Learning to Discover Iterative Spectral Algorithms

Zihang Liu[*1,2] **Oleg Balabanov**[*1,2,4] **Yaoqing Yang**[3] **Michael W. Mahoney**[1,2,4]
[1]International Computer Science Institute [2]University of California, Berkeley
[3]Dartmouth College [4]Lawrence Berkeley National Laboratory
{zihang.liu,obalaban}@berkeley.edu
yaoqing.yang@dartmouth.edu mmahoney@stat.berkeley.edu

Abstract

We introduce AUTOSPEC, a neural network framework for discovering iterative spectral algorithms for large-scale numerical linear algebra and numerical optimization. Our self-supervised models adapt to input operators using coarse spectral information (e.g., eigenvalue estimates and residual norms), and predict recurrence coefficients for computing or applying a matrix polynomial tailored to a downstream task. The effectiveness of AUTOSPEC relies on three ingredients: an architecture whose inference pass implements short, executable numerical linear algebra recurrences; efficient training on small synthetic problems with transfer to large-scale real-world operators; and task-defined objectives that enforce the desired approximation or preconditioning behavior across the range of spectral profiles represented in the training set. We apply AUTOSPEC to discovering algorithms for representative tasks on spd matrices: accelerating matrix function approximation; accelerating sparse linear solvers; and spectral filtering/preconditioning for eigenvalue computations. On real-world matrices, the learned procedures deliver up to order-of-magnitude improvements in accuracy and/or reductions in iteration count, relative to spectrum-agnostic baselines. We find clear connections to classical theory: the induced polynomials may exhibit equioscillation behavior characteristic of Chebyshev polynomial approximation. The code is available at: https://github.com/zihanghliu/AutoSpec.

1 Introduction

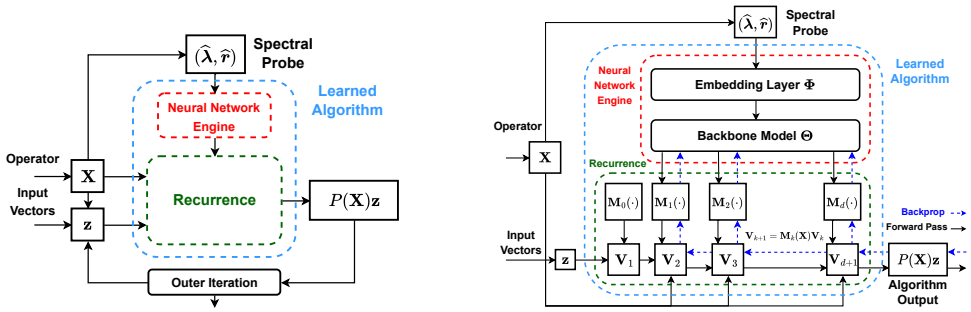
Machine learning (ML) has shown growing potential to scale scientific discovery by automating parts of the algorithm design pipeline [19, 20, 43, 55, 71]. However, many successful discovery frameworks search over discrete hypothesis classes (e.g., programs or pseudocode), which does not transfer cleanly to numerical linear algebra (NLA) and numerical optimization, where effective methods are typically continuous objects that must adapt to operator-specific (often spectral) structure.

In this paper, we propose AUTOSPEC, a neural network framework for discovering a broad class of iterative spectral methods for NLA and numerical optimization. Our models learn mappings from spectral probes—e.g., a small set of coarse eigenvalue approximations with associated residual norms—to iteration coefficients that define an operator polynomial (or, more generally, an operator function) tailored to a downstream task. At inference time, given a new operator, we (i) extract a probe using a short warm-start eigensolver run, (ii) feed this probe to the trained model to predict iteration coefficients, and (iii) execute the resulting iterative process. This ML workflow mirrors standard NLA practice, where inexpensive spectral information enables methods to adaptively exploit and shape the operator's spectrum.

Our algorithm discovery approach is fully self-supervised: we train end-to-end by minimizing NLA task-specific objectives, using only task performance as supervision. As a result, the learned methods

[*]These authors contributed equally.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) Inference in NLA tasks.

(b) Algorithm structure.

Figure 1: AUTOSPEC framework. (a) A trained neural network engine, paired with an executable recurrence, defines a discovered numerical algorithm for downstream NLA tasks. (b) An end-to-end differentiable algorithmic structure: given operator $\mathbf{X}$, a spectral probe $(\widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{r}})$ consisting of eigenvalue estimates and residual norms, is extracted and fed to the neural engine to produce the coefficients of a degree-d polynomial $P(\cdot)$. The polynomial is implemented in a matrix-free manner via a short recurrence: starting from $\mathbf{V}_0 = \mathbf{z}$, we iterate $\mathbf{V}_{k+1} = \mathbf{M}_k(\mathbf{X})\mathbf{V}_k$, where each $\mathbf{M}_k$ is parameterized by the engine. The terminal state returns the action $\mathbf{V}_{d+1} = P(\mathbf{X})\mathbf{z}$ of the polynomial to inputs $\mathbf{z}$, which allows self-supervised training by backpropagating task-defined NLA losses on $\mathbf{V}_{d+1}$.

are optimized to outperform fixed baselines through task feedback, rather than by imitating existing algorithms. Our design offers three advantages: (i) a unified parameterization spanning classical iterative spectral algorithmic structures, enabling discovery across multiple NLA tasks; (ii) training on small synthetic spectral instances with transfer to large-scale operators, producing a compact and deployable learning engine; and (iii) reliance only on coarse spectral information at inference time, improving applicability when spectral access is limited or computational budgets are tight.

Moreover, we observe clear connections between the discovered algorithms and classical polynomial approximation theory for matrix functions. For instance, some learned polynomials exhibit near-minimax, equiripple behavior, consistent with the Chebyshev equioscillation theorem for minimax approximation. At the same time, our approach is not limited to approximating a prescribed function: it can optimize task-level objectives, such as residual decay or conditioning of a preconditioned operator and the preconditioner. This flexibility can be beneficial for more complex NLA tasks.

We apply AUTOSPEC to discovering matrix-free iterative algorithms[1] for several tasks on positive-definite matrices: solving linear systems; computing eigenvalues; and approximating matrix-function actions. Across a suite of real-world sparse matrices, the learned methods significantly improve accuracy and/or reduce iteration counts, in some cases by orders of magnitude relative to basic baseline algorithms that do not exploit spectral information. We summarize our main contributions as follows:

- We propose AUTOSPEC, a general framework for discovering iterative spectral algorithms with neural networks. Unlike symbolic algorithm discovery methods, our approach learns continuous algorithmic updates suited to NLA and related numerical tasks.
- AUTOSPEC is demonstrated to discover nontrivial recurrence structures already when using a simple neural architecture and without supervision. We identify connections to classical approximation theory by noticing that some discovered recurrences exhibit behavior consistent with Chebyshev equioscillation theorem.
- We apply AUTOSPEC to three classical NLA tasks on symmetric positive-definite matrices and show that it learns to exploit spectral information from the input, yielding up to order-of-magnitude gains in accuracy and/or iteration count.

2 Related Works

Algorithm Discovery with ML. ML has become a key driver of scientific discovery by enabling automated algorithm design. This problem is naturally framed as a search over a high-dimensional combinatorial space of candidate operations. Many works adopt discrete program spaces, where symbolic regression remains a dominant paradigm, spanning classical evolutionary methods [35, 62]

[1] Using an operator only through its action on vectors (matvecs).

and neural-guided approaches [26, 33, 36, 45, 52, 68]. Moreover, by formulating the discovery process as a sequential decision-making task, reinforcement learning (RL) becomes a useful method for discovering more complex algorithms [20, 43, 50, 51]. More recently, researchers have used the reasoning capabilities of large language models (LLMs) to discover novel algorithms and solve advanced math problems [14, 49, 56]. In this work, we propose a framework for discovering algorithms that can be used in NLA and numerical optimization. For optimization, recent works have used deep unfolding [65], meta-learning [3, 38, 54, 70], and RL [9, 13] to discover optimization algorithms. For NLA, neural networks have been used to approximate the action of a matrix inverse on vectors, with the learned maps then applied as preconditioners within linear system solvers [25, 32, 37, 39, 42, 57, 66]. These methods are not directly comparable to AUTOSPEC as they typically train a separate model for each input matrix, rather than discovering a transferable algorithmic procedure. Notably, AUTOSPEC could be applied as a second-level accelerator for the resulting preconditioned system solves.

Appendix A discusses related works on spectral approximation and preconditioning methods in NLA.

3 AUTOSPEC Framework

In this section, we introduce the AUTOSPEC framework: an end-to-end differentiable approach built around a unified representation of iterative spectral algorithms, with modular components that can be learned by neural networks.

Framework Setup. For any diagonalizable matrix $\mathbf{X} \in \mathbb{R}^{n\times n}$, we define a *spectral probe* $(\widehat{\boldsymbol{\lambda}}(\mathbf{X}), \widehat{\boldsymbol{r}}(\mathbf{X}))$, consisting of a small set of approximate eigenvalues $\widehat{\boldsymbol{\lambda}}$ from a task-relevant portion of the spectrum (this could be the top eigenvalues, e.g., if one is interested in low-rank approximation, but for many applications it is not), optionally augmented with auxiliary quantities such as residual norms $\widehat{\boldsymbol{r}}$. This mirrors common NLA workflows, where inexpensive spectral estimates (e.g., Ritz values from short Lanczos or Arnoldi runs) are used to guide the construction of preconditioners and approximations. In particular, our models in Section 5 can take $(\widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{r}})$ to consist of a few extreme eigenvalue estimates and their associated residuals, obtained via a cheap eigensolver warm-start.

Then, we train a *neural network engine*,

$$(\widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{r}}) \;\mapsto\; P(\cdot), \tag{1}$$

that maps an input spectral probe to an operator polynomial (or more generally, an operator function) designed for downstream tasks. The designated objective may be classical, such as minimizing a matrix function approximation error $\|P(\mathbf{X}) - f(\mathbf{X})\|$, or non-standard, such as e.g. minimizing both the condition number of the preconditioned operator $P(\mathbf{X})\mathbf{X}$, which governs basic linear solver convergence, and the condition number of the preconditioner $P(\mathbf{X})$, which promotes robustness of Krylov solvers.

We parametrize the polynomials $P(\cdot)$ to admit an executable short recurrence that applies the matrix polynomial $P(\mathbf{X})$ efficiently to input vectors and matrices. In this setting, the neural engine predicts the recurrence coefficients and thereby can be realized to govern the convergence (and numerical stability) of the iterative process.

Here is the important point: the combination of a neural network engine and an executable recurrence defines a learned algorithm. This algorithm can then be deployed for NLA downstream tasks, such as preconditioning linear solver iterations. See Figure 1 for an illustration.

For robust training and for efficient evaluation at inference time, we can represent $P(\mathbf{X})$ implicitly via a state recurrence:

$$\mathbf{V}_{k+1} = M_k(\mathbf{V}_k, \mathbf{X}), \qquad 0 \le k \le d, \tag{2}$$

The initialization for matrix actions is $\mathbf{V}_0 = \mathbf{z}$ so that $\mathbf{V}_{d+1} = P(\mathbf{X})\mathbf{z}$; when $P(\mathbf{X})$ is needed to be formed explicitly (e.g., in small-scale training or diagnostics) we take $\mathbf{V}_0 = \mathbf{I}$, yielding $\mathbf{V}_{d+1} = P(\mathbf{X})$. The operators $M_k(\cdot,\cdot)$ are specified modularly to realize short NLA-style recurrences.

Modular linear transitions. We primarily focus on transitions that are linear in the state,

$$M_k(\mathbf{V}_k, \mathbf{X}) := \mathbf{M}_k(\mathbf{X})\,\mathbf{V}_k, \tag{3}$$

where $\mathbf{M}_k(\mathbf{X})$ is assembled from blocks $\mathbf{M}_k^{(i,j)} \in \mathbb{R}^{n\times n}$, each given by a low-degree matrix polynomial $M_k^{(i,j)}(\mathbf{X})$. The final transition $\mathbf{M}_d(\mathbf{X})$ is constrained to have a single block-row, so the terminal state extracts the desired polynomial output. With this parameterization, learning the neural engine (1) reduces to learning maps $(\widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{r}}) \mapsto M_k^{(i,j)}(\cdot)$ that specify the block polynomials. The same modular construction also supports richer transition classes; see Appendix B.1 for extensions to higher-order (state-polynomial) and rational recurrences.

Matrix-free regime. Our main target is the large-scale sparse setting, where constructing or storing $P(\mathbf{X})$ is infeasible. Accordingly, we seek recurrences that compute the action $\mathbf{z} \mapsto P(\mathbf{X})\mathbf{z}$ using only a small number of applications of $\mathbf{X}$. Under (3), the action is obtained by composing the transitions,

$$P(\mathbf{X})\mathbf{z} = \mathbf{M}_d(\mathbf{X})\,\mathbf{M}_{d-1}(\mathbf{X})\ldots\mathbf{M}_0(\mathbf{X})\,\mathbf{z},$$

without ever forming $P(\mathbf{X})$ (or any dense $n\times n$ block), provided that applying each block polynomial $M_k^{(i,j)}(\mathbf{X})$ is implemented via a short sequence of $\mathbf{X}$ matvecs. To control per-step cost and improve robustness in sparse, matrix-free regimes, we can further restrict the transition matrices to use at most one application of $\mathbf{X}$.

For many downstream tasks such as preconditioning, the polynomial action is only required up to a scale factor. The sequential state-transition representation allows normalization of intermediate states without affecting the result up to scaling, enabling control of numerical growth and improving stability during training and inference (see Section 4.2).

3.1 State Transition View

In our empirical evaluation, we chose linear transitions with the following block parameterization:

$$\mathbf{M}_0 = \begin{bmatrix}\mathbf{I} & \mathbf{I} & \mathbf{X}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}} \quad \mathbf{M}_d = \begin{bmatrix}\gamma_d\mathbf{I} & \eta_d\mathbf{I} & \alpha_d\mathbf{I}+\beta_d\mathbf{X}\end{bmatrix}$$

$$\mathbf{M}_k = \begin{bmatrix}\mathbf{I} & \mathbf{0} & \rho_k\mathbf{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \\ \gamma_k\mathbf{I} & \eta_k\mathbf{I} & \alpha_k\mathbf{I}+\beta_k\mathbf{X}\end{bmatrix} \qquad 1\le k<d. \tag{4}$$

Under (4), the neural engine outputs scalar coefficient sequences $\{\rho_k, \gamma_k, \eta_k, \alpha_k, \beta_k\}$.

To interpret the blocks in (4), write the state as three $n\times n$ blocks $\mathbf{V}_k = \begin{bmatrix}\mathbf{A}_k & \mathbf{B}_k & \mathbf{C}_k\end{bmatrix}^{\mathrm{T}}$. Then for $1\le k<d$, the update $\mathbf{V}_{k+1} = \mathbf{M}_k\mathbf{V}_k$ is equivalent to (assuming that $\mathbf{X}$ is symmetric)

$$\begin{aligned}\mathbf{A}_{k+1} &= \mathbf{A}_k + \rho_k\,\mathbf{C}_k,\\ \mathbf{B}_{k+1} &= \mathbf{C}_k,\\ \mathbf{C}_{k+1} &= \gamma_k\mathbf{A}_k + \eta_k\mathbf{B}_k + (\alpha_k\mathbf{I}+\beta_k\mathbf{X})\,\mathbf{C}_k.\end{aligned} \tag{5}$$

Thus, $\mathbf{B}_k$ is a one-step delay storing the previous polynomial state, $\mathbf{C}_k$ stores the current polynomial state, and $\mathbf{A}_k$ is an accumulator that enables learned linear combinations of intermediate states.

The propositions below isolate two useful special cases: (i) a direct affine three-term polynomial recurrence; and (ii) basis generation followed by a learned expansion. Together, they motivate (4) as a compact hypothesis class that subsumes many classical polynomial approximation templates, appropriate for use in an ML workflow. In this view, learning $(\widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{r}}) \mapsto P(\cdot)$ amounts to *selecting and tuning* a member of this classical design space from a spectral probe, rather than committing a priori to a fixed polynomial family or coefficient rule.

Scenario I: Affine Three-term Polynomial Recurrence. Set $\rho_k = 0$, so $\mathbf{A}_k \equiv \mathbf{I}$. Then $\mathbf{C}_k$ follows an affine three-term recursion with constant injection through $\gamma_k\mathbf{I}$.

Proposition 3.1 (Affine three-term polynomial recurrence)**.** *Consider (4). If $\rho_k = 0$ for all $1\le k\le d$, then $\mathbf{C}_k$ satisfies (6) with $\mathbf{C}_0 = \mathbf{I}$ and $\mathbf{C}_1 = \mathbf{X}$, and for $k\ge 1$,*

$$\mathbf{C}_{k+1} = \gamma_k\mathbf{I} + \eta_k\mathbf{C}_{k-1} + \alpha_k\mathbf{C}_k + \beta_k\mathbf{X}\mathbf{C}_k. \tag{6}$$

This regime covers classical polynomial iterations defined by orthogonal polynomial recurrences (e.g., Chebyshev on a spectral interval) [44, 60, 69, 72], Clenshaw evaluation of expansions in three-term polynomial bases [15], and matrix-free polynomial filters for eigenvalue computations [7, 61, 73].

Scenario II: Basis Generation Plus Learned Expansion. This specialization separates *basis generation* from *coefficient selection*. Setting $\gamma_k = 0$ for $k < d$ removes the affine injection in (4), so $\mathbf{C}_k$ is generated by a homogeneous three-term recurrence. The accumulator $\mathbf{A}_k$ then forms a learned linear combination of these basis polynomials via ρ_k.

