes. When valid codes occupy separated modes, their mean can lie between them and decode to a poor operator. Conditional flow models the distribution over these codes, allowing samples to reach distinct valid regions. Generation takes place in a fixed-dimensional latent space, so its cost is independent of matrix size. We compare the two heads after the full two-stage training, including residual-based joint optimization, and examine how many integration steps are needed.

Generation and selection. On 120 Helmholtz matrices ($N = 4096$), we keep the encoder–decoder architecture, data, and two-stage training the same, including joint residual training of each head and its decoder without latent labels. Regression yields mean $\rho = 0.518$ and $\rho > 1$ on 22 of 120 matrices (Table 5). Flow lowers the mean to 0.452; only 11 of 1,920 draws have $\rho > 1$, all on three matrices. Selecting one of 16 draws further lowers the mean to 0.402, with no selected output above one. Each draw shares the matrix condition but is sampled and decoded separately; selection uses power-iteration estimates of $\rho(I - AM)$.

Table 5: Latent generation and selection.

Method	Mean ρ	$\rho > 1$	
		Outputs	Matrices
Regression	0.518	22/120	22/120
Flow	0.452	11/1920	3/120
Flow + selection	**0.402**	**0/120**	**0/120**

Selection retains one of 16 draws per matrix.

Table 6: ODE steps (1,920 draws).

Steps	Mean ρ	Draws with $\rho \geq 1$
1	0.541	260 (13.5%)
2	0.484	80 (4.2%)
4	0.455	15 (0.8%)
8	0.452	11 (0.6%)
25	0.452	11 (0.6%)

Integration steps. Increasing the integration budget improves both mean quality and the fraction of contractive outputs. On 120 Helmholtz matrices with 16 draws each, moving from one to eight steps lowers mean ρ from 0.541 to 0.452 and the fraction with $\rho \geq 1$ from 13.5% to 0.6% (Table 6). Increasing to 25 steps gives no further improvement at the reported precision, suggesting that eight steps capture most of the benefit in this experiment.

Figure 2 illustrates the mechanism: regression lies between reference solutions, while flow samples reach separate low-ρ regions. Appendix G explains the projection and follows operator quality along a trajectory.

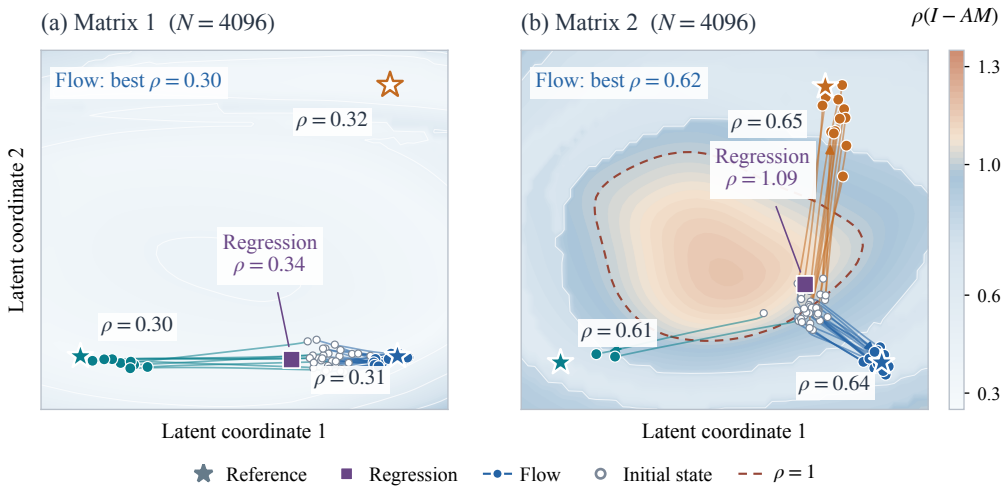


Figure 2: Regression and flow in latent space ($N = 4096$). Shading estimates $\rho(I - AM)$; 32 trajectories use 12 steps. Annotations evaluate the original latent codes. Appendix G.1 describes the projection.

5 Conclusion and Limitations

FlowSMO learns to construct implicit matrix operators from sparse inputs. Conditional flow generates a fixed-dimensional operator code, and sparse-plus-low-rank decoding recovers local and nonlocal interactions at the required matrix size. The same architecture supports inverse, inverse-square-root, and Schur-complement targets. Experiments show transfer to larger matrices and reduced Helmholtz solving time, while ablations establish the benefit of retaining multiple latent solutions and using several integration steps. The current models are trained separately for each operator type, lose spectral accuracy on highly ill-conditioned inputs, and are evaluated mainly on structured or semi-structured meshes. Broader evaluation on unstructured, non-SPD, and three-dimensional problems, including SuiteSparse systems, is a natural next step.

AI USE STATEMENT

AI assistance in preparing this manuscript included editorial revision, integration of previously obtained experimental results, plotting code for figures from saved numerical data, and bibliographic searches and metadata checks.

REFERENCES

M. S. Albergo and E. Vanden-Eijnden. Building normalizing flows with stochastic interpolants. In *ICLR*, 2023. URL https://openreview.net/forum?id=li7qeBbCR1t.

N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pp. 1–11, 2009. doi: 10.1145/1654059.1654078.

M. Benzi and G. H. Golub. Bounds for the entries of matrix functions with applications to preconditioning. *BIT Numerical Mathematics*, 39(3):417–438, 1999. doi: 10.1023/A:1022362401426.

M. Benzi and N. Razouk. Decay bounds and $O(n)$ algorithms for approximating functions of sparse matrices. *Electronic Transactions on Numerical Analysis*, 28:16–39, 2007. URL https://etna.ricam.oeaw.ac.at/vol.28.2007-2008/pp16-39.dir/pp16-39.pdf.

