MATRL: PROVABLY GENERALIZABLE ITERATIVE ALGORITHM DISCOVERY VIA MONTE-CARLO TREE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Iterative methods for computing matrix functions have been extensively studied and their convergence speed can be significantly improved with the right tuning of parameters and by mixing different iteration types Higham & Schreiber (1990). Hand-tuning the design options for optimal performance can be cumbersome, especially in modern computing environments: numerous different classical iterations and their variants exist, each with non-trivial per-step cost and tuning parameters. To this end, we propose **MatRL** – a reinforcement learning based framework that automatically discovers iterative algorithms for computing matrix functions. The key idea is to treat algorithm design as a sequential decision-making process. Monte-Carlo tree search is then used to plan a *hybrid* sequence of matrix iterations and step sizes, tailored to a specific input matrix distribution and computing environment. Moreover, we also show that the learned algorithms provably generalize to sufficiently large matrices drawn from the same distribution. Finally, we corroborate our theoretical results with numerical experiments demonstrating that MatRL produces algorithms that outperform various baselines in the literature.

1 INTRODUCTION

Matrix functions are everywhere - ranging from classical applications in control theory Denman & Beavers Jr (1976), high-dimensional ODEs Higham (2008), theoretical particle physics Chen & Chow (2014), Lin et al. (2009), Markov models Waugh & Abel (1967), to some recent applications in machine learning, e.g. covariance pooling Li et al. (2018), Wang et al. (2020), Song et al. (2021), graph signal processing Defferrard et al. (2016), Maskey et al. (2023), contrastive learning Richemond et al. (2023), and optimizer design Gupta et al. (2018), Jordan et al. (2024), Ahn & Xu (2025). For more applications see Higham (2008) and references therein. It isn't a surprise that computing matrix functions in a fast, precise, and stable manner has received numerous attention and many have tried to develop better algorithms with guarantees that it will work in some sense.

Iterative algorithms to compute matrix functions are particularly attractive in modern applications as they can avoid computing the matrix function directly using the singular value decomposition (SVD). Further, termination criteria can be chosen based on the needs of the application, which can lead to faster and more stable algorithms. Additionally, these algorithms are differentiable, leading to potential applications in auto-differentiation based settings Song et al. (2022). From an algorithm design perspective, it would be ideal to find an algorithm that is computationally efficient, numerically stable and has faster convergence guarantees. One can imagine using different iterations from different algorithms at each step (e.g. mixing the algorithms) and specifically tuning the parameters jointly to accelerate the algorithm. However, due to the large search space (see Section 2.1 and references therein to see a vast array of existing iterative algorithms) and highly sensitive and non-trivial per iteration costs in modern computing environments, handcrafting the ideal algorithm is tedious and impractical.

A motivating experiment is presented in Table 1. In single precision, the cost of computing a 4096×4096 inverse is roughly five times that of a matrix multiply on both CPU and GPU. However, switching to double precision more than halves this ratio – dropping to about 2.9 on CPU and 1.7 on

Table 1: Computation times for matrix inverse and multiplication (4096×4096) on CPU and GPU in single (float32) and double (float64) precision. The right-most column shows the ratio of inverse time to matmul time. All computations done using PyTorch on an RTX A6000 GPU.

Device	Precision	Inverse (ms)	MatMul (ms)	Inv : MatMul ratio
CPU	float32	251.2 ± 8.7	48.8 ± 1.8	5.15
	float64	356.6 ± 16.3	122.9 ± 4.6	2.90
GPU	float32	28.3 ± 0.5	5.6 ± 0.1	5.05
	float64	423.9 ± 6.9	252.6 ± 6.4	1.68

GPU – because matrix multiplies become comparatively more expensive. This pronounced change in relative costs with precision underscores the need for automatic algorithm discovery, as the optimal sequence of matrix iterations will depend sensitively on both hardware and numerical precision. Besides the change in relative speeds, the gains from switching to GPU turns into a slowdown with double precision for both inverse and matmuls. This is due to the significantly lower emphasis on high precision compute capability in modern GPUs.