Proposition 3.2 (Expansion in a learned three-term basis)**.** *Consider (4). Assume $\gamma_k = 0$ for all $1 \le k < d$ and assume the readout uses $\eta_d = \alpha_d = \beta_d = 0$. Then $\mathbf{C}_k$ satisfies (7) with $\mathbf{C}_0 = \mathbf{I}$, $\mathbf{C}_1 = \mathbf{X}$, and for $1 \le k < d$,*

$$\mathbf{C}_{k+1} = \eta_k \mathbf{C}_{k-1} + \alpha_k \mathbf{C}_k + \beta_k \mathbf{X}\mathbf{C}_k, \tag{7}$$

and the output has the expansion form $\mathbf{C}_d = \gamma_d \mathbf{I} + \gamma_d \sum_{k=1}^{d-1} \rho_k \mathbf{C}_k$

This regime represents a common polynomial approximation template in scientific computing: generate a numerically stable polynomial basis through a three-term recurrence (e.g., Chebyshev or Legendre polynomials), and then choose expansion weights to approximate a target function on a prescribed spectral domain using minimax or least-squares criteria [4, 5, 18, 60, 69]. AUTOSPEC learns both the recurrence defining the basis and the expansion weights from the spectral probe, and can optimize objectives beyond direct function fitting, such as downstream residual reduction, eigengaps or conditioning.

3.2 Learning Objective

We propose a general learning objective for discovering different NLA algorithms with AUTOSPEC. For each NLA task, we adopt the metric r representing convergence rate of the iteration, and we evaluate the ratio between the learned algorithm and commonly used (spectrum-agnostic) baseline methods on a log scale. We denote our learning objective as $\rho_{\log}$:

$$\rho_{\log} = \frac{\log r_{\text{model}}}{\log r_{\text{baseline}}}. \tag{8}$$

We choose the ratio in logarithmic scale to alleviate bias towards certain spectral profiles when performing training/evaluation on a large, diverse set of matrices, enforcing desired preconditioning behaviors uniformly on all $\mathbf{X}$ in the dataset. Details on the metric r and baseline methods for different NLA tasks are provided in Appendix B.2.

4 Neural Network Engine of AUTOSPEC

4.1 Model Structure

In this section, we introduce the neural network engine of AUTOSPEC. As indicated in Figure 1, the neural network engine learns to construct a set of transition operators $\mathbf{M}_1, \ldots, \mathbf{M}_d$ for each target matrix $\mathbf{X}$. It consists of an embedding layer that extracts the relevant spectral features from coarse spectral information, and a backbone model that maps the features to transition operators.

Backbone Model (Pre-training). The backbone model mirrors the iterative update formula (3) using an unfolded modular structure, where each module represents a state transition operator $\mathbf{M}_k$, parameterized by coefficients $\{\rho_k, \gamma_k, \eta_k, \alpha_k, \beta_k\}$ learned by a neural network layer given spectral features $\mathbf{e}$ as input. Subsequently, the transition matrices $\mathbf{M}_k$ formulate the recurrence $\mathbf{V}_{k+1} = \mathbf{M}_k \mathbf{V}_k$, that results in the algorithm output $P(\mathbf{X}) = \mathbf{V}_{d+1}$. Since the generated recurrence is applied directly to training data, the backbone model is end-to-end trainable. The detailed backbone design is shown in Algorithm 1 in Appendix C.

Embedding Layer (Post-training). The embedding layer g_ϕ takes as input a coarse spectral probe, obtained from a cheap eigensolver warm-start. It learns a mapping from the spectral probe to an embedding vector, $(\widehat{\boldsymbol{\lambda}}, \widehat{\boldsymbol{r}}) \mapsto \mathbf{e}$, encoding the essential spectral features. Details of embedding layer is shown in Algorithm 2 in Appendix C

During training, we first pre-train the backbone model with a synthetic dataset, using prescribed spectral features as input. For post-training, we fix the backbone and train the embedding layer using spectral probes of the dataset as input. We fix the model during inference time on realistic problems.

4.2 Training and Evaluation Strategy

Here, we provide neural network training and evaluation strategies. Additional training details are provided in Appendices D and E.

Training and Evaluation Data. Our training procedure does not follow a conventional in-distribution learning setup based on sampling spectra from a fixed probability distribution. Instead, we train on small synthetic operators specifically designed to exhibit spectral properties (e.g., slow versus fast versus heavy-tailed decay of dominant eigenvalues) representative of given application regimes. Both pre-training and post-training stages use synthetic spectrums. At test time, we apply the trained model on realistic matrices whose spectral profiles are unknown but share similar structural properties to the training data. Thus, our evaluation measures generalization across operator instances within a structurally defined spectral class, rather than i.i.d. distributional generalization. By conditioning solely on dimension-agnostic spectral summaries rather than size-dependent features, the model learns a size-independent mapping from spectral input to polynomial coefficients. This enables extrapolation from training operators with $n \leq \mathcal{O}(10^3)$ to real-world matrices with dimensions up to $\mathcal{O}(10^6)$, provided they share similar spectral structure. This setup reflects the intended use case in NLA tasks, where algorithms are designed to generalize across broad operator classes characterized by spectral properties rather than fixed input distributions. Details of the training data generation procedure are provided in Appendix D.2.

Efficient Training on Diagonal Matrices. Training on full matrices can incur significant computational overhead. Observe that if two diagonalizable matrices $\mathbf{X}$ and $\mathbf{Y}$ share the same spectrum, then $P(\mathbf{X})$ and $P(\mathbf{Y})$ also share the same spectrum. Furthermore, our objective depends only on spectral quantities. Consequently, without loss of generality, we may restrict training to diagonal matrices of the form $\mathbf{X} = \mathrm{diag}(\boldsymbol{\lambda})$. This representation significantly improves training efficiency by avoiding operations on full matrices, especially when realized on fully synthetic data.

Stable Training with Scale-invariant Update. Learning higher degree algorithms poses the risk of numerical overflows for the iterative update when training the neural network. We note that objectives in NLA tasks are often scale-invariant, such as maximizing eigenvalue gaps or minimizing condition numbers. Under the state-space view of the learned recurrence in Section 3.1, we can see that scaling the states does not change the recurrence relation $\mathbf{V}_{k+1} = \mathbf{M}_k \mathbf{V}_k$, but it prevents the recurrence from growing exponentially with polynomial degree. Therefore, in training, we can normalize the state by its norm after each recurrence update. This effectively controls the norm of the states and prevents numerical overflows.

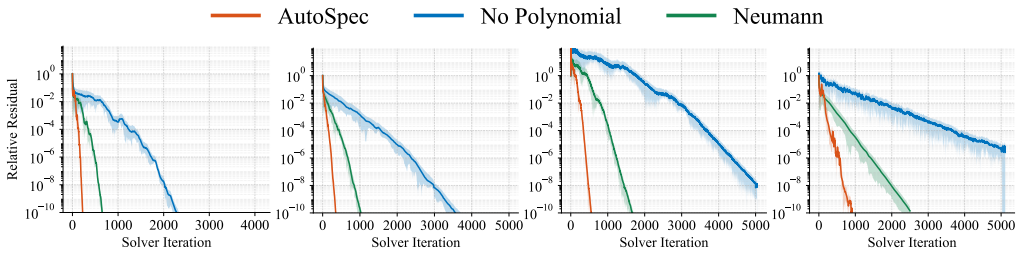
Optimization Objective. For all NLA tasks, we use a unified template for optimization objective of the model. Based on the learning objective (8), we propose a three-term unsupervised optimization loss $\mathcal{L}$ for training the neural network engine, that optimizes $\rho_{\log}$ for different spectral tasks:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left[c_1 \mathcal{L}_{\text{obj}}(\mathbf{X}_i) + c_2 \mathcal{L}_{\text{struct}}(\mathbf{X}_i) + c_3 \mathcal{L}_{\text{reg}}(\mathbf{X}_i) \right] \tag{9}$$

where N is the size of dataset $\{\mathbf{X}_i\}_{i=1}^N$. In (9): $\mathcal{L}_{\text{obj}}$ is the inverse of the learning objective $\rho_{\log}$, i.e., $\mathcal{L}_{\text{obj}} = \frac{1}{\rho_{\log}} = \frac{\log r_{\text{baseline}}}{\log r_{\text{model}}}$, which enforces desired spectral properties uniformly across the dataset; $\mathcal{L}_{\text{struct}}$ is a structural objective that imposes structural constraints on the learned algorithm (e.g., the condition number of the preconditioner); and $\mathcal{L}_{\text{reg}}$ is a regularization term that penalizes outliers and promotes learning more generalizable algorithms. More details on the loss function can be found in Appendix D.3.

5 Experimental Evaluation

We trained neural network engines for linear systems, eigenvalue problems, and matrix inverse square roots. This section evaluates their quality in adapting to spectral information through comparisons with spectrum-agnostic polynomial baselines.



(a) `G2_circuit` (b) `parabolic_fem` (c) `Shipsec5` (d) `Fault_639`

Figure 2: Convergence of CG solver with polynomial preconditioning. Curves show the median error and the 20th/80th percentiles over five random seeds.

5.1 Evaluation Setup

Linear systems and eigenproblems. We measure the quality of polynomials via the reduction in the number of outer Krylov iterations required to reach a prescribed tolerance, a standard metric for comparing polynomial preconditioners [8, 11, 58, 60, 61]. A trained AUTOSPEC neural engine is deployed as a plug-in and invoked from MATLAB to output the coefficient sequence that defines the polynomial updates. Spectral probes are obtained from a short warm start: for eigenproblems, we run 20 iterations of `eigs` (Krylov–Schur algorithm) using the same search subspace dimension as the main run; for linear systems, we run 50 or 200 Lanczos steps without storing the Lanczos basis.

We use a broad set of sparse test matrices drawn from the SuiteSparse Matrix Collection [16, 34], which spans electronic-structure, thermal finite-element, structural/mechanics, and circuit/model-reduction problems, with sizes $n \in [468,\ 1.6 \times 10^6]$ and condition numbers ranging by orders of magnitude. We also generated matrices using finite-difference discretizations of the Laplacian PDE and a finite-volume discretization of a diffusion PDE with spatially varying conductivity. Furthermore, all test matrices are spd. For linear systems, we use polynomial preconditioning as a second-level accelerator on top of inexpensive explicit preconditioners, such as Jacobi, AMG or incomplete Cholesky. In eigenvalue problems, the indefinite operators are implicitly squared. Full experimental details are given in Appendix G.

Matrix functions. We consider approximation of $\mathbf{X}^{-1/2}$ with a polynomial for covariance whitening of a regularized DNA similarity Gram matrix derived from *E. coli* K-12 MG1655 (U00096.3). Using a 9-mer spectrum embedding with $262{,}144$ features and 250 sequences, we construct a feature matrix $\mathbf{A}$ and compute inverse square root approximation of $\mathbf{X} = \mathbf{A}\mathbf{A}^{\mathrm{T}} + \lambda\mathbf{I}$. The matrix $\mathbf{A}$ is highly sparse (with nonzeros density around 0.375%).

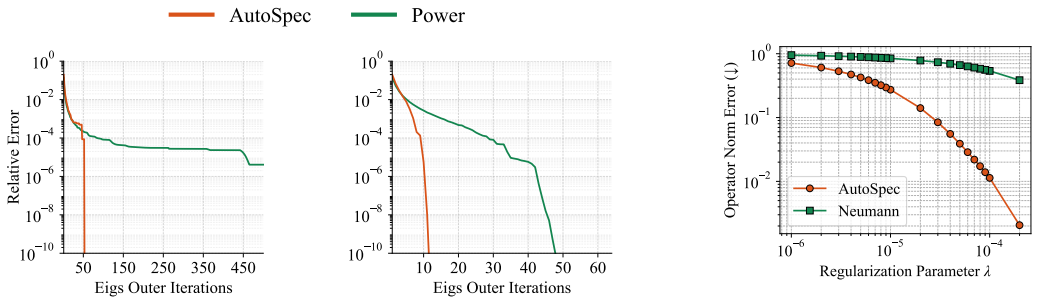
5.2 Preconditioned Linear Systems

We train a neural network engine to generate degree-11 polynomials accelerating Conjugate Gradient (CG) solvers, and evaluate its performance on test systems with random Gaussian right-hand sides. Table 3a reports the number of CG iterations required to reach a relative residual of 10^{-10} using learned and Neumann (truncated Taylor expansion) polynomial preconditioners, as well as CG without polynomial acceleration. The learned preconditioners substantially reduce outer iteration counts across all tested systems, with reductions of up to an order of magnitude relative to unaccelerated CG. Convergence curves for representative test systems are shown in Figure 2. See Appendix G for extended real-world experiments and Appendix I.2 for results on synthetic systems.

5.3 Eigenvalue Problems

For eigenvalue problems, the downstream task is to compute the k smallest magnitude eigenpairs using `eigs` under a fixed search subspace dimension budget l (with user-defined k and l [2]). After applying an identity shift to make the target modes dominant, we use a degree-21 polynomial transform generated by the neural network engine to enhance spectral separation near the target boundary, here chosen to be defined by the eigengap between the k-th and $1.5k$-th eigenvalues.

[2]Typically $l = c\,k$ for a small constant c, e.g., $c = 4$.



(1a) λ_{10} of `SiO2` ($k = 10$) (1b) λ_{20} of `thermal2` ($k = 20$) (2) Approximation of $\mathbf{X}^{-1/2}$

Figure 3: **Left:** Convergence of eigenvalue approximations versus `eigs` outer iterations for (shifted) `SiO2` and `thermal2`, for varying target numbers of eigenvalues k and subspace dimension $l = 4k$. The NN preconditioner is produced using a spectral probe from 20 `eigs` warm-start iterations. **Right:** Whitening the covariance matrix of the *E. coli* K-12 MG1655 reference genome sequence.

Figures 3 and 8 compare `eigs` with AUTOSPEC preconditioning versus `eigs` with a spectrum-agnostic power preconditioner baseline $P(\mathbf{X}) = \mathbf{X}^d$. The AUTOSPEC preconditioning consistently significantly outperforms the baseline across different (k, l) configurations and matrices, which indicates generalization. Figure 14 further shows that performance improves with probe quality, suggesting the neural engine adapts to the spectral profiles. See Appendix G for extended real-world matrix experiments and Appendix I.1 for complementary synthetic results.

5.4 Approximating Matrix Functions

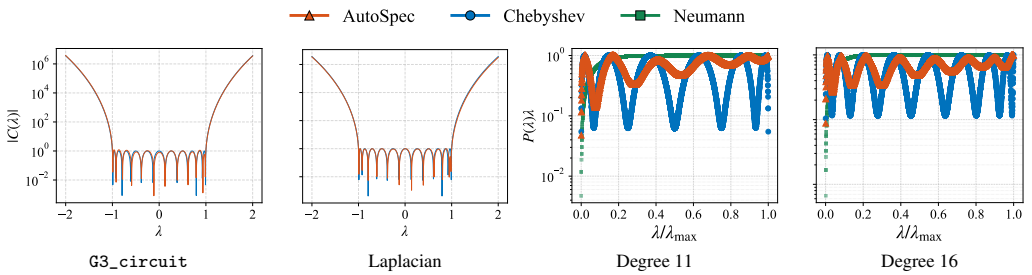
We train a neural network to predict a degree-21 polynomial $P(\mathbf{X})$ that approximates the matrix inverse square root, $P(\mathbf{X}) \approx \mathbf{X}^{-1/2}$, and we apply it to the covariance whitening of the regularized DNA sequence similarity Gram matrix. We measure whitening quality via the operator-norm residual $\|P(\mathbf{X})\,\mathbf{X}\,P(\mathbf{X}) - \mathbf{I}\|_2$, and we compare against a spectrum-agnostic truncated Neumann-series baseline. As shown in Figure 3, the learned approximation algorithm achieves order-of-magnitude improvement. See Appendix I.3 for complementary results on synthetic matrices.

6 Analysis and Potential to Advance the State of the Art

6.1 Connections to Classical Algorithms

Chebyshev polynomials are commonly used for spectral acceleration in NLA [60, 61]. Their characteristic properties are minimax optimality and equioscillation. For degree $d \geq 1$, the normalized Chebyshev polynomial $2^{1-d}T_d(x)$ minimizes $\max_{x\in[-1,1]} |P(x)|$ among all degree-d polynomials P with leading coefficient one. It reaches its maximum absolute magnitude 2^{1-d} at $d + 1$ points. An affine change of variables gives the corresponding construction on any interval $[a, b]$. To relate the algorithms discovered by AUTOSPEC to classical methods, we investigate whether the generated polynomials exhibit these properties. For each polynomial, we optimize an affine transformation of the spectral interval and compare its peak magnitude, normalized by the absolute leading coefficient, with the corresponding minimax bound (Appendix J). Across the four eigenproblems in Table 10, the normalized maximum norms range from 1.69×10^{-6} to 1.74×10^{-6}. These values are roughly six orders of magnitude below the random polynomial baseline (0.66) and within a factor of 2 of the theoretical Chebyshev minimum (0.95×10^{-6}).

For linear systems, the Chebyshev preconditioner minimizes the residual $\max_{\lambda\in[a,b]} |r(\lambda)|$, where $[a, b]$ is the spectral interval and yields equioscillation behavior. Figure 4a shows that residual polynomials generated by AUTOSPEC without $\mathcal{L}_{\text{struct}}$ exhibit similar behavior after affine transformation.



(a) Post-fitted residual polynomials. (b) Eigenvalues $P(\lambda)\lambda$ of the preconditioned Laplacian

Figure 4: (a) Residual polynomials $r(\lambda) = 1 - \lambda P(\lambda)$ induced by AUTOSPEC trained without structural objective $\mathcal{L}_{\text{struct}}$ can be post-fitted by an affine shift/scale $C(\lambda) = a\, r(b\,\lambda + c) + d$ to resemble Chebyshev polynomials. (b) Incorporating $\mathcal{L}_{\text{struct}}$ allows AUTOSPEC to depart from standard minimax constructions and generate $P(\cdot)$ avoiding clusters of small eigenvalues in $P(\mathbf{X})\mathbf{X}$.

6.2 Beyond Classical Constructions

For a prescribed polynomial degree and spectral interval, Chebyshev preconditioning for linear systems provides an optimal worst-case condition number bound. However, it can introduce clusters of small eigenvalues that can adversely affect CG convergence.

AUTOSPEC can alleviate this shortcoming by optimizing objectives beyond classical inverse approximation. By incorporating $\mathcal{L}_{\text{struct}}$ in the training objective (9), we aim to keep the condition number of the preconditioner $P(\mathbf{X})$ close to the natural lower bound imposed by the desired conditioning of $P(\mathbf{X})\mathbf{X}$, namely $\text{cond}(P(\mathbf{X})) \geq \text{cond}(\mathbf{X})/\text{cond}(P(\mathbf{X})\mathbf{X})$. The structural term balances these competing objectives, allowing a modest reduction in the conditioning improvement of $P(\mathbf{X})\mathbf{X}$ in exchange for better conditioning of $P(\mathbf{X})$ itself. Figure 4b illustrates the resulting spectral shapes for degree-11 and degree-16 polynomials on a finite-difference Laplacian on the unit square. The learned polynomials suppress the repeated low troughs of the Chebyshev mapping $\lambda P(\lambda)$ and reduce the accumulation of eigenvalues near its lower extrema. This illustrates how a task-specific objective can produce spectral transformations beyond those prescribed by minimax inverse approximation.