Michele Benzi, Carl D. Meyer, and Miroslav Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17(5):1135–1149, 1996. doi: 10.1137/S1064827594271421.

S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*. Springer, 3rd edition, 2008. doi: 10.1007/978-0-387-75934-0.

R. Bridson and W.-P. Tang. Ordering, anisotropy, and factored sparse approximate inverses. *SIAM Journal on Scientific Computing*, 21(3):867–882, 1999. doi: 10.1137/S1064827598335842.

J. Chen. Graph neural preconditioners for iterative solutions of sparse linear systems. In *ICLR*, 2025. URL https://openreview.net/forum?id=Tkkrm3pA35.

S. Demko, W. F. Moss, and P. W. Smith. Decay rates for inverses of band matrices. *Mathematics of Computation*, 43(168):491–499, 1984. doi: 10.1090/S0025-5718-1984-0758197-9.

P. Häusner, O. Öktem, and J. Sjölund. Neural incomplete factorization: learning preconditioners for the conjugate gradient method. *Transactions on Machine Learning Research*, 2024. URL https://openreview.net/forum?id=FozLrZ3CI5.

Paul Häusner, Aleix Nieto Juscafresa, and Jens Sjölund. Learning incomplete factorization preconditioners for GMRES. In *Proceedings of the Northern Lights Deep Learning Conference*, volume 265 of *Proceedings of Machine Learning Research*, pp. 85–99, 2025. URL https://proceedings.mlr.press/v265/hausner25a.html.

N. J. Higham. *Functions of Matrices: Theory and Computation*. SIAM, 2008. doi: 10.1137/1.9780898717778.

J. Jo, S. Lee, and S. J. Hwang. Score-based generative modeling of graphs via the system of stochastic differential equations. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 10362–10383, 2022. URL https://proceedings.mlr.press/v162/jo22a.html.

L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditionings I: Theory. *SIAM Journal on Matrix Analysis and Applications*, 14(1):45–58, 1993. doi: 10.1137/0614004.

A. Kopaničáková and G. E. Karniadakis. DeepONet based preconditioning strategies for solving parametric linear systems of equations. *SIAM Journal on Scientific Computing*, 47(1):C151–C181, 2025. doi: 10.1137/24M162861X.

K. W. Lan, E. Gueidon, A. Kaneda, J. Panetta, and J. Teran. A neural-preconditioned Poisson solver for mixed Dirichlet and Neumann boundary conditions. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 25976–25994, 2024. URL https://proceedings.mlr.press/v235/lan24a.html.

Mou Li, He Wang, and Peter K. Jimack. Generative modeling of sparse approximate inverse preconditioners. In *Computational Science – ICCS 2024*, pp. 378–392. Springer, 2024. doi: 10.1007/978-3-031-63759-9_40.

Y. Li, P. Y. Chen, T. Du, and W. Matusik. Learning preconditioners for conjugate gradient PDE solvers. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19425–19439, 2023. URL https://proceedings.mlr.press/v202/li23e.html.

Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, and M. Le. Flow matching for generative modeling. In *ICLR*, 2023. URL https://openreview.net/forum?id=PqvMRDCJT9t.

W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par 2016: Parallel Processing*, pp. 617–630. Springer, 2016. doi: 10.1007/978-3-319-43659-3_45.

X. Liu, C. Gong, and Q. Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *ICLR*, 2023. URL https://arxiv.org/abs/2209.03003.

G. S. Martin, R. Wiley, and K. J. Marfurt. Marmousi2: An elastic upgrade for Marmousi. *The Leading Edge*, 25 (2):156–166, 2006. doi: 10.1190/1.2172306.

Michael McCabe, Bruno Régaldo-Saint Blancard, Liam Parker, Ruben Ohana, Miles Cranmer, Alberto Bietti, Michael Eickenberg, Siavash Golkar, Geraud Krawezik, Francois Lanusse, Mariel Pettee, Tiberiu Tesileanu, Kyunghyun Cho, and Shirley Ho. Multiple physics pretraining for spatiotemporal surrogate models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 119301–119335, 2024. doi: 10.52202/079017-3791.

Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolay Markovskiy, Istvan Reguly, Nikolai Sakharnykh, Vijay Sellappan, and Robert Strzodka. AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 37(5):S602–S626, 2015. doi: 10.1137/140980260.

NVIDIA. *cuSPARSE Library*. NVIDIA, 2020. URL https://docs.nvidia.com/cuda/archive/11.0/cusparse/index.html. CUDA Toolkit 11.0 documentation.

NVIDIA. *NVIDIA cuDSS Documentation*. NVIDIA, n.d. URL https://docs.nvidia.com/cuda/cudss/. Version 0.8.0. Accessed September 25, 2026.

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. doi: 10.7551/mitpress/3206.001.0001.

A. Rudikov, V. Fanaskov, E. Muravleva, Y. M. Laevsky, and I. Oseledets. Neural operators meet conjugate gradients: The FCG-NO method for efficient PDE solving. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 42766–42782, 2024. URL https://proceedings.mlr.press/v235/rudikov24a.html.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2nd edition, 2003. doi: 10.1137/1.9780898718003.

V. Trifonov, E. Muravleva, and I. Oseledets. Message-passing GNNs fail to approximate sparse triangular factorizations. *Transactions on Machine Learning Research*, 2026. URL https://arxiv.org/abs/2502.01397.