We propose an automated solution based on Monte-Carlo tree search to decide which combination of iterations and parameters one should use given a desiderata of the user. Our solution assumes that the matrix of interest is sampled repeatedly from a certain symmetric random matrix distribution, and we want to find a good algorithm for that matrix distribution. The main idea is that iterative algorithms can be understood as a sequential decision-making process: at each step, one should choose which iteration to use and which parameters to use in that iteration. This corresponds to choosing actions in decision making, where the choice leads to the next step. At each step we get a certain reward signal based on the given desiderata, so that we can evaluate whether the action was worth it or not. Finding a good algorithm mounts to finding a good planning strategy for the given environment. Specifically,

- We propose **MatRL** (Algorithm 1), an automated algorithm searching scheme based on Monte-Carlo tree search.
- The algorithms we find are faster than existing baselines, and even faster than implementations in torch.linalg. Moreover, the found algorithms differ with problem sizes, computation environment, and precision (Section 5.1), meaning that MatRL adapts to different enviroments with ease.
- We have a guarantee using random matrix theory that the found algorithm will generalize to different matrices drawn from the same distribution (Section 4), and matrices drawn from the distribution with identical limiting eigenvalues.

The paper is organized as the following: in Section 2 we discuss relevant background on iterative matrix function computation algorithms, Monte-Carlo tree search and learning algorithms via RL. Next we describe our environment in Section 3 by describing how the states, actions, state transition, and reward signals are defined. In Section 4, we provide the generalization guarantee stemming from limiting distribution of the spectrum. We show experimental results in Section 5 showing the performance and adaptivity of MatRL, and conclude the paper in Section 6.

2 Background

2.1 Iterative Methods to Compute Matrix Functions

The basic idea of obtaining an iteration to compute matrix functions is using Newton's method Higham (2008). For instance, say we want to compute the matrix square root. We would like to compute the root of the function $f(X) = X^2 - A$, hence Newton's method can be written as

$$X_{k+1} = X_k - f'(X_k)^{-1} f(X_k) = X_k - \frac{1}{2} X_k^{-1}(X_k^2 - A) = \frac{1}{2}(X_k + X_k^{-1} A). \tag{1}$$

Newton's method uses a first-order approximation of f at X_k: we may use higher order approximations to obtain Chebyshev method Li et al. (2011) or Halley's method Nakatsukasa et al. (2010),

Guo (2010). These higher-order methods converge cubically with appropriate initialization, but each iteration is slower than Newton's method. We could also use inverse-free methods such as Newton-Schulz and its variants Higham (2008), Higham (1997), which approximates X_k^{-1} with a polynomial of X_k.

Appropriate scaling and shifting of the spectrum to yield faster convergence has been a popular idea Nakatsukasa et al. (2010), Byers & Xu (2008), Byers (1987), Chen & Chow (2014), Iannazzo (2003), Hoskins & Walton (1979), Pan & Schreiber (1991). The intuition is that by introducing additional parameters and solving an optimization problem on the spectrum, we can find a sequence of optimized parameters that depend on the spectrum of A, the matrix that we would like to compute matrix function. For instance, Chen & Chow (2014) finds a cubic function $h : [0, 1] \to [0, 1]$ that maximizes $h'(0)$ to find better scaling of Newton-Schulz iteration. One drawback of these approaches is that in many cases we need to compute smallest / largest eigenvalues of the matrix Byers & Xu (2008), Chen & Chow (2014), Pan & Schreiber (1991), Nakatsukasa et al. (2010), Hoskins & Walton (1979), which may be expensive to compute. In this case we use approximations of the smallest eigenvalue, such as $1/\|A^{-1}\|_F$. Another drawback is that each new scaling scheme needs a complicated mathematical derivation and proof.