6.3 Comparison with State-of-the-Art Polynomial Acceleration

Chebyshev Preconditioning. Table 1 compares degree-11 AUTOSPEC and Chebyshev accelerators using spectral estimates obtained from the same 50- or 200-step Lanczos probes. AUTOSPEC achieves Chebyshev-level acceleration and substantially reduces iteration counts on several problems, while exhibiting greater robustness to changes in the probe. The results in Table 1 use Jacobi as the first-level preconditioner. Appendix G.7 also compares accelerators on top of AMG and incomplete Cholesky preconditioning.

Comparison with Direct Polynomial Fitting. We compare AUTOSPEC with least-squares and minimax fitting using the same Ritz values (Appendix G.8). AUTOSPEC generally requires fewer CG iterations than either fitting method and exhibits lower performance variance than minimax fitting. The AUTOSPEC construction thus improves on direct inverse fitting while producing recurrence coefficients in a single inference pass, without solving a new optimization problem for each operator.

Runtime. Table 1 reports total runtimes on realistic test matrices using MATLAB's built-in `pcg` solver and an external multithreaded C++ MEX implementation for both AUTOSPEC and Chebyshev polynomial application. The timings include preconditioning setup and the iterative solve. Across all ten matrices, AUTOSPEC reduces total runtime by a factor of 1.5–2.7 relative to unaccelerated CG, with runtimes comparable to or lower (often substantially) than those of Chebyshev acceleration. Appendix G.9 details the timing protocol.

6.4 Ablation Study

Polynomial Degree and Model Architecture. In Appendix H.1, we study the training pipeline of AUTOSPEC engine under various polynomial degrees and different structures of the backbone and embedding layer. We show that the same training recipe can be used to train AUTOSPEC engine

Table 1: pcg iteration counts and total runtimes for degree-11 polynomial preconditioning. Iteration counts use 50- or 200-step Lanczos probes. Runtimes are provided for 200-step probes and include the polynomial construction cost. We omit Dubcova2 in runtime comparisons as it is too small. Evaluation protocols are given in Appendix G.9.

	AUTOSPEC iterations		Chebyshev iterations		Runtime (s)		
Matrix	200	50	200	50	No poly.	AUTOSPEC	Chebyshev
Dubcova2	36 ± 0	35 ± 0	67 ± 1	58 ± 21	—	—	—
thermal1	154 ± 2	162 ± 2	265 ± 50	145 ± 1	1.87	0.95	1.30
G2_circuit	205 ± 2	219 ± 2	301 ± 40	228 ± 56	2.35	1.24	1.59
shipsec5	541 ± 18	554 ± 4	540 ± 20	518 ± 15	23.50	10.65	10.54
thermal_cookie	312 ± 6	338 ± 1	653 ± 299	684 ± 331	5.40	2.48	4.33
laplacian_medium	195 ± 1	207 ± 1	257 ± 50	407 ± 91	5.07	3.07	3.70
parabolic_fem	318 ± 3	338 ± 3	394 ± 163	445 ± 207	13.20	8.55	10.21
Fault_639	794 ± 9	874 ± 20	752 ± 10	992 ± 12	183.90	103.73	98.57
thermal2	592 ± 5	626 ± 5	558 ± 3	557 ± 4	144.53	53.37	51.63
laplacian_large	384 ± 2	409 ± 2	391 ± 27	618 ± 322	53.05	30.26	30.88
G3_circuit	402 ± 9	438 ± 6	918 ± 430	389 ± 5	70.94	36.04	75.66

with various degrees. For the AUTOSPEC engine architecture, we compare our design with simple variants, and show that similar or better performance can be achieved with more expressive layer structures, but at the cost of higher training cost. These comparisons support the selected architecture for the evaluated tasks, and motivate the directions to explore better AUTOSPEC engine architectures.

Sensitivity to Spectral Coverage and Spectral Probe Accuracy. In Appendix H.2, we study AUTOSPEC training with narrow and broad spectral profile coverage. We found that incorporating richer spectral profile coverage in training can improve the quality of algorithms generated by AUTOSPEC. In addition, in Appendix H.3 we show that AUTOSPEC engine is able to generate algorithms robustly under perturbation to input spectral probes.

Role of Spectral Residual Features in AUTOSPEC Algorithm Generation. In Appendix H.4, we study the role of components in the input spectral probe. We compare the algorithm generated by AUTOSPEC with and without the residual spectral features, and show that the residual features provide essential spectral-quality signals for the model to generate effective preconditioning algorithms.

6.5 Limitations

The current evaluation focuses on spd problems, where strong classical methods, including Chebyshev preconditioners, already exist. Extending AUTOSPEC to indefinite and unsymmetric problems is a natural direction for future work. AUTOSPEC requires a rich synthetic training set whose spectra contain the key features of the matrices encountered at test time. For example, in eigenvalue problems, if the model is trained only on spectra with larger eigengaps than those encountered at test time, performance can degrade (see Appendix H.2). Furthermore, for spd preconditioning, the generated polynomial must satisfy $P(\mathbf{X}) > 0$. We observed violation of this property on $< 0.01\%$ of the synthetic training spectra and none on the realistic test cases.

We also note that AUTOSPEC can support richer update classes beyond the linear recurrences studied in the main text, including higher-order polynomial recurrences and nonlinear transition maps that preserve the iterative structure while increasing expressivity; see Appendix B.1 for details. Together, these results point to a broader pathway for using ML methodology to learn robust, deployable NLA and numerical optimization algorithms that generalize across operator instances.

7 Conclusion

We introduced AUTOSPEC, a neural network framework for discovering spectrum-adaptive numerical iterative algorithms by training a neural engine to output the coefficients of an executable recurrence. Across multiple NLA tasks, the resulting learned recurrences consistently improve convergence and/or accuracy over spectrum-agnostic baselines, demonstrating the effectiveness of the algorithm discovery.

Acknowledgments and Disclosure of Funding

The authors would like to acknowledge the NSF and the DARPA DIAL and DARPA AIQ programs for partial support of this work. Yaoqing Yang would like to additionally acknowledge NSF-DMS2616827 for the support of this work.

References

[1] Kwangjun Ahn, Byron Xu, Natalie Abreu, Ying Fan, Gagik Magakyan, Pratyusha Sharma, Zheng Zhan, and John Langford. Dion: Distributed orthonormalized updates. *arXiv preprint arXiv:2504.05295*, 2025.

[2] Noah Amsel, David Persson, Christopher Musco, and Robert M. Gower. The Polar Express: Optimal matrix sign methods and their application to the Muon algorithm. *arXiv preprint arXiv:2505.16932*, 2025. URL `https://arxiv.org/abs/2505.16932`.

[3] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

[4] Steven F Ashby. Minimax polynomial preconditioning for Hermitian linear systems. *SIAM Journal on Matrix Analysis and Applications*, 12(4):766–789, 1991.

[5] Steven F Ashby, Thomas A Manteuffel, and James S Otto. A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):1–29, 1992.

[6] Owe Axelsson. *Iterative solution methods*. Cambridge university press, 1996.

[7] Amartya S. Banerjee, Lin Lin, Wei Hu, Chao Yang, and John E. Pask. Chebyshev polynomial filtered subspace iteration in the discontinuous Galerkin method for large-scale electronic structure calculations. *The Journal of Chemical Physics*, 145(15):154101, 2016.

[8] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[9] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *International Conference on Machine Learning*, pages 459–468. PMLR, 2017.

[10] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[11] Luca Bergamaschi and Angeles Martinez Calomardo. Parallel newton–chebyshev polynomial preconditioners for the conjugate gradient method. *Computational and Mathematical Methods*, 3(6):e1153, 2021.

[12] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, second edition, 2000.

[13] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36:49205–49233, 2023.

[14] Yuri Chervonyi, Trieu H Trinh, Miroslav Olšák, Xiaomeng Yang, Hoang H Nguyen, Marcelo Menegali, Junehyuk Jung, Junsu Kim, Vikas Verma, Quoc V Le, et al. Gold-medalist performance in solving olympiad geometry with alphageometry2. *Journal of Machine Learning Research*, 26(241):1–39, 2025.

[15] C. W. Clenshaw. A note on the summation of chebyshev series. *Mathematical Tables and Other Aids to Computation*, 9(51):118–120, 1955. doi: 10.1090/S0025-5718-1955-0071856-0.

[16] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011. doi: 10.1145/2049662.2049663.

[17] Vladimir Druskin and Leonid Knizhnerman. Extended Krylov subspaces: Approximation of the matrix square root and related functions. *SIAM Journal on Matrix Analysis and Applications*, 19(3):755–771, 1998.

[18] Charles B Dunham. Choice of basis for chebyshev approximation. *ACM Transactions on Mathematical Software (TOMS)*, 8(1):21–25, 1982.

[19] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2021.

[20] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, October 2022.

[21] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, 1997.

[22] Ekaterina Grishina, Matvey Smirnov, and Maxim Rakhuba. Accelerating Newton–Schulz iteration for orthogonalization via Chebyshev-type polynomials. *arXiv preprint arXiv:2506.10935*, 2025.

[23] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[24] Stefan Güttel. Rational Krylov approximation of matrix functions: Numerical methods and optimal pole selection. *GAMM-Mitteilungen*, 36(1):8–31, 2013.

[25] Paul Häusner, Ozan Öktem, and Jens Sjölund. Neural incomplete factorization: learning preconditioners for the conjugate gradient method. *arXiv preprint arXiv:2305.16368*, 2023.

[26] Conor F Hayes, Felipe Leno Da Silva, Jiachen Yang, T Nathan Mundhenk, Chak Shing Lee, Jacob F Pettit, Claudio Santiago, Sookyung Kim, Joanne T Kim, Ignacio Aravena Solis, et al. Deep symbolic optimization: Reinforcement learning for symbolic mathematics. *arXiv preprint arXiv:2505.10762*, 2025.

[27] Magnus R Hestenes, Eduard Stiefel, et al. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49(6):409–436, 1952.

[28] Nicholas J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, 2008.

[29] Lei Huang, Dawei Yang, Bo Lang, and Jia Deng. Decorrelated batch normalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 791–800, 2018.

[30] Olin G. Johnson, Charles A. Micchelli, and George Paul. Polynomial preconditioners for conjugate gradient calculations. *SIAM Journal on Numerical Analysis*, 20(2):362–376, 1983.

[31] Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. *Cited on*, page 10, 2024. URL `https://kellerjordan.github.io/posts/muon/`.

[32] Ayano Kaneda, Osman Akar, Jingyu Chen, Victoria Alicia Trevino Kala, David Hyde, and Joseph Teran. A deep conjugate direction method for iteratively solving linear systems. In *International Conference on Machine Learning*, pages 15720–15736. PMLR, 2023.

[33] Samuel Kim, Peter Y Lu, Srijon Mukherjee, Michael Gilbert, Li Jing, Vladimir Čeperić, and Marin Soljačić. Integration of neural network-based symbolic regression in deep learning for scientific discovery. *IEEE transactions on neural networks and learning systems*, 32(9): 4166–4177, 2020.

[34] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019. doi: 10.21105/joss.01244.

[35] John R. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, June 1994.

[36] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[37] Bar Lerer, Ido Ben-Yair, and Eran Treister. Multigrid-augmented deep learning preconditioners for the helmholtz equation using compact implicit layers. *SIAM Journal on Scientific Computing*, 46(5):S123–S144, 2024.

[38] Jiwei Li, Alexander H Miller, Sumit Chopra, Marc'Aurelio Ranzato, and Jason Weston. Learning through dialogue interactions by asking questions. *arXiv preprint arXiv:1612.04936*, 2016.

[39] Yichen Li, Peter Yichen Chen, Tao Du, and Wojciech Matusik. Learning preconditioners for conjugate gradient PDE solvers. In *International Conference on Machine Learning*, pages 19425–19439. PMLR, 2023.

[40] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. Universal style transfer via feature transforms. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[41] Jörg Liesen and Zdenek Strakos. *Krylov subspace methods: principles and analysis*. Numerical Mathematics and Scientific Computation, 2013.

[42] Jian Luo, Jie Wang, Hong Wang, huanshuo dong, Zijie Geng, Hanzhu Chen, and Yufei Kuang. Neural Krylov iteration for accelerating linear system solving. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[43] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[44] Thomas A Manteuffel. The tchebychev iteration for nonsymmetric linear systems. *Numerische Mathematik*, 28(3):307–327, 1977.

[45] Terrell Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P Santiago, Brenden K Petersen, et al. Symbolic regression via deep reinforcement learning enhanced genetic programming seeding. *Advances in Neural Information Processing Systems*, 34:24912–24923, 2021.

[46] Yuji Nakatsukasa, Olivier Sète, and Lloyd N. Trefethen. The AAA algorithm for rational approximation. *SIAM Journal on Scientific Computing*, 40(3):A1494–A1522, 2018.

[47] Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*, volume 87 of *Applied Optimization*. Springer, New York, NY, 2004.

[48] Yurii E. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. *Soviet Mathematics Doklady*, 27(2):372–376, 1983.

[49] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[50] Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33:1060–1070, 2020.

[51] Junhyuk Oh, Gregory Farquhar, Iurii Kemaev, Dan A. Calian, Matteo Hessel, Luisa Zintgraf, Satinder Singh, Hado van Hasselt, and David Silver. Discovering state-of-the-art reinforcement learning algorithms. *Nature*, 648(8093):312–319, December 2025.

[52] Brenden K Petersen, Mikel Landajuela, T Nathan Mundhenk, Claudio P Santiago, Soo K Kim, and Joanne T Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. *arXiv preprint arXiv:1912.04871*, 2019.

[53] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[54] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International conference on learning representations*, 2017.

[55] Esteban Real, Chen Liang, David So, and Quoc V. Le. AutoML-Zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pages 8007–8019. PMLR, 2020.

[56] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024.

[57] Alexander Rudikov, Vladimir Fanaskov, Ekaterina Muravleva, Yuri M Laevsky, and Ivan Oseledets. Neural operators meet conjugate gradients: The fcg-no method for efficient pde solving. *arXiv preprint arXiv:2402.05598*, 2024.

[58] Youcef Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881, 1985.

[59] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7 (3):856–869, 1986.

[60] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[61] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. Society for Industrial and Applied Mathematics, second edition, 2011.

[62] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

[63] Yue Song, Nicu Sebe, and Wei Wang. Why approximate matrix square root outperforms accurate svd in global covariance pooling? In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE Computer Society, 2021.

[64] Yue Song, Nicu Sebe, and Wei Wang. Fast differentiable matrix square root and inverse square root. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 45(6): 7367–7380, 2023.

[65] Satoshi Takabe and Tadashi Wadayama. Convergence acceleration via Chebyshev step: Plausible interpretation of deep-unfolded gradient descent. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 105(8):1110–1120, 2022.

[66] 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. *Computational Methods in Applied Mathematics*, (0), 2026.

[67] Ulrich Trottenberg, Cornelius W. Oosterlee, and Anton Schüller. *Multigrid*. Academic Press, 2000.

[68] Silviu-Marian Udrescu and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020.

[69] Richard S. Varga. *Matrix Iterative Analysis*, volume 27 of *Springer Series in Computational Mathematics*. Springer, second revised and expanded edition, 2000.

[70] Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. In *International conference on machine learning*, pages 3751–3760. PMLR, 2017.

[71] S. Yang, Z. Wang, O. Balabanov, N. B. Erichson, and M. W. Mahoney. PRISM: Distribution-free adaptive computation of matrix functions for accelerating neural network training. *arXiv preprint arXiv:2601.22137*, 2026.

[72] David M Young. *Iterative solution of large linear systems*. Elsevier, 2014.

[73] Yunkai Zhou and Yousef Saad. A Chebyshev–Davidson algorithm for large symmetric eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 29(3):954–971, 2007.

Appendix

A Related Work in Numerical Linear Algebra

Spectral Approximation and Preconditioning Methods in NLA. Polynomial accelerations and approximations are among the earliest and most widely used NLA tools for iterative solution of large, sparse linear systems and related tasks. A common strategy is to build low-degree polynomial transforms guided by coarse spectral bounds, that damp slowly converging error components and accelerate convergence [6, 60, 69, 72]. Krylov subspace methods provide a complementary paradigm and can be viewed as constructing analogous polynomial filters implicitly and adaptively, by optimizing residuals over progressively enriched subspaces [21, 27, 41, 59, 61]. On modern architectures, Krylov performance is increasingly constrained by communication/synchronization (e.g., global reductions), renewing interest in polynomial preconditioners and filtering [7, 11, 61, 73]. Extensive work develops polynomial preconditioners and smoothers via truncated series, minimax, or least-squares designs, as matrix-free surrogates for inverses or spectrum-shaping operators, used both standalone and within multilevel solvers [4, 5, 10, 12, 30, 67]. Beyond first-order methods, second-order matrix iterations such as Newton methods yield rapidly convergent approximations for inverses, inverse roots, and matrix sign functions, with kernels dominated by matrix-matrix products that map well to modern parallel hardware [28]. These loops now appear in large-scale optimizers and training systems (e.g., Shampoo; Muon; and variants such as PolarExpress) [1, 2, 22, 23, 31]. Related iterative matrix (inverse) square-root primitives also underpin second-order vision layers and whitening/decorrelated normalization [29, 40, 63, 64]. However, these second-order iterations remain far less applicable in sparse and matrix-free regimes, where matvec-based kernels are essential. Finally, when shifted solves (or approximate shift-and-invert actions) are available, rational approximation extends these designs via rational filters, rational Krylov/subspace methods, and adaptive rational fitting such as AAA [17, 24, 28, 46].

B Details of the AUTOSPEC Learning Framework

B.1 Extensions

We outline extensions of the linear state-transition model used in the AUTOSPEC framework in main text.