Vladislav Trifonov, Alexander Rudikov, Oleg Iliev, Yuri M. Laevsky, Ivan Oseledets, and Ekaterina Muravleva. Learning from linear algebra: A graph neural network approach to preconditioner design for conjugate gradient solvers. *arXiv preprint arXiv:2405.15557*, 2024. doi: 10.48550/arXiv.2405.15557.

C. Vignac, I. Krawczuk, A. Siraudin, B. Wang, V. Cevher, and P. Frossard. DiGress: Discrete denoising diffusion for graph generation. In *ICLR*, 2023. URL https://openreview.net/forum?id=UaAD-Nu86WX.

J. Xu and L. Zikatanov. Algebraic multigrid methods. *Acta Numerica*, 26:591–721, 2017. doi: 10.1017/S0962492917000083.

Z. Yang, Z. Li, K. Lyu, Y. Li, T. Du, and L. Liu. Learning sparse approximate inverse preconditioners for conjugate gradient solvers on GPUs. In *NeurIPS*, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/23fcc63005ac1a6e460ec4e209d17607-Abstract-Conference.html.

L. Zhou, Y. Du, and J. Wu. 3D shape generation and completion through point-voxel diffusion. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 5826–5835, 2021. doi: 10.1109/ICCV48922.2021.00577.

A Architecture details

A.1 Network modules

Condition encoder. The encoder maps matrix entries to node features and a global condition. Initial node features contain A_{ii}, row sums, and local degree statistics. In each of the L_c graph blocks, an MLP forms messages from source and destination features, a sigmoid edge gate weights them, and their sum drives a residual node update followed by layer normalization. Periodic global layers return the mean-pooled representation to the nodes. A final mean pool and linear projection produce $g \in \mathbb{R}^{d_g}$.

Probe encoder. Target responses are represented by their per-node mean, standard deviation, minimum, and maximum, together with correlations across neighboring nodes. The probe encoder combines these statistics with h_{cond} through condition gates and graph layers, then pools the node features into $z \in \mathbb{R}^{d_z}$ using cross-attention with learnable queries. The encoder uses $K_{\mathrm{enc}} = 48$ query tokens; task-specific graph depths appear in Table 7.

Decoder. The sparse head combines endpoint sums and products, hop embeddings, matrix entries, and the global condition. Four experts predict edge weights, which are combined by learned gates. A learned baseline for each hop distance is added as a residual term; self-loop baselines are initialized to one. For Helmholtz operators, analytical Fourier modes of the discrete Laplacian initialize the low-rank directions U. A graph-Laplacian eigenbasis provides an analogous initialization for general graphs.

Initialization and joint training. Supervised initialization trains the operator encoding and reconstruction paths, while flow matching learns the resulting latent targets. Perturbations $z \leftarrow z + \epsilon\xi$, with ϵ annealed from 0.15 to zero, regularize the latent space. The second stage freezes the encoders and jointly trains the flow and decoder on larger matrices through operator losses. Appendix H gives the two objectives and their gradient paths.

A.2 Hyperparameters

Table 7 gives the task-specific hyperparameters. All tasks use latent dimension $d_z = 256$, $K_{\mathrm{dec}} = 16$ decoder tokens, four experts in the sparse prediction head, an eight-layer flow MLP, and AdamW with learning rate 3×10^{-4}. Flow sampling uses 12 Heun steps; task-specific post-processing follows decoding.

Table 7: Model hyperparameters for the three operator tasks.

	A^{-1}	S	$A^{-1/2}$
Hidden width D	256	256	192
Global dimension d_g	256	256	192
Condition GNN blocks	6	4	4
Probe encoder blocks	3	3	2
Decoder GNN layers	4	4	1
Probe columns P	64	64	32
Flow MLP width	640	640	512
Distill MLP width	960	960	768
Post-processing	NS ($\times 3$)	—	DB ($\times 3$)
Low-rank head	yes	no	no

B Inference complexity breakdown

Table 8 separates the cost of matrix processing from latent integration. Here D is the graph feature width, E and E_k count edges in A and $\mathcal{S}_k$, S is the ODE step count, and d_h is the flow MLP width. For bounded graph degree and fixed network dimensions, k, and r, encoding, decoding, and

application scale linearly in N. Flow sampling acts only on $z \in \mathbb{R}^{d_z}$ and has no direct dependence on N.

Table 8: Computational cost of FlowSMO inference.

Stage	Cost	Depends on N?
GNN encoding	$O(L_c \cdot E \cdot D^2)$	Yes (linear)
Flow sampling	$O(S \cdot d_z \cdot d_h)$	No (d_z, d_h only)
Cross-attention	$O(N \cdot K_{\text{dec}} \cdot D)$	Yes (linear)
Edge prediction	$O(E_k \cdot D^2)$	Yes (linear)
Low-rank head	$O(N \cdot r \cdot D)$	Yes (linear)
Application of M	$O(\text{nnz}(M_{\text{sparse}}) + Nr)$	Yes (linear)

C MARMOUSI-II TEST DATA CONSTRUCTION

Data source. We use the P-wave velocity field from the Marmousi-II elastic model (Martin et al., 2006), downloaded from the SEG Open Data repository.[1] The velocity field contains $13{,}601 \times 2{,}801$ grid points with spacing 1.25 m, covering a 17.0 km $\times$ 3.5 km region. The P-wave velocity range is 1028–4700 m/s.

Patch selection. For each grid size n, we extract ten square $n \times n$ patches from distinct locations below the top 500 m water layer. We retain patches satisfying $v_{p,\max}/v_{p,\min} < 10.0$, selecting relatively smooth regions for comparison with the smooth Fourier fields used in training.