Naively applying Newton's method can be unstable for computing matrix square-root and p-th root. To ensure stability, we can introduce an auxillary variable Y_k and simultaneously update X_k and Y_k Higham (1997), Iannazzo (2006). We will refer to such iterations as coupled iterations. Coupled iterations are obtained by manipulating the formula so that we do not have A in the iteration. Going back to Newton's method for computing square roots in Eq. (1), we can introduce an auxillary variable $Y_k = A^{-1}X_k$ and initialize $X_0 = A, Y_0 = I$ to obtain the coupled iteration known as Denman-Beavers iteration Denman & Beavers Jr (1976),

$$\begin{cases} X_{k+1} = \frac{1}{2}(X_k + Y_k^{-1}) \\ Y_{k+1} = \frac{1}{2}(Y_k + X_k^{-1}). \end{cases} \tag{2}$$

Using perturbation analysis, Higham (1997) shows that the iteration in Eq. (1) is unstable when the condition number $\kappa(A) > 9$, whereas the iteration in Eq. (2) is stable.

Iterative algorithms are not limited to Newton's method. We may have fixed-point iterations such as Visser iteration Higham (2008), where we iteratively compute

$$X_{k+1} = X_k + \alpha_k(A - X_k^2),$$

to compute matrix square root. We may also use higher-order rational approximations of the function of interest to obtain algorithms that converge in only a few steps Nakatsukasa & Freund (2016), Gawlik (2019), and with sufficient parallelization they can be faster than existing methods.

2.2 Monte-Carlo Tree Search

Suppose we have a deterministic environment $\mathcal{E}$ which is defined by a 5-tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{T}, r, t)$. $\mathcal{S}$ denotes the set of states, $\mathcal{A}$ denote the set of actions that one can take in each state, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ gives how state transition occurs from state s when we apply action a, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ gives the amount of reward one gets when we do action a at state s, and $t : \mathcal{S} \to \{0, 1\}$ denotes whether the state is terminal or not. Monte-carlo tree search enables us to find the optimal policy Browne et al. (2012) $\pi : \mathcal{S} \to \mathcal{A}$ that decides which action to take at each state to maximize the reward over the trajectory that π generates.

The basic idea is to traverse over the "search tree" in an asymmetric manner, using the current value estimation of each state. Each node of the search tree has a corresponding state s, its value estimation V_s, and visit count N_s. The algorithm is consisted of four steps, the selection step, the expansion step, the simulation step, and the backpropagation step.

During the selection step, the algorithm starts at the root node s_0 and selects the next node to visit using $V_{\mathcal{T}(s,a)}$ and $N_{\mathcal{T}(s,a)}$. One widely-used method is using the upper confidence bounds for trees (UCT) Kocsis & Szepesvári (2006). At node s, the UCT is defined by

$$V_{\mathcal{T}(s,a)} + C\sqrt{\frac{\log(N_s)}{N_{\mathcal{T}(s,a)}}},$$

where C determines the exploration-exploitation tradeoff. The algorithm selects the next state $\mathcal{T}(s, a)$ that maximizes UCT, and such selection continues until the algorithm meets a node of which not all child nodes have been visited.

In the expansion step, the algorithm adds a child node that has not been visited to the search tree.

In the simulation step, the algorithm starts from the added node in the expansion step and uses a default policy to generate a trajectory until they meet a termination criterion. The default policy can either be a random policy or handcrafted heuristics Chaslot et al. (2008).

Finally, in the backpropagation step, each value estimation V_s on the trajectory is updated.

Monte-Carlo tree search gained its popularity to obtain strategies for games such as Tictactoe Veness et al. (2011) or Go Silver et al. (2017), Hoock et al. (2010), as well as real-time stategic games Soemers (2014). Not only that, the algorithm was also applied to combinatorial optimization problems Sabharwal et al. (2012), Rimmel et al. (2011), symbolic regression Kamienny et al. (2023), and complex scheduling problems Chaslot et al. (2006), Matsumoto et al. (2010), Li et al. (2021) - which is most relevant to our work.

2.3 Automated Algorithm Discovery

We are not the first to discover algorithms using ideas from sequential decision-making. RL-based approaches which parametrize the policy as a neural network have proven to be successful: Li & Malik (2017) learns to optimize neural networks using an RL framework. In their framework, the states consist of past variables x_i, past gradients $\nabla f(x_i)$, and past objectives $f(x_i)$, and the policy aims to learn appropriate Δx. Fawzi et al. (2022) used reinforcement learning to find faster matrix multiplication algorithms, and Mankowitz et al. (2023) finds faster sorting algorithms. Khodak et al. (2024) has a similar flavor with our work, where they use contextual bandits to optimize relaxation parameters in symmetric success-over-relaxation.