Higher-order updates. Higher-order updates can be realized by allowing nonlinear dependence on the state $\mathbf{V}_k$. For example, a quadratic transition can be defined as

$$M_k(\mathbf{V}_k, \mathbf{X}) := \mathbf{I}_k(\mathbf{X})\, \mathbf{V}_k\, \mathbf{J}_k(\mathbf{X})\, \mathbf{V}_k^{\mathsf{T}}\, \mathbf{K}_k(\mathbf{X}), \tag{10}$$

where matrices $\mathbf{I}_k(\mathbf{X}), \mathbf{J}_k(\mathbf{X}), \mathbf{K}_k(\mathbf{X})$ admit block decompositions with subblocks given by low-degree polynomials in $\mathbf{X}$. Such higher-order updates are less suited to sparse, matrix-free regimes than linear transitions, since they generally require explicit matrix–matrix products. However, on parallel hardware, especially in optimization loops that already use explicit matrix-function approximations (e.g., inverse-root preconditioning in Shampoo and orthogonalization/polar updates in Muon), they can be highly attractive [1, 2, 22, 23, 28, 31]. Discovering and training such higher-order recurrences for optimization is a promising direction for future work.

Rational Models. In some settings, such as approximating matrix functions with sharp variation near the spectrum (e.g., $\mathbf{X}^{-1/2}$ or $\log(\mathbf{X})$), or computing interior eigenvalues, it can be advantageous to model $P(\cdot)$ as a *rational* function rather than a polynomial function. Methods based on rational spectral transformations play a central role in NLA and are particularly prominent in solvers based on Krylov approximation [17, 24, 28, 46]. Within the AUTOSPEC framework, rational constructions can be incorporated naturally by allowing the transition blocks $M_k^{(i,j)}(\cdot)$ to be rational functions. For example, in the three-term update formula (4), the linear shifts may be replaced by resolvents,

$$\alpha_k \mathbf{I} + \beta_k \mathbf{X} \to (\alpha_k \mathbf{I} + \beta_k \mathbf{X})^{-1}, \text{ for } 1 \le k \le d.$$

This substitution trades inexpensive matvecs with $\mathbf{X}$ for more costly shifted linear solves during inference, but it can yield substantially improved approximations and preconditioners.

Gradient-driven transitions. The same state-transition view can also express first-order optimization recurrences by treating the operator in (2) as a nonlinear map accessed through gradient evaluations, i.e., $\mathbf{X}(\mathbf{z}) = \nabla f(\mathbf{z})$.[3] For instance, with an augmented state $\mathbf{V}_k = (\boldsymbol{\theta}_k, \mathbf{m}_k)$, the transition recovers gradient descent with momentum,

$$\mathbf{m}_{k+1} = \beta_k \mathbf{m}_k + \mathbf{X}(\boldsymbol{\theta}_k), \qquad \boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \mathbf{m}_{k+1},$$

and yields Nesterov acceleration by evaluating the gradient at an extrapolated point (e.g., $\mathbf{X}(\boldsymbol{\theta}_k + \gamma_k \mathbf{m}_k)$) [47, 48, 53]. This suggests learning *gradient-driven* neural engines that map cheap probe features (e.g., Hessian spectral estimates, gradient norms/inner products, or curvature sketches) to coefficient sequences $(\alpha_k, \beta_k, \gamma_k, \ldots)$, thereby discovering accelerated or task-adaptive optimizers within universal recurrence formalism.

B.2 Learning Objectives for Different Tasks

In (8), we introduced the generalized objective $\rho_{\log}$, which compares a learned method against a task-specific baseline through a scalar performance metric $r(\mathbf{X})$. Concretely, each application requires the following: (i) a metric r that quantifies convergence rate/accuracy; and (ii) a baseline method (without learned acceleration) used to form r_{baseline}. For the tasks in this paper, we used the following.

- **Eigenvalue problems.** We measure separation after applying the polynomial spectral transform $P(\cdot)$ via

$$r(\mathbf{X}) = \frac{\min_{i=1,\ldots,k} \left|P(\lambda_i(\mathbf{X}))\right|}{\max_{I \subset \{1,\ldots,m\},\, |I|=l} \; \min_{j \in I} \left|P(\lambda_j(\mathbf{X}))\right|}, \tag{11}$$

  where $\{\lambda_i(\mathbf{X})\}$ denotes the spectrum of $\mathbf{X}$, k is the number of requested eigenpairs, and l is the dimension of Ritz approximation subspaces. As a baseline we use standard power/subspace iteration polynomial $P(\lambda_j(\mathbf{X})) = \lambda_j(\mathbf{X})^d$.

- **Linear systems.** We use the operator-norm residual of the normalized preconditioned operator,

$$r(\mathbf{X}) = \left\| \mathbf{I} - \frac{P(\mathbf{X})\mathbf{X}}{\|P(\mathbf{X})\mathbf{X}\|_2} \right\|_2, \tag{12}$$

  with Richardson iteration polynomial as the baseline.

- **Inverse square root.** For approximating $\mathbf{X}^{-1/2}$, we use the worst-case relative error in operator norm,

$$r(\mathbf{X}) = \left\| \mathbf{I} - P(\mathbf{X})\, \mathbf{X}^{1/2} \right\|_2, \tag{13}$$

  with a Neumann (truncated Taylor expansion) polynomial as the baseline.

C Details of Neural Network Engine

We provide the details of the neural network engine, including the backbone model and embedding layer, and specifications for different NLA tasks.

C.1 Backbone Model

Corresponding to Section 4.1, the detailed structure of each layer of the backbone model is shown in Algorithm 1.

C.2 Embedding Layer

[3] Here we overload $\mathbf{X}$ to denote an operator/oracle rather than a matrix.

Algorithm 1 NN Layer k of Backbone Model f_{θ_k}

Parameters: Weight matrices $\mathbf{W}_k \in \mathbb{R}^{d\times 5}, \mathbf{w}_k \in \mathbb{R}^d$, Bias $\mathbf{b}_k \in \mathbb{R}^5, b_k \in \mathbb{R}$, $\epsilon = 10^{-8}$
Input: Input to neural network layer $\mathbf{e} \in \mathbb{R}^d$, matrix $\mathbf{X}$.
Output: State transition operators $\mathbf{M}_k(\mathbf{X})$

$[\rho_k, \gamma_k, \eta_k, \alpha_k, \beta_k]^{\mathrm{T}} \leftarrow \mathbf{W}_k^{\mathrm{T}}\mathbf{e} + \mathbf{b}_k$ ▷ Coefficients

$\delta_k \leftarrow \mathbf{w}_k^{\mathrm{T}}\mathbf{e} + b_k$ ▷ Learned Scaling Factor

$\gamma_k, \eta_k, \alpha_k, \beta_k \leftarrow \frac{\gamma_k}{\delta_k+\epsilon}, \frac{\eta_k}{\delta_k+\epsilon}, \frac{\alpha_k}{\delta_k+\epsilon}, \frac{\beta_k}{\delta_k+\epsilon}$

if $1 \le k < d$ **then** define $\mathbf{M}_k$ as

$$\big(\mathbf{x}, \mathbf{y}, \mathbf{z}\big) \mapsto \big(\mathbf{x} + \rho_k\mathbf{z},\ \mathbf{z},\ \gamma_k\mathbf{x} + \eta_k\mathbf{y} + \alpha_k\mathbf{z} + \beta_k\mathbf{X}\mathbf{z}\big)$$

else if $k = d$ **then** define $\mathbf{M}_k$ as

$$\big(\mathbf{x}, \mathbf{y}, \mathbf{z}\big) \mapsto \gamma_k\mathbf{x} + \eta_k\mathbf{y} + \alpha_k\mathbf{z} + \beta_k\mathbf{X}\mathbf{z}$$

end if

return $\mathbf{M}_k$

Embedding Layer for Eigenvalue Problems. To compute the largest or smallest k eigenvalues of $\mathbf{X}$ using iterative eigensolvers such as Krylov-Schur with subspace dimension l, an effective preconditioning algorithm should enlarge the spectral gap between the k-th and l-th eigenvalues. Varying k or l changes the target portion of the spectrum, requiring the model to generalize across different target spectral regions. We design an embedding layer that allows the model to adapt to different k and l and construct effective preconditioners, requiring only truncation and padding to the input spectral probes.

Algorithm 2 shows the structure of the embedding layer for eigenvalue problems. It uses two multi-layer subnetworks to exploit spectral information from the input spectral probe, and it returns the spectral embedding for the k-th and l-th eigenvalue, respectively. The first subnetwork, parameterized by $\mathbf{W}_1, \mathbf{W}_3$, first obtains $\mathbf{y}_1$ that encodes the spectral information of top-k eigenvalues, and then selects the smallest magnitude y_1 as the embedding for the k-th eigenvalue. The second subnetwork, parameterized by $\mathbf{W}_2, \mathbf{W}_4$, outputs an embedding $\mathbf{y}_2$ encoding spectral information in the neighborhood of the l-th eigenvalue. Then the two embeddings are concatenated to form the final output embedding. The layer has a fixed input size of $2l_0$, corresponding to l approximated eigenvalues and corresponding residuals.

In practice, the target rank k' and subspace dimension l' may differ from the fixed input window size (k_0, l_0) expected by the embedding layer. To produce a fixed-length input of l_0 spectral probes (eigenvalue estimates and residuals), we construct a normalized window by truncation/padding while preserving the two boundary indices. Specifically, we first form the leading block of k_0 probes from the top-k' approximate eigenvalues by truncating or padding as needed, and we enforce that the k_0-th probe corresponds exactly to the k'-th approximate eigenvalue. We then form the remaining $l_0 - k_0$ probes from the range $\{\widehat{\lambda}_{k'}, \ldots, \widehat{\lambda}_{l'}\}$, again truncating or padding to length $l_0 - k_0$, and we enforce that the final (l_0-th) probe corresponds exactly to the l'-th approximate eigenvalue. This construction ensures that, for any (k', l'), the network input always contains a consistent spectral window anchored at the k'-th and l'-th eigenvalues.

Embedding Layer for Preconditioned Linear Systems. The learning objective of preconditioning algorithms for linear systems it to reduce the condition number of operators. Therefore, the task-relevant portion of the spectrum is the two ends of the spectrum, and the spectral probe consists of the largest and smallest k eigenvalues (with corresponding residuals). In our experiments, we choose $k = 20$. Algorithm 3 describe the procedure of the embedding layer for preconditioned linear systems. We use two subnetworks to process the spectral probe for the top and bottom eigenspectrum separately, and we concatenate their outputs to form the embedding. Each subnetwork uses the spectral probe of corresponding spectral regions and exploit the structure of the eigenspectrum.

Algorithm 2 Embedding Layer g_ϕ for Eigenvalue Problems

Configurations: Number of eigenvalues to compute k_0 with subspace dimension l_0, Model input dimension $d_{\text{in}} = 2l_0$, Hidden dimension $d_{\text{hid}} = 4k_0 + 8$
Parameters: $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d_{\text{in}} \times d_{\text{hid}}}, \mathbf{W}_3 \in \mathbb{R}^{d_{\text{hid}} \times k_0}, \mathbf{W}_4 \in \mathbb{R}^{d_{\text{hid}} \times 4}$, GeLU activation function σ
Input: Approximated eigenvalues $\widehat{\boldsymbol{\lambda}} = \left[\widehat{\lambda}_1, \ldots, \widehat{\lambda}_{l_0}\right]^{\mathrm{T}}$, Residuals $\widehat{\boldsymbol{r}} = [\widehat{r}_1, \ldots, \widehat{r}_{l_0}]^{\mathrm{T}}$
Output: Embedding $\mathbf{e} = [e_1, \ldots, e_5]^{\mathrm{T}} \in \mathbb{R}^5$

$\mathbf{x} \leftarrow \left[\widehat{\boldsymbol{\lambda}}^{\mathrm{T}}, \widehat{\boldsymbol{r}}^{\mathrm{T}}\right]^{\mathrm{T}}$ ▷ Construct input $\mathbf{x} \in \mathbb{R}^{2l_0}$
$\mathbf{y}_1 \leftarrow \mathbf{W}_3^{\mathrm{T}} \sigma(\mathbf{W}_1^{\mathrm{T}} \mathbf{x})$ ▷ First subnetwork, $\mathbf{y}_1 \in \mathbb{R}^{k_0}$
$y_1 = \min_{1 \le i \le n} |(\mathbf{y}_1)_i|$
$\mathbf{y}_2 \leftarrow \mathbf{W}_4^{\mathrm{T}} \sigma(\mathbf{W}_2^{\mathrm{T}} \mathbf{x})$ ▷ Second subnetwork, $\mathbf{y}_2 \in \mathbb{R}^4$
$\mathbf{e} \leftarrow \left[y_1, \mathbf{y}_2^{\mathrm{T}}\right]^{\mathrm{T}}$
return $\mathbf{e}$

Algorithm 3 Embedding Layer g_ϕ for Preconditioned Linear Systems

Configurations: Number of largest/smallest input eigenvalue estimations k, Model input dimension $d_{\text{in}} = 2k$, Hidden dimension $d_{\text{hid}} = 2k$
Parameters: $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d_{\text{in}} \times d_{\text{hid}}}, \mathbf{W}_3, \mathbf{W}_4 \in \mathbb{R}^{d_{\text{hid}} \times 5}$, GeLU activation function σ
Input: Approximated largest eigenvalues $\widehat{\boldsymbol{\lambda}}_{\text{max}} = \left[\widehat{\lambda}_1, \ldots, \widehat{\lambda}_k\right]^{\mathrm{T}}$, Residuals $\widehat{\boldsymbol{r}}_{\text{max}} = [\widehat{r}_1, \ldots, \widehat{r}_k]^{\mathrm{T}}$, Approximated smallest eigenvalues $\widehat{\boldsymbol{\lambda}}_{\text{min}} = \left[\widehat{\lambda}_1^{\text{min}}, \ldots, \widehat{\lambda}_k^{\text{min}}\right]^{\mathrm{T}}$, Residuals $\widehat{\boldsymbol{r}}_{\text{min}} = \left[\widehat{r}_1^{\text{min}}, \ldots, \widehat{r}_k^{\text{min}}\right]^{\mathrm{T}}$
Output: Embedding $\mathbf{e} = [e_1, \ldots, e_{10}]^{\mathrm{T}} \in \mathbb{R}^{10}$

$\mathbf{x}_1, \mathbf{x}_2 \leftarrow \left[\widehat{\boldsymbol{\lambda}}_{\text{max}}^{\mathrm{T}}, \widehat{\boldsymbol{r}}_{\text{max}}^{\mathrm{T}}\right]^{\mathrm{T}}, \left[\widehat{\boldsymbol{\lambda}}_{\text{min}}^{\mathrm{T}}, \widehat{\boldsymbol{r}}_{\text{min}}^{\mathrm{T}}\right]^{\mathrm{T}}$ ▷ Construct input $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^{2k}$
$\mathbf{y}_1 \leftarrow \mathbf{W}_3^{\mathrm{T}} \sigma(\mathbf{W}_1^{\mathrm{T}} \mathbf{x}_1)$ ▷ First subnetwork, $\mathbf{y}_1 \in \mathbb{R}^5$
$\mathbf{y}_2 \leftarrow \mathbf{W}_4^{\mathrm{T}} \sigma(\mathbf{W}_2^{\mathrm{T}} \mathbf{x}_2)$ ▷ Second subnetwork, $\mathbf{y}_2 \in \mathbb{R}^5$
$\mathbf{y}_1 \leftarrow \text{sort}_{\downarrow}(|\mathbf{y}_1|)$ ▷ Sort by magnitude (descending)
$\mathbf{y}_2 \leftarrow \text{sort}_{\downarrow}(|\mathbf{y}_2|)$ ▷ Sort by magnitude (descending)
$\mathbf{e} \leftarrow \left[\mathbf{y}_1^{\mathrm{T}}, \mathbf{y}_2^{\mathrm{T}}\right]^{\mathrm{T}}$
return $\mathbf{e}$

Embedding Layer for Approximating Matrix Functions. For approximating matrix function such as inverse square root, we adopt the same embedding layer as for linear systems, as the top and bottom region of the eigenspectrum are also task-relevant. In our experiments, we choose $k = 20$.

D AUTOSPEC Training Configurations

D.1 Computer Resources

All stages of the training of the neural network engine were performed on one NVIDIA A100 GPU with 40GB of memory.

D.2 Training Data Curation

As shown in Section 4.2, we can train the model with only diagonal matrices, represented as vectors of eigenspectra. We construct a synthetic eigenspectrum generator that generates positive eigenvalue spectra by sampling a small set of continuous shape parameters using quasi-random Sobol sequences, ensuring broad and uniform coverage of admissible spectral configurations. Each spectrum is formed

as a normalized blend of flat, exponential, and power-law decay profiles, with additional concavity modulation to control the condition number and induce slow spectral decay. To better reflect non-idealized operators, we introduce mild local irregularities through multiplicative noise and randomized tail perturbations, while preserving the overall spectral structure.

For specific NLA tasks, the eigenspectrum generator also explicitly controls the spectral features most relevant to the task. For eigenvalue problems, since in practice polynomial preconditioners are most effective for matrices with clustered eigenvalues (slow decay), we apply rejection criteria that enforce unit normalization and small leading eigenvalue separation (e.g., λ_2/λ_1 close to one), yielding a curated set of eigenspectra. For preconditioned linear systems and matrix function approximation, since the condition numbers of operators are most critical for preconditioning and approximation, we explicitly control the condition number of the generated eigenspectra, and create a training set that covers a wide range of condition numbers.

Training Data Normalization. To maintain training stability while ensuring robustness of the learned model, we preprocess the training matrices with normalization. Specifically, we ensure that the largest eigenvalue of the matrix is bounded and around 1. During data generation, we first enforce all synthetic matrices to have unit spectral norm, and we then perform a perturbation scaling for each matrix by either 1) randomly scaling each matrix with a random scalar in $[1-\epsilon, 1+\epsilon]$ (we choose $\epsilon = 0.2$), or 2) scaling with a coarse approximation of the spectral norm. This ensures that the training matrices have bounded norms but not strictly unit-norm, which could otherwise lead to overfitting to this specific spectral property, as in practice we cannot ensure unit norm for realistic matrices and can only normalize the matrix by approxiate spectral norm.

D.3 Details of Optimization Objectives

In the AUTOSPEC framework, we use the unified optimization loss function defined in (9) of Section 4.2 to train neural networks to discover spectral algorithms. Here we describe the detailed loss function and its design philosophy for the primary applications in our paper.