Coefficient mapping. Each extracted P-wave velocity patch $v_p(x)$ is linearly mapped to a diffusion coefficient by

$$\kappa(x) = 0.5 + 2.5\,\frac{v_p(x) - v_{p,\min}}{v_{p,\max} - v_{p,\min}}, \tag{7}$$

where $v_{p,\min} = 1028$ m/s and $v_{p,\max} = 4700$ m/s. This maps the real velocity field to $\kappa(x) \in [0.5, 3.0]$, matching the coefficient range used in training. The reaction coefficient $\sigma(x)$ is spatially constant: it is positive with probability 0.6, and otherwise mildly negative with $|\sigma| \le 0.2\lambda_1$.

Discretization. Each patch is discretized with the same five-point finite-difference stencil used for the synthetic Helmholtz data, producing an $N = n^2$ variable-coefficient Helmholtz matrix with the same sparsity pattern as the training matrices.

D EXPERIMENT DETAILS

Inverse square root. We generate sparse SPD matrices from random symmetric patterns with average degree about 15 and enforce diagonal dominance. Eigendecomposition supplies target operators for supervised initialization. Training uses 200 autoencoder epochs and 150 flow epochs. We evaluate the residual $I - MAM$; a spectral radius below one gives a contracting stationary iteration. Exact GPU and CPU references use `torch.linalg.eigh` (cuSOLVER) and `numpy.linalg.eigh` (LAPACK DSYEVD).

Schur complement. Random geometric graphs on $[0, 1]^2$, with average degree about 10, define $A_{\text{full}} = (D - W) + \mu I$, where $\mu \sim \mathcal{U}(0.1, 2.0)$. Fiedler-vector bisection assigns interior and separator nodes, and binary labels provide this partition to the model. Training covers 11 sizes, $N \in \{100, \ldots, 4400\}$, with 150–500 samples each. Sparse LU supplies initialization targets; training uses 200 autoencoder epochs and 150 flow epochs. All methods use the original test matrices and graph partitions. Neumann-q uses the first q terms of the Jacobi–Neumann expansion of A_{II}^{-1}. cuDSS solves $A_{II}X = A_{IS}$, followed by $S = A_{SS} - A_{SI}X$ on GPU. Both use FP64 on the RTX 5090 and are checked against sparse LU. Timings include method-specific preparation, allocation,

[1] `https://s3.amazonaws.com/open.source.geoscience/open_data/elastic-marmousi/elastic-marmousi-model.tar.gz`

host–device transfers, and synchronized Schur construction; graph generation, partitioning, shared block extraction, and reference evaluation are excluded. Each sample uses the median of three timings.

Helmholtz inverse. We discretize $-\nabla \cdot (\kappa(x)\nabla u) + \sigma(x)u = f$ on $n \times n$ grids using a five-point stencil, with $N = n^2$. The diffusion field combines 3–10 Fourier modes, with $\kappa_0 \in [0.5, 3.0]$ and amplitude at most $0.5\kappa_0$. The reaction coefficient is positive in 60% of samples and mildly negative, with $|\sigma| \leq 0.3\lambda_1$, in the remaining 40%. The low-rank branch is enabled. Training mixes ten sizes, $N \in \{100, \ldots, 4096\}$, with 300–500 samples per size. Training uses 350 autoencoder epochs and 300 flow epochs, learning rate 3×10^{-4}, and an eight-layer AdaLN flow MLP of width 640.

Three Neumann–Schulz steps refine the decoded inverse through $M_{k+1} = M_k(2I - AM_k)$. The residual satisfies $I - AM_{k+1} = (I - AM_k)^2$, so $\rho(I - AM_k) = \rho(I - AM_0)^{2^k}$. Refinement and operator application run on GPU. GMRES uses restart length 10, relative residual tolerance 10^{-8}, and a cap of 500 total inner iterations.

Jacobi and ILU use the same GMRES settings; ILU construction uses `scipy.sparse.linalg.spilu`. Direct inverses use GPU `torch.inverse` and CPU `numpy.linalg.inv`. SPAI-GNN (Yang et al., 2025) and NeuralIF (Häusner et al., 2024) are retrained on the same data. cuDSS (NVIDIA, n.d.) uses symmetric factorization in FP64 on the RTX 5090. Its timings sum synchronized analysis, factorization, and solve, excluding data transfers and resource allocation.

E Component ablations and distribution shift

We first examine how the sparse and low-rank components contribute to approximation accuracy and learned preconditioning, then test transfer beyond the training distribution.

E.1 Sparse and low-rank components

Theorem 1 predicts that a sparse correction improves on spectral truncation. Table 9 examines this benefit across four matrix families through numerical minimization of the Frobenius error with a low-rank component, a sparse component, or both. Each family contains 50 matrices with $N = 4096$, using rank $r = 16$ and a two-hop sparsity pattern. We report relative error $\varepsilon_F = \|f(A) - M\|_F / \|f(A)\|_F$ and spectral radius: $\rho(I - AM)$ for inverse operators and $\rho(I - MAM)$ for inverse square roots.

The trained comparison uses the full model, a sparse-only variant, and a low-rank-only variant on the same Helmholtz data for 100 epochs, changing only the output branches. Table 10 reports their spectral radii before Neumann–Schulz refinement.

Table 9: Approximation quality of the representations in Theorem 1. Mean relative Frobenius error ε_F and spectral radius ρ over 50 matrices per family ($N = 4096$, $r = 16$, two-hop pattern).

	Low-rank only		Sparse only		Sparse + low-rank	
Matrix family / target	ε_F	ρ	ε_F	ρ	ε_F	ρ
Poisson, A^{-1}	0.260	1.000	0.937	18.11	0.172	6.27
Diag.-dominant SPD, A^{-1}	0.804	1.000	0.040	1.281	0.038	0.238
Diag.-dominant SPD, $A^{-1/2}$	0.982	1.000	0.017	0.608	0.016	0.111
Helmholtz, A^{-1}	0.465	1.000	0.823	13.65	0.272	5.54

Table 10: Effect of the sparse and low-rank components on Helmholtz preconditioning. Mean $\rho(I - AM)$ before refinement and the fraction of cases with $\rho < 1$.

Variant	$N = 1600$	$N = 4096$	$\rho < 1$
Sparse + low-rank	**0.659**	**0.836**	**100%**
Sparse only	5.76	5.81	0%
Low-rank only	1.0000	1.0000	0%

The combined representation has the lowest Frobenius error in all four families, consistent with the complementary roles of the two components in Theorem 1. For the SPD inverse, adding a low-rank term to the sparse approximation changes the relative error from 0.040 to 0.038 but reduces ρ much more sharply, from 1.281 to 0.238. The modest change in approximation error thus accompanies a substantial improvement in contraction. Residual-based training further exploits this representation for preconditioning: on Helmholtz at $N = 4096$, it achieves $\rho = 0.836$, compared with 5.54 for the Frobenius approximation (Table 10). The network therefore learns a better preconditioner within the same sparse-plus-low-rank class.

Both components are needed for contraction in the trained comparison. Removing the low-rank branch raises $\rho(I - AM)$ from 0.659 to 5.76 at $N = 1600$ and from 0.836 to 5.81 at $N = 4096$. Removing the sparse branch gives $\rho = 1.0000$ on every sample. If $Mx = 0$ for a nonzero x, then $(I - AM)x = x$, so a rank-deficient operator leaves at least one direction uncontracted. The low-rank component improves the approximation, while the sparse component supplies the coverage needed for contraction across the full space.

E.2 Distribution shift and adaptation

We vary coefficient contrast and anisotropy to test transfer beyond the training distribution. A single set of weights is trained jointly on Poisson ($\sigma = 0$) and Helmholtz ($\sigma < 0$) samples. The training distribution contains smooth isotropic diffusion coefficients κ, coefficient contrast at most 8, reaction coefficients $\sigma \in [-0.3\lambda_1, 0]$, and matrix dimensions $N \leq 3600$, where λ_1 denotes the smallest eigenvalue of the corresponding diffusion operator. Evaluation uses $N = 4096$; each setting contains ten matrices, five Poisson and five Helmholtz. We report mean $\rho(I - AM)$ and GMRES iterations to relative residual 10^{-8}.

Coefficient contrast. The jointly trained model retains effective preconditioning as isotropic contrast increases far beyond the training range (Table 11). GMRES needs about nine iterations through contrast 21, although training stops at 8. Even at contrast 660, about $83\times$ the training maximum, all ten systems converge in a mean of 19.3 iterations. The mean spectral radius is 2.277 but the median is 0.991, so a few difficult matrices account for much of the increase. The solve results show that the model remains useful under this substantial coefficient shift even when the mean spectral radius exceeds one.

Table 11: Generalization to larger coefficient contrast. Mean spectral radius and GMRES iterations at $N = 4096$; training contrast is at most 8.

Coefficient contrast	1	7	21	81	660
Within training range	Yes	Yes	No	No	No
Mean $\rho(I - AM)$	0.417	0.407	0.418	0.805	2.277
Mean GMRES iterations	9.5	9.4	9.4	12.1	19.3

Anisotropy. We next change the directional ratio $\varepsilon = \kappa_y/\kappa_x$, while training uses only $\varepsilon = 1$. This shift is substantially harder (Table 12). At $\varepsilon = 0.5$, the mean iteration count increases from 9.6 to 113.5; at $\varepsilon \in \{0.1, 0.01\}$, the solver does not reach tolerance within 600 iterations. Changing directional structure therefore degrades performance much more sharply than increasing isotropic contrast.

Adaptation with limited data. We adapt the decoder and flow for one epoch using the self-supervised objective, with the encoders frozen. The 160 fine-tuning matrices include 100 anisotropic examples and 60 isotropic examples to retain performance on the original distribution. Training takes 142 seconds on one GPU and includes only $\varepsilon = 1$ and 0.1.

This short adaptation improves solves at both anisotropy ratios absent from fine-tuning: at $\varepsilon = 0.5$, the mean iteration count falls from 113.5 to 16; at $\varepsilon = 0.01$, systems that fail to converge within 600 iterations require 51 iterations on average after adaptation. At the fine-tuned ratio $\varepsilon = 0.1$, it drops to 26. Mean ρ is below one in all four settings, while isotropic iterations increase from 9.6 to 14. Thus 160 matrices and 142 seconds of self-supervised training extend the model to anisotropy ratios not used for fine-tuning, with a moderate loss of performance on the original isotropic distribution.

Table 12: Adaptation to anisotropic diffusion. Mean spectral radius and GMRES iterations over ten matrices per setting at $N = 4096$, before and after one epoch of fine-tuning.