Eigenvalue Problems. To discover a preconditioning algorithm $P(\cdot)$ for eigenvalue problems, we expect the learned algorithm to have the following properties: 1) it outperforms standard power or subspace iteration methods; and 2) it can be extended to higher degrees, so that for any first k iterations of the algorithm, it can be a degree-k preconditioner outperforming k power/subspace iterations. For 1), we choose the baseline method as standard power/subspace iteration; and for 2), we adopt a layer-wise loss function that optimizes the objective for the first k neural network layers ($k = 1, \ldots, d$). Adopting the formula in (11), we define the objective for our model:

$$r_{\text{nn}}(\mathbf{X}) = \frac{\min_{i=1,\ldots,k} |P(\lambda_i(\mathbf{X}))|}{\max_{I\subset\{1,\ldots,m\},|I|=l} \min_{j\in I} |P(\lambda_j(\mathbf{X}))|}, \tag{14}$$

and for standard subspace iteration:

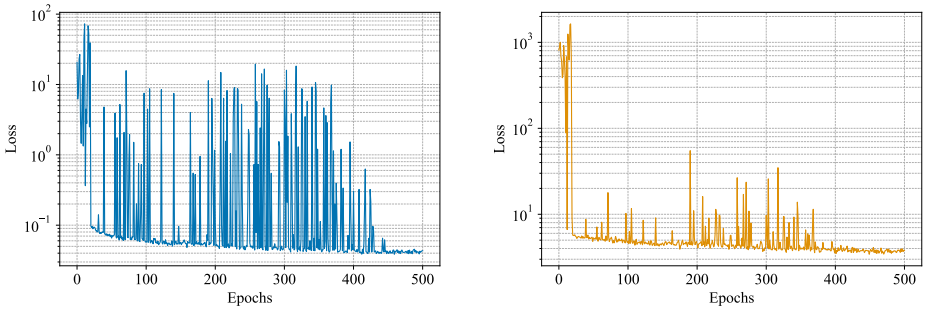
$$r_{\text{subspace}}(\mathbf{X}) = \frac{\min_{i=1,\ldots,k} |\lambda_i(\mathbf{X})|}{\max_{I\subset\{1,\ldots,m\},|I|=l} \min_{j\in I} |\lambda_j(\mathbf{X})|}. \tag{15}$$

Therefore, the loss function $\mathcal{L}$ computes a weighted sum of the loss term $\mathcal{L}_k$ for each layer, with weight w_k for layer k. During training, the weight term w_k is dynamically adjusted to focus on optimizing the loss of different layers. For each layer, the loss follows the form in (9) (we omit the $\mathcal{L}_{\text{struct}}$ as it is not necessary): the inverse of $\rho_{\log}$, and $\mathcal{L}_{\text{reg}}$, which penalizes samples on which the algorithms fail:

$$\mathcal{L} = \sum_{k=1}^{D} w_k \mathcal{L}_k \tag{16}$$

$$\mathcal{L}_k = \frac{1}{N}\sum_{i=1}^{N} \left[\underbrace{\frac{k \cdot \log r_{\text{subspace}}}{\text{clamp}\left(\log r_{\text{nn}}, \epsilon\right)}}_{\mathcal{L}_{\text{obj}}} + \underbrace{10 \exp\left[-10 \cdot r_{\text{nn}} - \epsilon\right]}_{\mathcal{L}_{\text{reg}}} \right]. \tag{17}$$

Note that $\rho_{\log}$ is a ratio of two logarithms, and the denominator is not guaranteed to be positive throughout training. Therefore we clamp the denominator with a small positive scalar. Figure 5 shows the trajectory of two loss terms of the last layer. At the early stage of training, $\mathcal{L}_{\text{reg}}$ dominates the loss, during which the model learns to produce algorithms that perform moderately well on each training sample. Once $\mathcal{L}_{\text{reg}}$ diminishes, $\rho_{\log}$ becomes the primary optimization objective, encouraging the model to find parametrization outperforming the baseline method as much as possible.



(a) Inverse of $\rho_{\log}$ (b) $\mathcal{L}_{\text{reg}}$

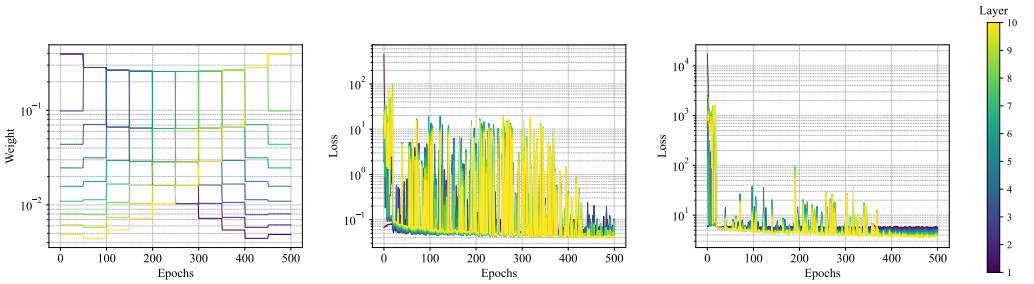
Figure 5: Last-layer training loss of training neural network to discover preconditioning algorithms for eigenvalue problems.

Dynamic Weight Adjustment to Loss Function. At the beginning of the training, w is larger for earlier layers, while towards the end of the training, w is larger for later layers. Suppose the total number of training steps is T, the scheduling function for w_k at step t is:

$$s = \lfloor \frac{T}{t} \rfloor \tag{18}$$

$$w_k^t = \begin{cases} 1 & \text{if } k = s \\ \frac{1}{(k-s)^2} & \text{otherwise} \end{cases}. \tag{19}$$

During training, we normalize the weight by the sum across all layers: $w_k = \frac{w_k}{\sum_{i=1}^{D} w_i}$. We call s the "Anchor Layer," which has the largest weight in the loss function. As training proceeds, the anchor layer moves from the first layer to the last layer. The motivation of this design is to ensure that the polynomial of *any* degree generated by the model can accelerate the convergence when applied to the matrix, instead of only the largest-degree polynomial. This can better disentangle the roles of each layer.



(a) Loss weight of each layer (b) Inverse of $\rho_{\log}$ (c) $\mathcal{L}_{\text{reg}}$

Figure 6: Dynamic adjustment of training loss of all layers. In the early stage of training, earlier layers have a larger weight in the total loss, and are optimized more; towards the end of training, later layers have a larger weight in the total loss, and are optimized more.

Preconditioned Linear Systems. Similarly, we design the optimization objective for generating preconditioning algorithm $P(\cdot)$ for solving linear systems, based on the objective in (12). When training with diagonal matrices $\mathbf{X} = \text{diag}(\boldsymbol{\lambda})$, the residual objective can be reduced to

$$r_{\text{nn}}(\mathbf{X}) = \max_j \left| 1 - \frac{P(\lambda_j(\mathbf{X}))\lambda_j(\mathbf{X})}{\max_i |P(\lambda_i(\mathbf{X}))\lambda_i(\mathbf{X})|} \right|, \tag{20}$$

which is easy to obtain during training. Note that due to the normalization, we have $r_{\text{nn}} \in [0, 2]$. As the baseline, we use the following residual:

$$r_{\text{Richardson}}(\mathbf{X}) = 1 - \frac{\min_i \lambda_i(\mathbf{X})}{\max_i \lambda_i(\mathbf{X})}. \tag{21}$$

This represents the convergence rates of a basic Richardson method for solving linear systems. Following the form in (9) define the optimization objective as

$$\mathcal{L}_k = \frac{1}{N}\sum_{i=1}^{N}\left[\underbrace{\text{clamp}\left(\frac{k \cdot \log r_{\text{Richardson}}(\mathbf{X}_i)}{\log r_{\text{nn}}(\mathbf{X}_i)}, \epsilon\right)}_{\mathcal{L}_{\text{obj}}} + \underbrace{r_{\text{nn}}(\mathbf{X}_i)}_{\mathcal{L}_{\text{reg}}} + \underbrace{c \cdot \frac{\max_i |P(\lambda_i(\mathbf{X}_i)|}{\min_i |P(\lambda_i(\mathbf{X}_i)|}}_{\mathcal{L}_{\text{struct}}}\right] \tag{22}$$

on training dataset $\{\mathbf{X}_i\}_{i=1}^N$, where we set $c = 5 \times d^3$, d is the degree of the constructed polynomial. The term $\mathcal{L}_{\text{struct}}$ is a structural constraint that minimizes the condition number of the preconditioner $P(\mathbf{X})$. Our empirical study on realistic matrices suggests that constraining the condition number of the preconditioner can be useful for improving convergence stability.

In the early stage of training, r_{nn} is unstable and would revolve around 1, causing the $\mathcal{L}_{obj}$ to fluctuate around 5, which is difficult to optimize. Therefore we perform sample-wise clipping of $\mathcal{L}_{\text{obj}}$ from below with a small positive constant (10^{-8}), while optimizing $\mathcal{L}_{\text{reg}}$ and $\mathcal{L}_{\text{struct}}$. As the training proceeds, $\mathcal{L}_{\text{reg}}$ becomes smaller than 1 across the dataset, where $\mathcal{L}_{\text{obj}}$ is activated, becoming the primary optimization objective.

Approximating Matrix Functions. In this work, we consider learning a polynomial $P(\cdot)$ that approximate matrix inverse square root, and we select the Neumann (Taylor) series approximation $T(\cdot)$ as our baseline reference method. Following (13), we define the objective for our model and baseline for training on synthetic diagonal matrices $\mathbf{X} = \text{diag}(\boldsymbol{\lambda})$:

$$r_{\text{nn}}(\mathbf{X}) = \max_i |1 - P(\widetilde{\lambda}_i(\mathbf{X}))\widetilde{\lambda}_i(\mathbf{X})^{\frac{1}{2}}| \tag{23}$$

$$r_{\text{Neumann}}(\mathbf{X}) = \max_i |1 - T(\lambda_i(\mathbf{X}))\lambda_i(\mathbf{X})^{\frac{1}{2}}|, \tag{24}$$

where $\widetilde{\lambda}(\mathbf{X})$ are eigenvalues augmented with random sampled values in $[\lambda_{\text{min}}(\mathbf{X}), \lambda_{\text{max}}(\mathbf{X})]$. This is to regularize the approximation accuracy of $P(\cdot)$ not only on the discrete eigenvalues, but also on the continuous region in which the eigenspectrum is located. This augmentation promotes the learning of a more robust approximation algorithm. Based on the form in (9) (we omit the $\mathcal{L}_{\text{struct}}$ as it is not necessary for matrix function approximation), we design the loss function as

$$\mathcal{L}_k = \frac{1}{N}\sum_{i=1}^{N}\left[\underbrace{\text{clamp}\left(\frac{\log r_{\text{Neumann}}(\mathbf{X}_i)}{\log r_{\text{nn}}(\mathbf{X}_i)}, \epsilon\right)}_{\mathcal{L}_{\text{obj}}} + \underbrace{r_{\text{nn}}(\mathbf{X}_i)}_{\mathcal{L}_{\text{reg}}}\right]. \tag{25}$$

D.4 Extending Polynomial Degree

For eigenvalue problems, problem difficulty strongly correlates with eigenvalue gaps, which requires polynomials of different degree to achieve reliable performance. Our AUTOSPEC framework offers an easy way to extend the backbone model to higher degrees, where one can append more neural network to already pre-trained backbone models, and continue training the model. This works because the optimization objective in (16) promotes the model to learn a solution such that, for any degree k, polynomials generated by the first k layers can sufficiently accelerate the convergence on eigenvalue problems. Figure 15a shows that any degree k polynomial preconditioner from the first k layers consistently has a larger eigenvalue gap than standard iterative methods.

E AUTOSPEC Training Procedures

E.1 Backbone Model (Pre-training)

For pre-training, we follow the optimization objectives described in Appendix D.3 for each NLA task. We perform training using synthetic diagonal matrices (represented as vectors of eigenvalues).

Eigenvalue Problems. For eigenvalue problems, we create a synthetic dataset of size 50000 with matrix dimensions ranging from $[50, 1000]$, which contains matrices that have slow decay in dominant eigenvalues. We use a batch size of 500 and train the model for 1000 epochs. We use the AdamW optimizer with cosine learning rate decay after 10% of warmup steps. We search for the best learning rate among $\{10^{-3}, 5 \times 10^{-4}, 10^{-4}\}$ and select the best learning rate, and use a weight decay of 5×10^{-4}.

Preconditioned Linear Systems. For preconditioner linear systems, we create a synthetic dataset of size 50000 with matrix dimensions ranging from $[50, 1000]$, which contains matrices whose condition number ranges $[10^2, 10^5]$. We use a batch size of 500 and train the model for 1000 epochs. We use the AdamW optimizer with cosine learning rate decay after 10% of warmup steps. We search for the best learning rate among $\{10^{-3}, 5 \times 10^{-4}, 10^{-4}\}$ and select the best learning rate, and use a weight decay of 5×10^{-4}.

Approximating Matrix Functions. For matrix function approximations, we adopt the same training strategy as for preconditioned linear systems.

E.2 Embedding Model (Post-training)

Following the introduction of the structure of the embedding layer in Section C.2, we introduce the training process of the embedding layer. The post-training stage aims to enable the neural network engine to generate effective algorithms when only given spectral probes (coarse spectral estimates). We therefore train an embedding layer that encodes information about structures of task-relevant spectral regions and generates embeddings. In practice, we use the same optimization objective as training the backbone model, and we freeze the backbone model while training the embedding layer.

Eigenvalue Problems For eigenvalue problems, we design the embedding layer as shown in Algorithm 2, which adapts to different target numbers of eigenvalues k and subspace dimensions l via adaptive truncation/padding of the input spectral probe. To enable the model that generalizes to different k and l, we construct a training dataset with a mixture of k, l, and spectral probes obtained with a varying number of subspace iterations.

Specifically, we create a synthetic dataset with matrix dimension ranging from $[200, 2000]$, spectral probes using subspace dimension l in $[20, 100]$, and number of subspace iterations in $[1, 500]$. We train the embedding layer while freezing the backbone model for 500 epochs, with a dataset of size 50000 and a batch size of 200. We use AdamW optimizer with cosine learning rate decay after 10% of warmup steps. We search for the best learning rate among $\{10^{-4}, 10^{-5}, 5 \times 10^{-6}, 10^{-6}\}$ and select the best learning rate, and use a weight decay among the best of $\{10^{-2}, 10^{-3}, 5 \times 10^{-4}\}$.

Preconditioned Linear Systems. For preconditioned linear systems, we use the embedding layer design in Algorithm 3, we design a training dataset that consists of matrices with dimensions in $[200, 2000]$, and we constrain the condition number of matrices spanning $[10, 10^5]$. Spectral probes are obtained with different numbers of subspace iterations in $[1, 500]$. The spectral probe contains the largest and smallest k eigenvalue estimates and corresponding residual norms, obtained using subspace itreations. In practice, we choose $k = 20$, and for estimates of the smallest eigenvalues, we obtain them by performing subspace iteration on shifted matrices to obtain the largest eigenpairs of the shifted matrix. We then obtain the estimates for the smallest eigenvalue by reverse-shifting the estimations and compute the residual using approximated eigenvectors on the unshifted matrix.

We train the embedding layer while freezing the backbone model for 500 epochs, with a dataset of size 50000 and a batch size of 200. We use AdamW optimizer with cosine learning rate decay after

10% of warmup steps. We search for the best learning rate among $\{10^{-3}, 5\times10^{-4}, 10^{-4}\}$ and select the best learning rate, and we use a weight decay among the best of $\{10^{-2}, 10^{-3}, 5\times10^{-4}\}$.

Approximating Matrix Functions. For approximating matrix functions, we adopt the same embedding layer structure as preconditioned linear systems, shown in Algorithm 3. We construct the training dataset with the same method as for linear systems, while increasing the portion of matrices with small condition numbers, as the polynomial method is most suitable for moderate condition number regime.

We train the embedding layer while freezing the backbone model for 500 epochs, with a dataset of size 50000 and a batch size of 200. We use AdamW optimizer with cosine learning rate decay after 10% of warmup steps. We search for the best learning rate among $\{10^{-3}, 5\times10^{-4}, 10^{-4}\}$ and select the best learning rate, and we use a weight decay among the best of $\{10^{-2}, 10^{-3}, 5\times10^{-4}\}$.

F Expressive Power of the Neural Engine

We show that AUTOSPEC can exactly generate classical Chebyshev filters and polynomial preconditioners using the transitions in (4), with coefficients generated by an affine layer followed by learned normalization (Algorithm 1) and suitable fixed weights and biases.

F.1 Polynomial filtering for eigenproblems

Let $[a, b]$ be a prescribed interval with $a < b$. The affine map

$$\theta(\lambda) = \frac{2\lambda - (a+b)}{b-a}$$

maps $[a, b]$ to $[-1, 1]$. Let T_d denote the degree-d Chebyshev polynomial, defined by

$$T_0(t) = 1, \qquad T_1(t) = t, \qquad T_{j+1}(t) = 2tT_j(t) - T_{j-1}(t), \quad j \geq 1.$$

The shifted polynomial $T_d(\theta(\lambda))$ provides a classical construction for polynomial filtering.

The following proposition shows that the neural engine, with each transition component parameterized as in Algorithm 1, can generate the coefficients of this filter from the exact input endpoints a, b, using fixed weights and biases.

Proposition F.1 (Exact generation of Chebyshev polynomials)**.** *For every polynomial degree $d \geq 2$, there exist fixed weights and biases for the neural engine in Algorithm 1 such that, for every interval $[a, b]$ with $a < b$, supplying the exact endpoints a, b as input yields*

$$P(\mathbf{X}) = T_d(\theta(\mathbf{X})), \qquad \theta(\lambda) = \frac{2\lambda - (a+b)}{b-a}.$$

The construction uses d transitions of the form (4), and holds in exact arithmetic. The weights and biases may depend on d, but are independent of a, b.

Proof. Set $\rho_k = 0$ and write $h = b - a$ and $m = a + b$. Choose

$$(\gamma_k, \eta_k, \alpha_k, \beta_k) = \frac{1}{h}\begin{cases}(-m, 0, 2, 0), & k = 1,\\ (-h, 0, -2m, 4), & k = 2,\\ (0, -h, -2m, 4), & 3 \leq k \leq d.\end{cases}$$

The numerators are affine functions of a, b, so fixed weights and biases can generate them. Choosing the normalization output $\delta_k = b - a - \epsilon$ supplies the denominator h.