Anisotropy ε	1.0	0.5	0.1	0.01
Before: $\rho(I - AM)$	0.414	0.952	7.75	37.1
Before: GMRES iterations	9.6	113.5	> 600	> 600
In fine-tuning set	Yes	No	Yes	No
After: $\rho(I - AM)$	0.599	0.731	0.855	0.916
After: GMRES iterations	14	16	26	51

F Sparse and low-rank approximation

The proof separates spectral truncation from correction on the graph pattern. Orthogonality gives the error of each step. We work in the following general setting: A is real symmetric with orthonormal eigenpairs (λ_m, v_m), and f is real and finite on its spectrum. For a set I_r of r selected modes, define

$$L_r = \sum_{m \in I_r} f(\lambda_m) v_m v_m^\top, \qquad H_r = f(A) - L_r, \qquad M^\star = L_r + [H_r]_{\mathcal{S}_k}.$$

The admissible representation class is $\mathcal{M}_{k,r} = \{S + L : \mathrm{supp}(S) \subseteq \mathcal{S}_k,\ \mathrm{rank}(L) \le r\}$. We first establish the exact sparse-correction identity, then relate it to a polynomial approximation on the same pattern.

F.1 Proof of the sparse-correction identity

Write $P_m = v_m v_m^\top$. Since the eigenvectors are orthonormal, these rank-one projectors are orthonormal in the Frobenius inner product:

$$\langle P_m, P_n \rangle_F = \mathrm{tr}(v_m v_m^\top v_n v_n^\top) = (v_m^\top v_n)^2 = \delta_{mn}. \tag{8}$$

Consequently, the residual after retaining the modes in I_r has norm

$$H_r = \sum_{m \notin I_r} f(\lambda_m) P_m, \qquad \|H_r\|_F^2 = \sum_{m \notin I_r} |f(\lambda_m)|^2. \tag{9}$$

An orthonormal eigenbasis suffices, including within repeated eigenspaces.

The pattern restriction is also an orthogonal projection, now onto the matrix entries indexed by $\mathcal{S}_k$. Its output $[H_r]_{\mathcal{S}_k}$ and the remaining entries $H_r - [H_r]_{\mathcal{S}_k}$ have disjoint support. Therefore,

$$\begin{aligned} \|f(A) - M^\star\|_F^2 &= \|H_r - [H_r]_{\mathcal{S}_k}\|_F^2 \\ &= \|H_r\|_F^2 - \|[H_r]_{\mathcal{S}_k}\|_F^2 \\ &= \sum_{m \notin I_r} |f(\lambda_m)|^2 - \|[H_r]_{\mathcal{S}_k}\|_F^2. \end{aligned} \tag{10}$$

Moreover, for any S supported on $\mathcal{S}_k$, the same decomposition gives

$$\|f(A) - (L_r + S)\|_F^2 = \|H_r - [H_r]_{\mathcal{S}_k}\|_F^2 + \|[H_r]_{\mathcal{S}_k} - S\|_F^2. \tag{11}$$

The second term vanishes exactly when $S = [H_r]_{\mathcal{S}_k}$, proving uniqueness for the fixed L_r. Since $\operatorname{rank}(L_r) \le r$, this construction belongs to $\mathcal{M}_{k,r}$ and yields

$$\inf_{M \in \mathcal{M}_{k,r}} \|f(A) - M\|_F^2 \le \sum_{m \notin I_r} |f(\lambda_m)|^2 - \|[H_r]_{\mathcal{S}_k}\|_F^2. \tag{12}$$

The construction thus bounds the best error in the full class. Its sparse term strictly improves on the chosen spectral truncation whenever $[H_r]_{\mathcal{S}_k} \neq 0$. The optimality statement in Theorem 1 holds with L_r fixed; jointly optimizing both components can improve the approximation further.

For SPD A and $f(\lambda) = \lambda^{-\alpha}$, $\alpha > 0$, the largest coefficients $f(\lambda_m)$ correspond to the smallest eigenvalues of A. Taking $I_r = \{1, \dots, r\}$ in increasing eigenvalue order gives Equation (6). In this case, if $r < N$, the nonzero positive semidefinite residual H_r has positive trace. Since $\mathcal{S}_k$ includes the diagonal, its restriction is nonzero, and the sparse correction strictly improves the Frobenius error.

F.2 A POLYNOMIAL SPARSE COMPONENT

A polynomial of degree at most k provides another sparse correction. The graph argument below underlies polynomial approaches to matrix-function approximation (Demko et al., 1984; Benzi & Golub, 1999; Benzi & Razouk, 2007).

Lemma 2 (Polynomial sparsity). *Let $\mathcal{S}_k = \{(i,j) : d_G(i,j) \le k\}$, including the diagonal, where G is the sparsity graph of the symmetric matrix A. If $\deg p \le k$, then $\operatorname{supp}(p(A)) \subseteq \mathcal{S}_k$.*

Proof. An entry $[A^\ell]_{ij}$ is a sum of products along walks with at most ℓ graph edges. Thus it is zero when $d_G(i,j) > \ell$. Every term in $p(A) = \sum_{\ell=0}^{k} c_\ell A^\ell$ is zero outside $\mathcal{S}_k$, with $A^0 = I$ supported on the diagonal. □

By the lemma, $L_r + p(A)$ belongs to $\mathcal{M}_{k,r}$. To evaluate its error, define $g_m = 0$ for $m \in I_r$ and $g_m = f(\lambda_m)$ otherwise. Expanding in the same orthonormal projectors as above gives

$$\begin{aligned} f(A) - L_r - p(A) &= \sum_m \big(g_m - p(\lambda_m)\big) P_m, \\ \|f(A) - L_r - p(A)\|_F^2 &= \sum_{m \notin I_r} \big(f(\lambda_m) - p(\lambda_m)\big)^2 + \sum_{m \in I_r} p(\lambda_m)^2. \end{aligned} \tag{13}$$

The polynomial fits the residual modes in the first sum, but also acts on the retained modes, producing the second sum. Setting $p \equiv 0$ recovers pure spectral truncation; optimizing its coefficients gives a least-squares problem. Every such polynomial is supported on $\mathcal{S}_k$, so the exact pattern restriction in Equation (10) achieves at least as small an error.

For inverse operators, one can exhibit a strict improvement even with a linear polynomial. If A is nonsingular, $f(\lambda) = 1/\lambda$, $r < N$, and $k \ge 1$, then $p(A) = cA$ is admissible and

$$\|H_r - cA\|_F^2 = \|H_r\|_F^2 - 2c(N - r) + c^2 \|A\|_F^2. \tag{14}$$

Here $\langle H_r, A\rangle_F = \sum_{m\notin I_r} \lambda_m^{-1}\lambda_m = N - r$. Taking $c = (N-r)/\|A\|_F^2$ therefore reduces the squared error by $(N-r)^2/\|A\|_F^2$. This calculation makes the contribution of a sparse component explicit for both definite and indefinite inverse operators.

F.3 Scope of the result

The general identity applies whenever A is symmetric and f is real and finite on its spectrum. For a nonsingular symmetric indefinite matrix and $f(\lambda) = 1/\lambda$, selecting the r eigenvalues smallest in modulus removes the largest inverse coefficients. Fractional inverse powers in Equation (6) are stated for SPD matrices. A Schur complement depends on a block partition and is not, in general, a scalar spectral function of the full input matrix; its performance is evaluated empirically.

The approximation bounds support the sparse-plus-low-rank representation, while training optimizes its operators for the target task. Appendix E reports the numerical references and trained ablations that connect this approximation capacity to preconditioning performance. Section 4.4 evaluates the role of conditional generation.

G Latent geometry and flow trajectories

G.1 Construction of the latent-space visualization

Figure 2 visualizes two matrices with $N = 4096$. For each matrix, three reference latent codes b_1, b_2, b_3 define an affine plane. The reference codes come from optimization in latent space with the decoder fixed, retaining separated candidates that decode to low spectral-radius estimates. Let $\mu = (b_1 + b_2 + b_3)/3$. We form an orthonormal basis of their plane by

$$q_1 = \frac{b_2 - b_1}{\|b_2 - b_1\|_2}, \qquad \widetilde{q}_2 = (b_3 - b_1) - q_1 q_1^\top (b_3 - b_1), \qquad q_2 = \frac{\widetilde{q}_2}{\|\widetilde{q}_2\|_2}. \tag{15}$$

Any latent code z is plotted at $\big(q_1^\top(z-\mu), q_2^\top(z-\mu)\big)$. The three reference codes lie exactly on this plane; other codes are projected onto it.

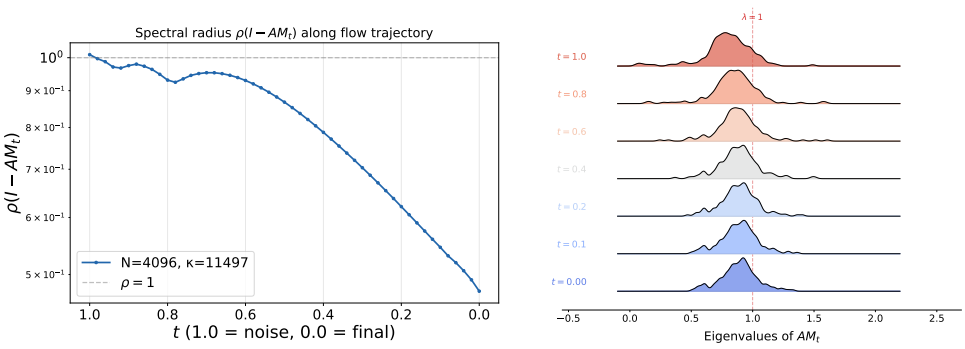
To obtain the background, we evaluate a 61×45 grid. At each coordinate (x, y), we reconstruct $z = \mu + xq_1 + yq_2$ and decode it while holding the matrix condition fixed. For the resulting residual operator $E = I - AM(z)$, we start from five Gaussian unit vectors and apply 15 normalized power iterations to each. The plotted estimate is the largest $\|Ew_j\|_2$ over the five final unit vectors. The same initial directions are used across grid points. Contours therefore show estimated operator quality on the affine plane.

For the trajectories, we draw 32 independent Gaussian latent initializations and use 12 Heun steps, recording the initial state and every subsequent state. The paths are projected onto the reference plane and colored by the reference code nearest to their endpoint in the full latent space. Stars mark the references, the square marks the deterministic regression output, and open and filled circles mark flow starts and endpoints. The annotations evaluate operators at their original latent codes, so they can differ from the background at the corresponding projected location. The paths follow the learned velocity in the full latent space. Their projection shows where these paths lie relative to operator quality on the reference plane. The best-of-32 annotations describe these two examples, whereas Table 5 evaluates 16 draws per matrix across the test set.

G.2 Operator quality along a flow trajectory

We decode intermediate flow states to examine how operator quality changes between noise and the generated code. For a Helmholtz matrix with $N = 4096$ and condition number $\kappa = 11{,}497$, we integrate from $t = 1$ to $t = 0$ in 12 steps and evaluate the decoded operators M_t.

Spectral radius decreases overall from approximately 1.0 to 0.45 (Figure 3a). The eigenvalues of AM_t move from a broad distribution over $[-0.5, 2.5]$ toward a concentration near one (Figure 3b). Both views show progressive improvement along this sampled trajectory. Training evaluates the decoded endpoint and does not impose monotonic improvement at intermediate times.