Starting from $\mathbf{A}_1 = \mathbf{B}_1 = \mathbf{I}$ and $\mathbf{C}_1 = \mathbf{X}$, the first two updates in (5) give

$$\mathbf{C}_2 = \theta(\mathbf{X}), \qquad \mathbf{C}_3 = 2\theta(\mathbf{X})^2 - \mathbf{I}.$$

The remaining transitions satisfy

$$\mathbf{C}_{k+1} = 2\theta(\mathbf{X})\mathbf{C}_k - \mathbf{C}_{k-1}.$$

These are precisely the Chebyshev initial values and recurrence, hence $\mathbf{C}_{d+1} = T_d(\theta(\mathbf{X}))$. All network weights and biases are independent of a, b. □

F.2 Polynomial preconditioners for linear systems

Assume that $0 < a < b$ and that $[a, b]$ contains the spectrum of the spd operator. Define the normalized Chebyshev residual polynomial

$$R_{d+1}(\lambda) = \frac{T_{d+1}(\theta(\lambda))}{T_{d+1}(\theta(0))}$$

and the associated preconditioner

$$P(\lambda) = \frac{1 - R_{d+1}(\lambda)}{\lambda}.$$

Since $R_{d+1}(0) = 1$, the numerator is divisible by λ, and P is a polynomial of degree d. Among polynomial preconditioners of degree at most d, this construction minimizes the maximum absolute residual $\max_{\lambda \in [a,b]} |1 - \lambda P(\lambda)|$.

The following proposition shows that AUTOSPEC can generate this preconditioner from the exact endpoints a, b, using weights and biases that are shared across all admissible intervals.

Proposition F.2 (Exact generation of Chebyshev polynomial preconditioners)**.** *For every polynomial degree $d \geq 2$, there exist fixed weights and biases for the neural engine, with each transition component parameterized as in Algorithm 1, such that, for every interval $[a, b]$ with $0 < a < b$, supplying the exact endpoints a, b as input yields $P(\mathbf{X})$, where*

$$\theta(\lambda) = \frac{2\lambda - (a+b)}{b-a}, \qquad R_{d+1}(\lambda) = \frac{T_{d+1}(\theta(\lambda))}{T_{d+1}(\theta(0))}, \qquad P(\lambda) = \frac{1 - R_{d+1}(\lambda)}{\lambda}.$$

The construction uses $d+1$ transitions of the form (4), and holds in exact arithmetic. The weights and biases may depend on d, but are independent of a, b.

Proof. The roots of $T_{d+1}(\theta(\lambda))$ are

$$t_j = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{(2j-1)\pi}{2(d+1)}\right), \qquad j = 1, \ldots, d+1.$$

Since $R_{d+1}(0) = 1$,

$$R_{d+1}(\lambda) = \prod_{j=1}^{d+1} \left(1 - \frac{\lambda}{t_j}\right).$$

Set $\rho_j = \eta_j = 0$ and choose

$$(\gamma_1, \alpha_1, \beta_1) = \frac{1}{t_1}(1, 0, 0), \qquad (\gamma_j, \alpha_j, \beta_j) = \frac{1}{t_j}(1, t_j, -1), \quad 2 \leq j \leq d+1.$$

Let $C_k(\lambda)$ denote the scalar polynomial corresponding to the state $\mathbf{C}_k$ in (5). The first transition gives $C_2(\lambda) = 1/t_1$, and the remaining transitions give

$$C_{j+1}(\lambda) = \frac{1}{t_j} + \left(1 - \frac{\lambda}{t_j}\right) C_j(\lambda).$$

Consequently,

$$1 - \lambda C_{j+1}(\lambda) = \left(1 - \frac{\lambda}{t_j}\right)\big(1 - \lambda C_j(\lambda)\big),$$

so induction yields

$$1 - \lambda C_{d+2}(\lambda) = R_{d+1}(\lambda).$$

Thus, the final output is $C_{d+2} = P$.

Each t_j is a positive affine function of a, b. Fixed weights and biases can therefore generate the coefficient numerators, while choosing $\delta_j = t_j - \epsilon$ supplies the denominator through normalization. These parameters depend only on d and j, and are independent of a, b. □

G Extended Discussion of Numerical Experiments on Real-World Eigenproblems and Linear Systems

This section provides a detailed description of the experimental setup used in the evaluation in Section 5, together with additional results and analysis. This includes absolute and relative performance of our discovered polynomial preconditioning, sensitivity to spectral probe quality, and comparisons with state-of-the-art Chebyshev preconditioning.

G.1 Experimental setting

We mainly evaluate preconditioners by iteration count reduction, a natural metric for comparing polynomial preconditioners of the same degree [11, 60, 61]. To support our main claims, including the small effect of spectral probing on total runtime, we also provide end-to-end runtime comparisons. We note that absolute runtimes and relative speedups are implementation- and architecture-dependent.

The AUTOSPEC engines are trained offline and, at inference time, are called from MATLAB to produce polynomial recurrence coefficients from spectral probes. For eigenvalue problems, these probes are obtained by running 20 iterations of `eigs`, which implements the Krylov–Schur algorithm. From the resulting Ritz spectrum, the engine receives 10 eigenvalue estimates associated with the target region, corresponding to the first k requested eigenvalues, and 94 eigenvalue estimates associated with the out-of-target region, corresponding to the subsequent Ritz values. If fewer than 10 target values are available, we sample from the available values with replacement; if more than 10 are available, we uniformly subsample 10 values. The same procedure is used to obtain exactly 94 out-of-target values.

For linear systems, spectral probes are produced by 50 or 200 Lanczos steps without storing the Lanczos basis. From the resulting approximate spectrum, we select the 20 largest and 20 smallest Ritz values and use them as the spectral input to the neural engine.

The forward pass of the AUTOSPEC engine should have a negligible cost compared to other computations. The overall preconditioner construction cost is dominated by the spectral probe stage: a cost equivalent to 20 iterations of the corresponding unpreconditioned eigensolver or 50 iterations of the corresponding unpreconditioned linear solver, and is also negligible for most tested systems, while modest for the remaining cases.

We focus on spd operators and spd preconditioners. Indefinite eigenvalue problems are converted to spd form by implicitly squaring the operator.

G.2 Polynomial Preconditioners as Second-level Accelerators

In linear systems experiments, we first apply a classical inexpensive preconditioner and then construct a polynomial preconditioner for second-level acceleration. In most cases, the first-level preconditioner is Jacobi (the inverse action $\mathbf{B} = \mathrm{diag}(\mathbf{A})^{-1}$). Whenever other preconditioners, such as AMG or iChol(0), are used instead, this is stated explicitly. We describe this approach below.

Consider iterative solution of a linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{26}$$

with an spd first-level preconditioner $\mathbf{B}$. We can further accelerate the solver by applying a polynomial preconditioner to $\mathbf{AB}$. Specifically, we can construct a polynomial $P(\mathbf{AB})$ and use the composite preconditioner $\widehat{\mathbf{B}} = \mathbf{B}P(\mathbf{AB})$ for the solution of (26). The resulting preconditioned operator is $\mathbf{A}\widehat{\mathbf{B}} = \mathbf{AB}P(\mathbf{AB})$. Its spectrum coincides with that of

$$\mathbf{X}P(\mathbf{X}), \qquad \mathbf{X} = \mathbf{B}^{1/2}\mathbf{A}\mathbf{B}^{1/2}.$$

Thus, polynomial acceleration can be realized on an spd operator $\mathbf{X}$.

In practice, however, an explicit factorization of $\mathbf{B}$ may be unavailable, as is often the case for AMG. This makes it impractical to apply an eigensolver warm start directly to $\mathbf{X}$ in order to compute a spectral probe. We therefore obtain spectral probes through a generalized Lanczos process applied to eigenvalue problem $\mathbf{A}\mathbf{v}_i = \lambda_i \mathbf{B}^{-1}\mathbf{v}_i$, which is equivalent to the standard eigenproblem $\mathbf{X}\mathbf{u}_i = \lambda_i \mathbf{u}_i$. This yields Ritz values λ_i together with residual estimates

$$\left\|\mathbf{A}\mathbf{v}_i - \lambda_i \mathbf{B}^{-1}\mathbf{v}_i\right\|_{\mathbf{B}} = \left\|\mathbf{X}\mathbf{u}_i - \lambda_i \mathbf{u}_i\right\|_2, \qquad \mathbf{u}_i = \mathbf{B}^{-1/2}\mathbf{v}_i,$$

which are used as inputs to the neural engine.

Table 2: Test matrices used in our experiments, drawn from the SuiteSparse Matrix Collection or generated by PDE discretization, with their dimensions and estimated condition numbers.

(a) Linear Systems

Matrix A	**dim**	**Precond. B**	$\mathbf{cond(X)}$
`laplacian`	6084	Jacobi	3.7e3
`Dubcova2`	65025	Jacobi	1.1e4
`thermal1`	82654	Jacobi	5.0e5
`FEM_3D_thermal2`	147900	Jacobi	5.3e1
`G2_circuit`	150102	Jacobi	6.9e5
`shipsec5`	179860	Jacobi	1.4e6
`thermal_cookie`	250000	Jacobi	1.8e6
`laplacian_medium`	389376	Jacobi	2.3e5
`parabolic_fem`	525825	Jacobi	2.3e5
`Fault_639`	638802	Jacobi	8.4e6
`Emilia_923`	923136	Jacobi	2.9e7
`thermal2`	1228045	Jacobi	7.6e6
`laplacian_large`	1557504	Jacobi	9.2e5
`G3_circuit`	1585478	Jacobi	7.1e5

(b) Eigenproblems

Matrix X	**dim**	**squared**	$\mathbf{cond(X)}$
`thermal1`	82654	no	3.2e5
`thermal2`	1228045	no	4.0e6
`SiO2`	155331	yes	5.6e6
`CO`	221119	yes	6.3e7

G.3 Test Matrices

Test matrices are drawn from the SuiteSparse Matrix Collection and cover electronic-structure, thermal finite element, structural/mechanics, and circuit/model-reduction applications. Matrix sizes range from $n = 468$ to $n = 1.6 \times 10^6$, with estimated condition numbers between 50 and 2.9×10^7. We also enriched the test set with few matrices obtained by finite-difference/finite-volume PDE discretization.

Table 2 summarizes the properties of matrices used in our experiments. The generated matrices comprise three finite-difference Laplacians and `thermal_cookie`, a finite-volume diffusion PDE discretization with spatially varying conductivity. For the listed linear systems we used Jacobi first-level preconditioning. For eigenvalue problems, the table indicates whether the operator is implicitly squared to make it spd.

G.4 Quality of Learned Second-level Accelerators for Linear Systems.

Table 3 evaluates the preconditioners produced by AUTOSPEC engines on the systems from Table 2.

We assess preconditioner quality by the number of CG iterations required to reach residual tolerance 10^{-10}, using spectral probes obtained from either 50 or 200 Lanczos steps. To ensure the accuracy of the computed solutions, we verify that the final true relative residual does not exceed 1.1 times the requested tolerance[4]. For reference, we also report iteration counts for Neumann acceleration and unaccelerated CG at the same tolerance. The results show that the learned preconditioners substantially reduce iteration counts across all tested problems and remain effective over a wide range of condition numbers, even when constructed from coarse spectral probes.

To broaden the empirical analysis, we evaluate all spd matrices in the SuiteSparse Matrix Collection with sizes between 10^4 and 10^5 and at most 2×10^6 nonzeros; see Figure 7 for a statistical summary of the resulting iteration counts. In this broader evaluation, polynomial preconditioners consistently reduce CG iteration counts, often by up to an order of magnitude.

G.5 Quality of Learned Preconditioners for Eigenvalue Problems.

For the eigenproblems defined by the matrices in Table 2b, Table 3b reports the number of `eigs` iterations[5] required to attain a relative eigenvalue error of 10^{-10} on the spectrally transformed operator $\mathbf{I} - \mathbf{X}/1.1c$, where $c \approx \lambda_{\max}(\mathbf{X})$. We report this count for the k-th largest eigenvalue of

[4]To mitigate stagnation at a finite-precision error floor on `shipsec5`, `Fault_639`, and `Emilia_923`, we run `pcg` with restarting, first targeting a relative residual of 10^{-7} and then solving the residual correction equation.

[5]In `eigs` we request a slightly larger number of eigenvalues, which is standard practice to improve convergence.

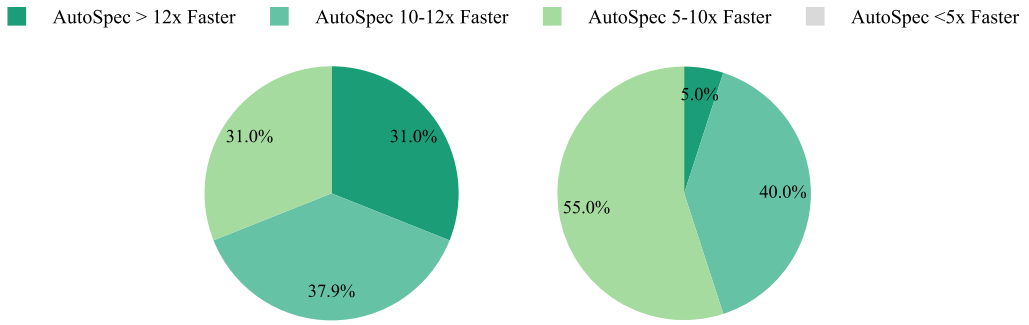
Table 3: Quality of preconditioners produced by AUTOSPEC.

(a) Linear systems: CG iteration counts for linear systems at residual tolerance 10^{-10} using AUTOSPEC acceleration with 50-step and 200-step spectral probes, Neumann acceleration, and no acceleration. Results are reported over five random seeds.

Matrix A	**numIters** (probe=200)	**numIters** (probe=50)	**numIters** (Neumann)	**numIters** (no acceleration)
`laplacian`	29 ± 0	27 ± 0	79 ± 0	275 ± 1
`Dubcova2`	36 ± 0	35 ± 0	103 ± 1	360 ± 2
`thermal1`	154 ± 2	162 ± 2	481 ± 6	1648 ± 18
`FEM_3D_thermal2`	7 ± 0	6 ± 0	16 ± 0	56 ± 0
`G2_circuit`	205 ± 2	219 ± 2	639 ± 3	2245 ± 24
`shipsec5`	541 ± 18	554 ± 4	1673 ± 11	5812 ± 21
`thermal_cookie`	312 ± 6	338 ± 1	967 ± 16	3396 ± 64
`laplacian_medium`	195 ± 1	207 ± 1	605 ± 4	2112 ± 14
`parabolic_fem`	318 ± 3	338 ± 3	989 ± 8	3451 ± 24
`Fault_639`	794 ± 9	874 ± 20	2499 ± 41	9278 ± 206
`Emilia_923`	1147 ± 25	1148 ± 6	3411 ± 27	13047 ± 126
`thermal2`	592 ± 5	626 ± 5	1850 ± 13	6369 ± 51
`laplacian_large`	384 ± 2	409 ± 2	1192 ± 3	4174 ± 11
`G3_circuit`	402 ± 9	438 ± 6	1218 ± 13	4369 ± 85

(b) Eigenproblems: AUTOSPEC preconditioners are compared to basic power method-based (i.e., $P(\mathbf{X}) = \mathbf{X}^d$) preconditioners. The number of `eigs` iterations required to reach relative eigenvalue error 10^{-10}. For all matrices, we compute k=10 and k=20 eigenvalues using `eigs` with k+5 requested eigenvalues and subspace dimension l=4k.

Matrix X	**numIters** (AUTOSPEC, λ_{10})	**numIters** (AUTOSPEC, λ_{20})	**numIters** (power, λ_{10})	**numIters** (power, λ_{20})
`thermal1`	4	3	19	12
`thermal2`	21	11	112	45
`SiO2`	23	23	>500	108
`CO`	55	27	>500	126



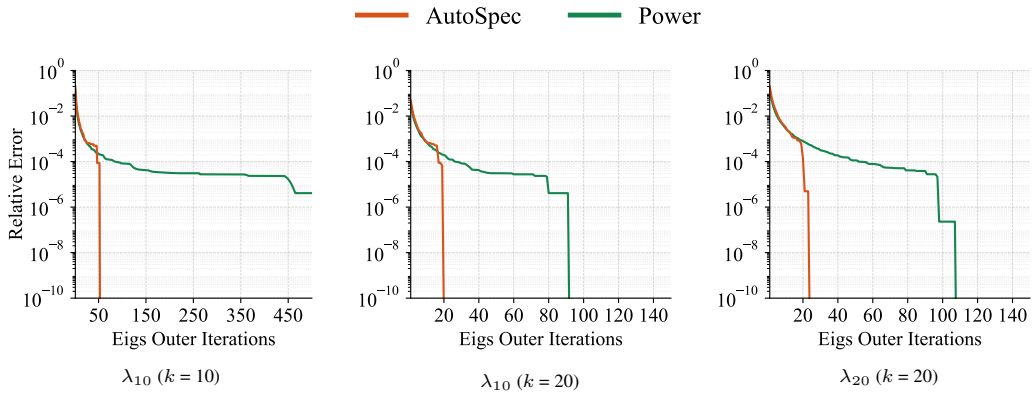
(a) 50 Probe Iterations (b) 200 Probe Iterations

Figure 7: Comparing average iteration counts over 5 seeds of unaccelerated CG, and CG accelerated by AUTOSPEC, on all spd SuiteSparse matrices with sizes between 10^4 and 10^5 and at most 2×10^6 nonzeros.