(a) Spectral radius

(b) Eigenvalue distribution

Figure 3: Operator quality along the flow trajectory. Spectral radius and eigenvalue distributions for a Helmholtz matrix with $N = 4096$ and $\kappa = 11{,}497$.

H Training and inference

Algorithm 1 gives the two training stages, and Algorithm 2 gives inference. Supervised initialization uses target operators to learn the latent representation. Joint optimization then evaluates operators generated from A alone. Throughout, M denotes the decoded operator.

Losses and gradient updates. Let e denote the training epoch, distinct from the flow time t. The two objectives are

$$\mathcal{L}_{\text{init}}(e) = w_{\text{edge}}(e)\mathcal{L}_{\text{edge}} + w_{\text{res}}(e)\mathcal{L}_{\text{res}} + w_{\text{flow}}(e)\mathcal{L}_{\text{flow}}, \tag{16}$$

$$\mathcal{L}_{\text{joint}}(e) = w_{\text{res}}(e)\mathcal{L}_{\text{res}} + w_{\text{power}}(e)\mathcal{L}_{\text{power}}. \tag{17}$$

During initialization, $\mathcal{L}_{\text{edge}}$ compares predicted and target entries on $\mathcal{S}_k$. It provides the main reconstruction signal, with $w_{\text{power}} = 0$. The encoders and decoder receive reconstruction and residual gradients. Flow matching updates only the flow, treating the encoder's code and condition as fixed targets for that update.

The residual loss tests generated operators on random directions: $(AM - I)V_{\text{loss}}$ for inverse operators and $(MAM - I)V_{\text{loss}}$ for inverse square roots. These residuals can be evaluated using A and M without target-operator responses. The power loss estimates the corresponding residual spectral radius through differentiable power iteration.

In the second stage, the probe path is bypassed and both encoders are frozen. The flow samples a code from A's condition, the decoder constructs M, and gradients through this path update both modules jointly. The supervised terms are removed. The residual term remains active throughout, while w_{power} increases from zero late in training. Thus the power term supplements the residual objective within the same training stage.

Input and loss probes. For each training dimension, input-probe entries are sampled independently as $(V_{\text{in}})_{ij} \sim \mathcal{N}(0, 1/N)$ and then held fixed. Keeping the probes fixed avoids adding resampling noise to the target responses $f(A)V_{\text{in}}$ and the resulting latent supervision during initialization. Probe counts are given in Table 7. Loss probes V_{loss} are separate random directions, resampled throughout residual optimization to test the generated operator using A, without target-operator responses.

Algorithm 1 Two-stage training of FlowSMO

Require: Small-matrix data $\mathcal{D}_{\text{sup}} = \{(A, f(A))\}$, larger-matrix data $\mathcal{D}_{\text{self}} = \{A\}$
Require: Fixed input probes V_{in}; epoch-dependent loss weights
Stage 1: supervised initialization
for each initialization epoch e **do**
  for each $(A, f(A)) \in \mathcal{D}_{\text{sup}}$ **do**
    $(h_{\text{cond}}, g) \leftarrow \text{CondEnc}(A)$
    $z_0 \leftarrow \text{ProbeEnc}(f(A)V_{\text{in}}, h_{\text{cond}}, g)$
    $\widetilde{z}_0 \leftarrow z_0 + \epsilon\xi, \xi \sim \mathcal{N}(0, I)$
    $M \leftarrow \text{Decoder}(\widetilde{z}_0, h_{\text{cond}}, g)$
    Resample loss probes if the residual term is active
    Update encoders and decoder with $w_{\text{edge}}(e)\mathcal{L}_{\text{edge}} + w_{\text{res}}(e)\mathcal{L}_{\text{res}}$
    Treat (z_0, g) as fixed targets; sample $z_1 \sim \mathcal{N}(0, I)$ and $t \sim \mathcal{U}[0, 1]$
    $z_t \leftarrow (1 - t)z_0 + tz_1$
    Update only the flow using $w_{\text{flow}}(e)\mathcal{L}_{\text{flow}}$ in (2)
  end for
end for
Stage 2: self-supervised joint optimization
Freeze both encoders; bypass the probe path; set $w_{\text{edge}} = 0$
for each joint-training epoch e **do**
  for each $A \in \mathcal{D}_{\text{self}}$ **do**
    $(h_{\text{cond}}, g) \leftarrow \text{CondEnc}(A)$ without gradients
    Sample fresh noise z_1 and loss probes V_{loss}
    $z \leftarrow \text{FlowSample}_\theta(z_1, g)$, retaining the gradient path
    $M \leftarrow \text{Decoder}(z, h_{\text{cond}}, g)$
    Evaluate $\mathcal{L}_{\text{res}}$ from A, M, V_{loss}
    Add $w_{\text{power}}(e)\mathcal{L}_{\text{power}}$ late in training, retaining the residual term
    Update flow and decoder jointly using (17)
  end for
end for

Algorithm 2 FlowSMO inference

Require: New matrix A, trained models, integration steps $S = 12$ by default
$h_{\text{cond}}, g \leftarrow \text{GNN}_{\text{cond}}(A)$ *// encode condition only*
$z_1 \sim \mathcal{N}(0, I)$ *// noise latent*
if $S = 1$ **then**
  $z_0 \leftarrow z_1 - v_\theta(z_1, 1, g)$ *// one-step flow*
else
  $z_0 \leftarrow \text{HeunODE}(v_\theta, z_1, g, S,\ 1 \to 0)$ *// 12 steps by default*
end if
$M \leftarrow \text{Decoder}(z_0, h_{\text{cond}}, g)$ *// sparse + low-rank output*
return $M = M_{\text{sparse}} + U\,\text{diag}(\sigma)\,U^\top$