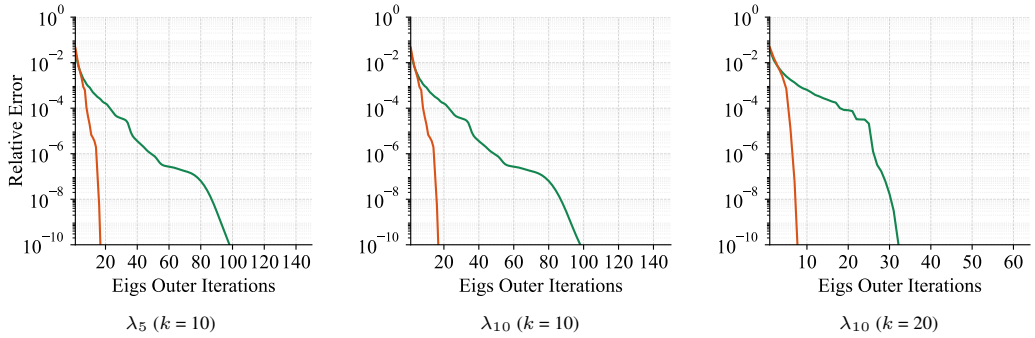
the shifted operator, with $k = 10$ and $k = 20$; these eigenvalues correspond to the k-th smallest eigenvalues of the original operator. The learned preconditioners consistently reduce the iteration counts relative to the power-preconditioner baseline $P(\mathbf{X}) = \mathbf{X}^d$, demonstrating their effectiveness in accelerating Krylov–Schur iterations under tight memory and communication budgets.

In addition to the results in Section 5.3, we present complementary experiments on convergence of preconditioned Krylov–Schur with varying numbers of requested eigenvalues k and corresponding subspace dimensions l. Figure 8 shows results for the `Si02` and `thermal2` matrices with targets $k = 10$ and $k = 20$.

In both cases, the preconditioners obtained with AUTOSPEC engine require substantially fewer Krylov–Schur iterations to reach target errors, indicating that the learned preconditioning method adapts and remains effective for different choices of k and l.



(a) `Si02`



(b) `thermal2`

Figure 8: Convergence of eigenvalue estimates versus Krylov–Schur outer iterations for `Si02` and `thermal2` matrices. Parameter k denotes the total number of smallest-magnitude eigenpairs computed with subspace dimension $l = 4k$.

G.6 Comparison with Chebyshev polynomial preconditioners.

We next compare AUTOSPEC preconditioners with classical Chebyshev polynomial preconditioners; see, e.g., [8, 58, 60, 61]. We implement the actions $\mathbf{z} \mapsto P_d(\mathbf{X})\mathbf{z}$ using the low storage, and inner-product free recurrence from Bergamaschi and Martinez Calomardo [11, Algorithm 2].

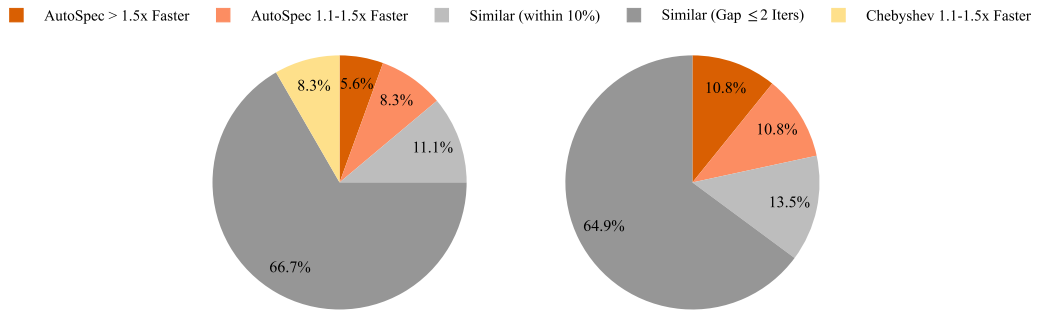
Table 4 compares the number of CG iterations needed to reach a residual tolerance of 10^{-10} when the Chebyshev method uses the same extremal eigenvalue estimates as the neural engine, obtained from either 50 or 200 Lanczos steps. In most cases, AUTOSPEC preconditioners match or outperform Chebyshev preconditioning, often by a large margin, and are more robust to changes in probe quality.

Remark G.1. In Bergamaschi and Martinez Calomardo [11, Algorithm 2], the coefficient indices in line 6 appear to be shifted relative to the derived Chebyshev recurrence. Replacing ρ_{k+1} and ρ_k with ρ_k and ρ_{k-1}, respectively, would align the update with that recurrence. We evaluated both formulations and obtained similar results.

To further strengthen our claims, we evaluate all spd matrices in the SuiteSparse Matrix Collection with sizes between 10^4 and 10^5 and at most 2×10^6 nonzeros. Figure 9 summarizes the resulting iteration counts.

Table 4: Comparison of AUTOSPEC and Chebyshev polynomial accelerators on the SuiteSparse matrices listed in Table 2a. Chebyshev accelerators use the same estimated extremal eigenvalues provided to the neural network engine. We report CG iteration counts required to reach residual tolerance 10^{-10}, using preconditioners constructed from 50- or 200-step Lanczos spectral probes.

Matrix A	**numIters** (AUTOSPEC, probe=200)	**numIters** (AUTOSPEC, probe=50)	**numIters** (Cheb, probe=200)	**numIters** (Cheb, probe=50)
`laplacian`	29 ± 0	27 ± 0	48 ± 0	39 ± 7
`Dubcova2`	36 ± 0	35 ± 0	67 ± 1	58 ± 21
`thermal1`	154 ± 2	162 ± 2	265 ± 50	145 ± 1
`FEM_3D_thermal2`	7 ± 0	6 ± 0	5 ± 0	5 ± 0
`G2_circuit`	205 ± 2	219 ± 2	301 ± 40	228 ± 56
`shipsec5`	541 ± 18	554 ± 4	540 ± 20	518 ± 15
`thermal_cookie`	312 ± 6	338 ± 1	653 ± 299	684 ± 331
`laplacian_medium`	195 ± 1	207 ± 1	257 ± 50	407 ± 91
`parabolic_fem`	318 ± 3	338 ± 3	394 ± 163	445 ± 207
`Fault_639`	794 ± 9	874 ± 20	752 ± 10	992 ± 12
`Emilia_923`	1147 ± 25	1148 ± 6	1087 ± 20	1034 ± 4
`thermal2`	592 ± 5	626 ± 5	558 ± 3	557 ± 4
`laplacian_large`	384 ± 2	409 ± 2	391 ± 27	618 ± 322
`G3_circuit`	402 ± 9	438 ± 6	918 ± 430	389 ± 5



(a) 50 Probe Iterations (b) 200 Probe Iterations

Figure 9: Comparing CG iteration counts over 5 seeds of AutoSpec and Chebyshev polynomial preconditioning, on all spd SuiteSparse matrices with sizes between 10^4 and 10^5 and at most 2×10^6 nonzeros.

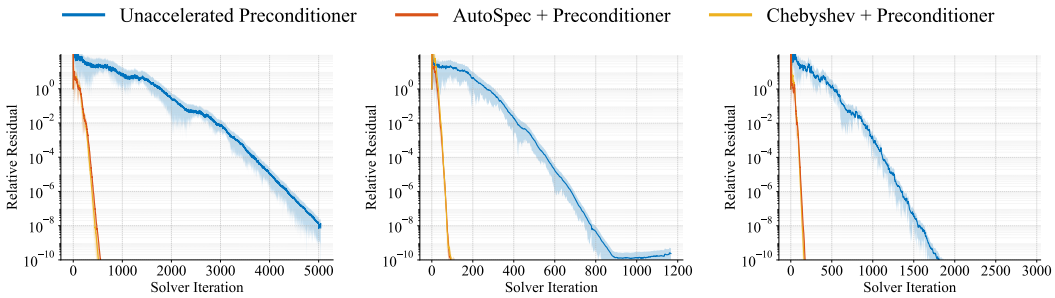
G.7 Comparison with Chebyshev Acceleration under Different First-Level Preconditioners

We evaluated AUTOSPEC on `Shipsec5` and `Fault_639` using Jacobi, AMG, and iChol(0) as first-level preconditioners, and compare its performance with Chebyshev acceleration under the same settings (see Figures 10 and 11).

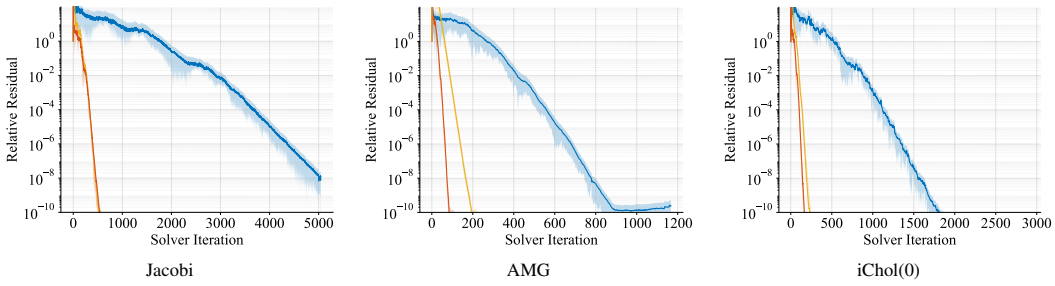
Overall, our results show that AUTOSPEC polynomial preconditioners provide strong and robust acceleration across diverse real-world matrices, supporting our algorithm-discovery approach. They further suggest that AUTOSPEC can help automate advances in numerical linear algebra: the discovered preconditioners operate reliably using only limited spectral information, adapt to the structure of the input operator, and sometimes can outperform widely used classical methods.

G.8 Comparison with Direct Polynomial Fitting and Expanded Linear-System

To compare the current AUTOSPEC neural network with simpler architectures, we compare AUTOSPEC with two direct polynomial-fitting baselines: least-squares (LS) fitting and minimax fitting. While AUTOSPEC predicts recurrence coefficients for different test operators using one trained neural engine, these baselines optimize polynomial coefficients separately for each test operator using its spectral probe.

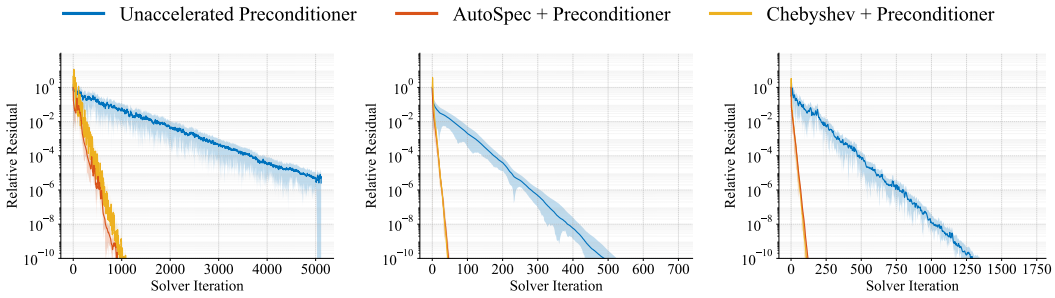


(a) `Shipsec5` with 50 Probe Iterations

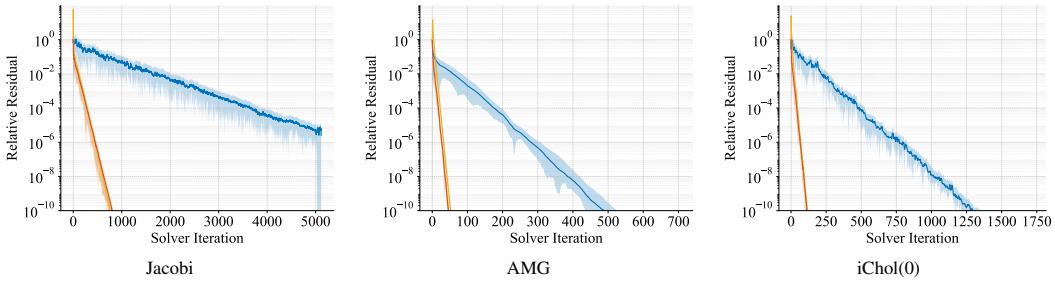


(b) `Shipsec5` with 200 Probe Iterations

Figure 10: AutoSpec and Chebyshev as second-level accelerators for CG, preconditioned using Jacobi, AMG, and iChol(0) on the `Shipsec5` system.



(a) `Fault_639` with 50 Probe Iterations



(b) `Fault_639` with 200 Probe Iterations

Figure 11: AutoSpec and Chebyshev as second-level accelerators for CG, preconditioned using Jacobi, AMG, and iChol(0) on the `Fault_639` system.

We use the 20 smallest and 20 largest Ritz values from a 50- or 200-step Lanczos run and augment them with 2,000 samples in the estimated spectral interval $[a, b]$. For a degree-11 polynomial, let

$$P(\lambda) = \sum_{j=0}^{11} c_j T_j(\theta(\lambda)), \qquad \theta(\lambda) = \frac{2\lambda - (a+b)}{b-a}. \tag{27}$$

Table 5: PCG iteration counts to reach a relative residual of 10^{-10}. Polynomial methods use degree-11 preconditioners constructed from 50- or 200-step Lanczos probes. LS and Minimax denote direct least-squares and minimax polynomial fitting, respectively.

Matrix	Probe Iters	No Polynomial	AUTOSPEC	Chebyshev	LS	Minimax
G2_circuit	50	2242 ± 16	213 ± 1	223 ± 56	254 ± 1	218 ± 44
	200	2242 ± 16	210 ± 2	262 ± 31	271 ± 1	264 ± 31
Dubcova2	50	359 ± 2	35 ± 0	62 ± 16	41 ± 0	66 ± 22
	200	359 ± 2	36 ± 0	69 ± 4	43 ± 0	70 ± 4
laplacian_small	50	274 ± 1	28 ± 0	37 ± 5	31 ± 0	40 ± 7
	200	274 ± 1	29 ± 0	48 ± 0	33 ± 0	49 ± 0
laplacian_medium	50	2112 ± 15	202 ± 1	263 ± 68	240 ± 2	264 ± 68
	200	2112 ± 15	199 ± 1	233 ± 24	255 ± 2	234 ± 25
laplacian_large	50	4152 ± 30	397 ± 3	426 ± 128	470 ± 3	427 ± 130
	200	4152 ± 30	390 ± 3	534 ± 143	500 ± 3	529 ± 136
thermal_cookie	50	3377 ± 76	322 ± 7	659 ± 337	382 ± 7	718 ± 393
	200	3377 ± 76	316 ± 7	417 ± 172	405 ± 6	422 ± 183
parabolic_fem	50	3438 ± 54	329 ± 3	445 ± 298	390 ± 7	453 ± 307
	200	3438 ± 54	324 ± 3	346 ± 85	414 ± 8	347 ± 86
G3_circuit	50	4440 ± 70	426 ± 6	394 ± 4	487 ± 9	392 ± 4
	200	4440 ± 70	418 ± 6	706 ± 100	519 ± 10	691 ± 95

Here T_j is the degree-j Chebyshev polynomial. The Chebyshev basis avoids the monomial basis's scaling problems, and the additional interval samples constrain behavior between the observed Ritz values. Writing $H_{ij} = \lambda_i T_j(\theta(\lambda_i))$, the least-squares fit solves $\min_c \|Hc - \mathbf{1}\|_2^2$ using an SVD-based solver. The minimax fit is the linear program

$$\min_{c,t} \; t \quad \text{subject to} \quad -t\mathbf{1} \le Hc - \mathbf{1} \le t\mathbf{1}, \qquad t \ge 0. \tag{28}$$

Both objectives approximate the inverse through $xP(x) \approx 1$.

In Table 5, we report PCG iteration counts to reach a relative residual of 10^{-10} over five random seeds. AUTOSPEC uses the fewest iterations on every matrix (except G3_circuit with 50 probe iterations, where AUTOSPEC has 10% more iterations than Minimax fitting).

G.9 Runtime and Resource Usage

Runtime experiments from Section 6.3 are performed on NERSC's Perlmutter system, equipped with AMD EPYC 7763 (Milan) processors with 64 physical cores per processor. We use MATLAB R2023b and its built-in pcg solver. Both MATLAB and the external C++ polynomial evaluator are configured to use 8 threads. The evaluator is invoked through a MEX interface for both AUTOSPEC and Chebyshev preconditioning.

Reported runtimes use degree-11 polynomials and 200-step Lanczos probes. Total runtime includes preconditioning setup and the iterative solve. Setup comprises Jacobi preconditioner construction and, for polynomial methods, spectral probing, coefficient generation, and native-kernel setup. After a short untimed warm-up, we collect three repetitions per seed, take the median total runtime within each seed, and average these medians across seeds.

H Ablation Studies

H.1 AUTOSPEC Polynomial Degree and Network Architecture

Training AUTOSPEC engine with varying degree. The same training recipe can be applied to training the AutoSpec neural engine with varying layers. Table 6 shows the PCG iteration count on the Fault_639 matrix when polynomial accelerators of different degrees are applied. We can see that AutoSpec consistently provides acceleration in iteration count compared to unaccelerated PCG and Chebyshev method.

Table 6: PCG iteration counts on `Fault_639` for polynomial accelerators of different degrees. Unaccelerated solves did not converge within 5,000 iterations.

Degree	Unaccelerated	AUTOSPEC	Chebyshev
6	5000+	1386 ± 18	1884 ± 157
9	5000+	919 ± 10	995 ± 7
11	5000+	827 ± 9	828 ± 8
13	5000+	712 ± 8	799 ± 46
16	5000+	560 ± 6	699 ± 49

Alternative AUTOSPEC backbone structures. The current AUTOSPEC backbone layer consists of a linear layer followed by a learned scaling. We evaluate alternative architectures: a vanilla linear layer (without the scaling), and an MLP with one hidden layer and ReLU activation. We train a 15-layer backbone and observe the convergence improvement $\rho_{\log}$.

As shown in Table 7, using only linear layers results in significantly worse performance than linear with scaling, while using MLP layer has similar performance compared to linear with scaling. We note that linear-with-scaling layer provides sufficient expressiveness. While MLP is more expressive, it increases the total parameter count and actual number of layers, which would cost more to train.

Table 7: Effect of the backbone layer architecture on convergence improvement $\rho_{\log}$. Higher values indicate greater improvement.

	`linear_norm` (paper)	`linear`	`mlp`	Chebyshev baseline
Convergence Improvement (↑)	16.90	9.015	17.43	16.37

Alternative AUTOSPEC embedding structures. Our intention for the embedding layer was to generate the embedding features of the largest and smallest end of the spectrum separately using two MLP layers, and concatenate them. This prevents the magnitude difference between the largest and smallest eigenvalues from affecting training. We perform an ablation study that trains one MLP layer taking all $\widehat{\lambda}$ and $\widehat{r}$ as input to generate the embedding vector. We train the embedding of a 15-layer backbone and observe the convergence improvement $\rho_{\log}$.

Table 8 shows that using only one MLP layer as embedding results in significantly worse convergence improvement than two MLPs. Therefore, our current design is effective for our chosen tasks, and developing more expressive network structures is an interesting direction for more complex NLA tasks.

Table 8: Effect of the spectral embedding architecture on convergence improvement $\rho_{\log}$. Higher values indicate greater improvement.

	Two MLPs (paper)	One MLP	Chebyshev baseline
Convergence Improvement (↑)	15.47	9.015	15.24

H.2 Sensitivity to Spectral Profile Coverage in Training

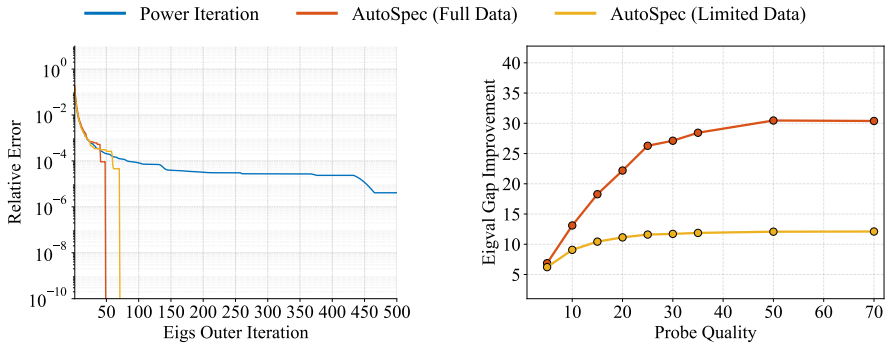
To illustrate the discussion on limitations in Section 6, we provide results on the sensitivity of the trained AUTOSPEC engine to spectral profile coverage in training, where insufficient training leads to performance degradation.

The table below shows the log eigengap improvement $\rho_{\log}$ for different methods evaluated on synthetic eigenspectra with varying eigengap between λ_{10} and λ_{100} (measured by $(\lambda_{10}-\lambda_{100})/\lambda_{10}$), AutoSpec trained with eigengap in $[10^{-2}, 10^{-1}]$ has only moderate $\rho_{\log}$ for eigengaps outside the training range. As we increase the span of eigengap in the training data, AutoSpec achieves significantly better performance with superior $\rho_{\log}$.

Table 9: Log eigengap improvement $\rho_{\log}$ across test eigengap ranges. Each row specifies the training eigengap range used for AUTOSPEC. Higher values indicate greater improvement.

Method, Training Eigengap Range	$(10^{-2}, 10^{-1}]$	$(10^{-3}, 10^{-2}]$	$(10^{-4}, 10^{-3}]$	$[10^{-5}, 10^{-4}]$
AUTOSPEC, $[10^{-2}, 10^{-1}]$	4.20	9.32	9.28	9.28
AUTOSPEC, $[10^{-5}, 10^{-1}]$	5.21	11.80	16.13	16.77

In addition, we evaluate the sensitivity on a realistic matrix. As shown in Figure 12, for eigenvalue problems, if the model is trained only on limited training data containing spectra with larger eigengaps than those encountered at test time (yellow line), performance can degrade compared to training with more diverse data (red line).



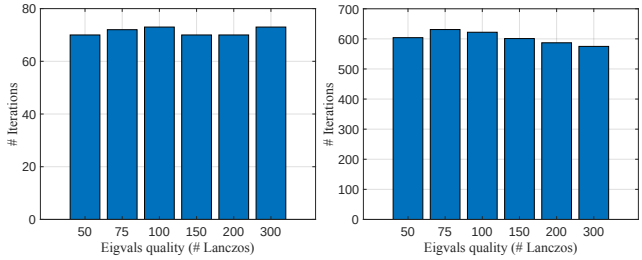
(a) Convergence of the eigenvalue approximations versus the number of outer iterations of `eigs`.

(b) Improvement in the eigenvalue gap relative to the baseline filter $P(X) = X^d$.

Figure 12: We compare the spectral filter produced for the `Si02` matrix using the neural-engine checkpoint from the paper with that produced by a neural engine trained only on spectra exhibiting relatively large eigenvalue gaps.

H.3 Robustness of Discovered Algorithms to Spectral Input Perturbations

To assess the robustness of the discovered linear solver accelerator, we perturb the spectral probe by varying the Lanczos iteration counts. Figure 13 (in the appendix) shows that once probe quality is sufficiently high (above 75 Lanczos iterations), the CG iteration count to reach 10^{-10} error stabilizes, indicating robustness to spectral-input perturbations.



(a) `G2_circuit`

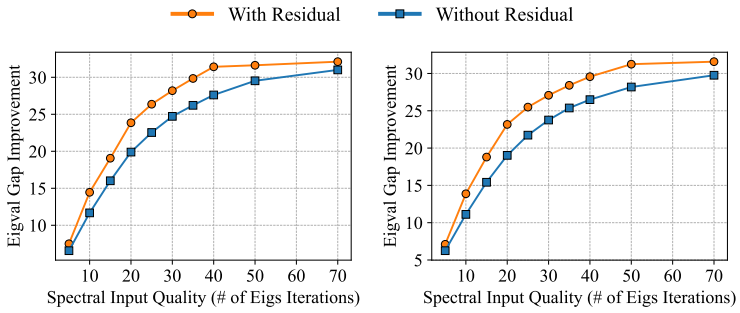
(b) `thermal2`

Figure 13: CG iterations to reach 10^{-10} error vs Lanczos iterations used for NN input spectral probe.

H.4 Role of Spectral Residual Features in Discovering Algorithms

Given eigenvalue estimations augmented by corresponding residual features, the embedding layer "reasons" and exploits critical spectral information, resulting in the discovery of effective algorithms. To further analyze how the embedding layer uses the spectral residuals, we perform an ablation

study by setting the residual to 0 and observing how it affects the construction of algorithms and performance on synthetic matrices.



(a) `SiO2` (b) `thermal2`

Figure 14: Comparison of learned eigen preconditioners on `SiO2` and `thermal2`, generated with versus without providing estimated eigenvalue residuals as input to the neural engine. Performance is reported as the log improvement in the target-boundary eigenvalue gap relative to standard subspace iteration baseline $P(\mathbf{X}) = \mathbf{X}^d$.

As shown in Figure 14, incorporating residuals as input improves the effectiveness of the learned preconditioners. When the model is provided with residual features, it consistently achieves larger eigenvalue gap improvements over subspace iteration $P(\mathbf{X}) = \mathbf{X}^d$ across all iteration budgets. This indicates that residual features provide essential spectral-quality signals for the model to generate effective preconditioning algorithms. In Figure 17, we present additional results on synthetic matrices, confirming that residual features play an important role in the algorithm generation of AUTOSPEC.

I Complementary Results on Synthetic Matrices

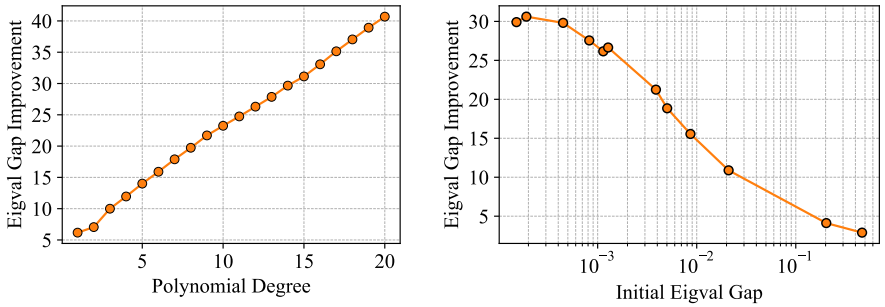
I.1 Eigenvalue Problems

Backbone Model on Synthetic Matrices. Figure 15a shows the eigenvalue gap improvement of polynomial preconditioning algorithms generated by the first k layers of the backbone model after pre-training ($1 \leq k \leq 20$), compared to the standard subspace iteration preconditioner. We can see that the polynomial generated by any first k layers of the backbone model brings significant improvement of the eigenvalue gap after preconditioning. This shows the pre-training of discovering preconditioning algorithms for eigenvalue problems achieves "length-generalization," that any of the first k layers can produce an effective algorithm. This enables us to extend the model to learn higher-degree preconditioning algorithms.

Embedding Model on Synthetic Matrices. Figure 15b shows the eigenvalue gap improvement of the model after pre-training, given the spectral probe input obtained with 100 subspace iterations. We can see that the model can construct effective algorithms for operators with a wide range of initial eigenvalue gaps. Furthermore, the learned algorithms bring larger improvements as the initial eigenvalue gap becomes smaller. This shows the effectiveness of our model to be applied to realistic settings, where the initial eigenvalues have small gaps.

I.2 Preconditioned Linear Systems

Evaluation on synthetic diagonal matrices. Following the results in Section 5.2, we present evaluation results on synthetic diagonal matrices $\mathbf{X} = \mathrm{diag}(\boldsymbol{\lambda})$ with different initial condition numbers, and we evaluate the normalized residual of learned preconditioning algorithm r_{nn} defined in (20) compared to the residual of standard Richardson iteration defined in (21) in logarithm scale. This effectively measures the improvement of the condition number of the preconditioned operator.

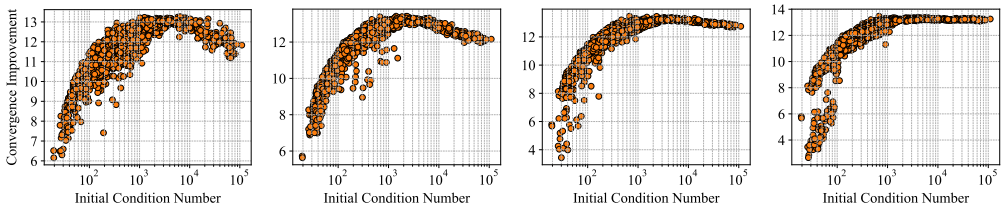


(a) Top-k degree polynomial by backbone model.

(b) Eigenvalue gap improvement of learned polynomial.

Figure 15: Eigenvalue gap improvement of learned polynomial preconditioning algorithms, compared to the standard subspace iteration preconditioner. *Left*: degree-k polynomial obtained by the first k layer of the backbone model after pre-training. *Right*: learned algorithm (after post-training) using spectral probe input obtained with 100 subspace iterations.

As shown in Figure 16, we plot the condition number improvement of each synthetic matrix, corresponding to the initial condition number of the operator. We can see that the learned algorithm consistently achieves improvement over Richardson iteration, where the algorithm has larger improvement when the initial condition number is larger. Furthermore, we observe that as the spectral probe input to the neural network becomes more accurate (by using more subspace iterations), the performance improvement becomes higher and more robust. This result shows the effectiveness and robustness of the neural network engine in generating effective preconditioning algorithms for operators of a wide range of condition numbers.



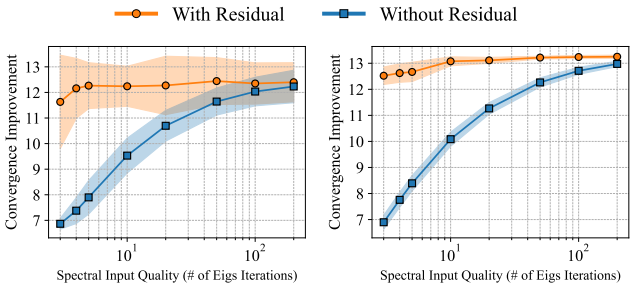
(a) 1 Subspace Iterations (b) 5 Subspace Iterations (c) 10 Subspace Iterations (d) 50 Subspace Iterations

Figure 16: Condition number improvement of the preconditioning algorithm generated by the neural network for synthetic linear systems with different initial condition numbers, given different spectral probe quality. The input spectral probe is obtained using subspace iteration; more iterations result in a higher-quality spectral probe.

Roles of Spectral Residual Features in Discovering Algorithms. Following the results in Section 6, we provide complementary results on synthetic matrices, demonstrating the role of residual features in the learned algorithm. As shown in Figure 17, we can see that the learned algorithm without residual features achieves significantly worse convergence improvement compared to using residual features, and the performance gap becomes larger as the spectral probe input quality becomes worse. Conversely, as the spectral input quality becomes better, the residual features becomes smaller, making the generated algorithm highly effective, regardless of the use of residuals.

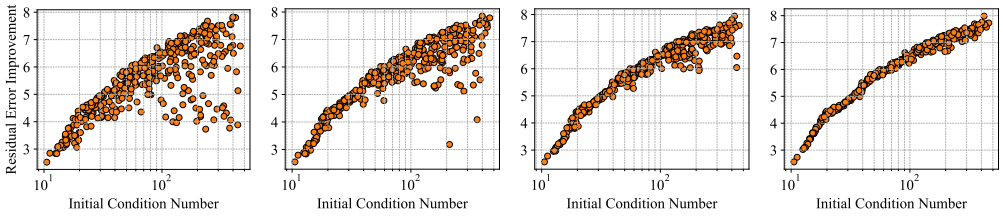
I.3 Approximating Matrix Functions

To comprehensively analyze the performance of algorithms constructed by the AUTOSPEC framework, we use synthetic diagonal matrices $\mathbf{X} = \mathrm{diag}(\boldsymbol{\lambda})$ with different dimensions and condition number, and we evaluate the worst-case approximation error of the learned algorithm compared to Neumann series approximation of inverse square root, defined in (23). We measure the ratio of errors in logarithms.



(a) dim = 500, $\boldsymbol{\kappa} \in [1e2, 5e3]$ (b) dim = 2000, $\boldsymbol{\kappa} \in [2e3, 1e5]$

Figure 17: Comparison of generated algorithms using eigenvalue estimation residual features (Ritz value residual norms) versus without residuals, under different spectral probe input quality (number of subspace iterations). Performance is measured by the ronvergence rate improvement relative to Richardson iteration in logarithms, on synthetic matrices with varying condition numbers κ.



(a) 1 Subspace Iterations (b) 5 Subspace Iterations (c) 10 Subspace Iterations (d) 50 Subspace Iterations

Figure 18: Residual improvement of the algorithm generated by the neural network that approximates the inverse square root for synthetic matrices with different initial condition numbers, given different spectral probe quality. The input spectral probe is obtained using subspace iteration; more iterations result in a higher-quality spectral probe.

As shown in Figure 18, we can see that the learned algorithm consistently improves the residual compared to the Neumann series approximation. Furthermore, we observe that the residual improvement increases as the initial condition number of the operator increases, and that using spectral probe with better quality results in the generation of more robust and effective approximation algorithms.

J Evaluating Properties of Learned Algorithms

Here, we examine whether the learned polynomial preconditioning algorithms have properties analogous to Chebyshev polynomials. Specifically, among all monic polynomials (polynomials with leading coefficient 1) of degree d, the scaled Chebyshev polynomial $\tilde{T}_d(t) = 2^{1-d}T_d(t)$ minimizes the infinity norm on the interval $[-1, 1]$.

After mapping the interval to $[-1, 1]$, minimax optimality is characterized by the following properties:

1. **Equioscillation:** The polynomial must oscillate between two bounds $\pm L$, achieving its maximum magnitude at $d + 1$ distinct points (the Chebyshev alternation theorem).
2. **Minimal Norm Bound:** For a polynomial with leading coefficient K, the maximum absolute value on $[-1, 1]$ is lower-bounded by $|K| \cdot 2^{1-d}$.

To evaluate whether our learned algorithms satisfy these properties, without prior knowledge of the spectral bounds, we formulate a window-discovery optimization problem. We seek an affine transformation $\theta(t) = c_1 t + c_2$ that maps the canonical domain $t \in [-1, 1]$ to the optimal interval of the learned polynomial. We define the verification objective as minimizing the absolute gap from the minimax bound on the interval $[-1, 1]$:

$$\mathcal{L}(c_1, c_2) = \frac{\max_{t\in[-1,1]} |P(c_1 t + c_2)|}{|A(c_1)|}, \tag{29}$$

where $A(c_1)$ represents the leading coefficient of the transformed polynomial. Given the recurrence relations used to construct $P(\lambda)$, the leading coefficient scales as $A(c_1) = a_d c_1^d$, where a_d is the leading coefficient of $P(\lambda)$ derived from the product of the recurrence scalars β_k. If the learned algorithm produces a true Chebyshev polynomial, there exist window parameters (c_1^*, c_2^*) such that $\mathcal{L} = 2^{1-d}$. Conversely, $\mathcal{L} \gg 2^{1-d}$ indicates that the polynomial is suboptimal (i.e., its maximum value is larger than the theoretical minimum for its degree and leading coefficient). In practice, we approximate the maximum using 200 equally spaced points in $[-1, 1]$. We solve this non-convex optimization problem using a grid search initialization followed by a Nelder-Mead simplex search.

Table 10 reports the resulting minimax optimality measures (29) for AUTOSPEC degree-21 polynomials, alongside a random polynomial baseline and the theoretical Chebyshev minimum.

Table 10: Minimax optimality measure (29), where lower is more Chebyshev-like. "Random" uses randomly generated polynomial coefficients. The Chebyshev column gives the theoretical minimum.

Matrix	`thermal1`	`thermal2`	`SiO2`	`CO`	Random	Chebyshev
Normalized Maximum Norm (↓)	1.700×10^{-6}	1.700×10^{-6}	1.692×10^{-6}	1.737×10^{-6}	0.659	9.537×10^{-7}

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction clearly stated the motivation and goal of the proposed method. It also describes the scope of application and potential influence of the method on related fields.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations are clearly discussed in Section 6.5 of the paper.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: The proofs of claims on the expressive power of the neural engine and ability to recover classical algorithms are described in Appendix F.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides algorithmic details of the proposed model structure and training configurations. The data used for evaluation (mainly SuiteSparse matrix collection) are publicly available and described in related sections.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The codebase for reproducing results in the paper has been open-sourced.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Details are provided in Appendix D.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: For main results such as Figure 2, experiments are conducted with multiple seeds, and error bars are provided in the convergence curves.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)

- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Compute resources used are outlined in Appendix D.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Potential impact of our work is discussed in Section 6.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The experiments in the paper uses SuiteSparse Matrix Collection for evaluation, which is cited in Section 5.1.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The asset introduced in this paper is the codebase for conducting the experiments. The codebase has been open-sourced.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: the paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: LLM are used solely for writing, editing and formatting of the paper.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.