3 MatRL: Iterative Matrix Function Algorithm Search via RL

3.1 Objective

Let $A \sim \mathcal{D}$, where $\mathcal{D}$ is a symmetric random matrix distribution defined in $\mathbb{R}^{n\times n}$ and $f : \mathbb{R} \to \mathbb{R}$ is the function that we would like to compute. We follow the definition of matrix functions in Higham (2008). For a symmetric matrix A, when we diagonalize $A = UDU^T$ for an orthogonal matrix U, $f(A) = Uf(D)U^T$ where $f(D)$ applies f to the diagonal entries of D. Denote $f_1(X, Y, A, a_1), f_2(X, Y, A, a_2), \cdots f_m(X, Y, A, a_m)$ as m choices of iterations that we can use, $a_j \in \mathbb{R}^{n_j}$ as tunable parameters for each f_j, and T_j the wall-clock time needed to run f_j. We use two variables (X, Y) as input to accomodate coupled iterations, and Y is not used for iterations that are not coupled. For instance, for $f = \sqrt{\cdot}$, $f_1(X, Y, a_1)$ can be the scaled Denman-Beavers iteration

$$f_1(X, Y, A, a_1) = \Big(\frac{1}{2}(a_{11}X + (a_{12}Y)^{-1}), \frac{1}{2}(a_{12}Y + (a_{11}X)^{-1})\Big),$$

whereas $f_2(X, Y, A, a_2)$ can be the scaled Visser iteration

$$f_2(X, Y, a_2) = \Big(a_{21}X + a_{22}(A - X^2), Y\Big). \tag{3}$$

Here, n_j denotes the number of tunable parameters in iteration f_j. Also, assume the error tolerance ϵ_{tol} is given.

Now, we specify the class of matrix iterations and the custom loss function $\mathcal{L} : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times n} \to \mathbb{R}$ that determines the termination condition. For this we define congruence invariant matrix functions.

Definition 1. *(Congruence Invariant Diagonal Preserving) Let $f : (\mathbb{R}^{n\times n})^k \to \mathbb{R}^{m\times m}$ a matrix function that takes k matrices as input and outputs a matrix. If $f(QX_1Q^T, QX_2Q^T, \cdots QX_kQ^T) = Qf(X_1, X_2, \cdots X_k)Q^T$ for all orthogonal Q, f is congruent invariant. If $f(X_1, X_2, \cdots, X_k)$ is diagonal for diagonal $X_1, X_2, \cdots X_k$, f is diagonal preserving. For functions that take matrix tuple as an input and outputs a matrix tuple, $F : (\mathbb{R}^{n\times n})^k \to (\mathbb{R}^{m\times m})^l$, we call F congruent invariant diagonal preserving if $F = (f_1, f_2, \cdots f_l)$ and all f_i are congruent invariant and diagonal preserving for $i \in [l]$.*

We limit the matrix iterations and the loss function to be congruent invariant functions, regarding A also as an input. For instance, the Newton-Schulz iteration to compute matrix inverse Pan & Schreiber (1991) $X_{k+1} = 2X_k - X_kAX_k$, $f(X, A) = 2X - XAX$ is a congruence invariant function and the iteration falls into our framework. Such limitation will enable us to understand actions and losses as functions of the spectrum of X, Y, and A, and as we will see in the next section, it will enable us to see the whole environment only as a function of the spectrum.

Our objective is given a sample A from $\mathcal{D}$, finding a sequence of iterations and coefficients $f_{t_1}(\cdot, a_{t_1}), f_{t_2}(\cdot, a_{t_2}), \cdots f_{t_N}(\cdot, a_{t_N})$ that is a solution to

$$\max_{N, t_i \in [m], a_{t_i} \in \mathbb{R}^{n_i},\ i \in [N]} \quad -\sum_{i=1}^{N} T_{t_i} \quad \text{subject to} \quad \mathcal{L}(X_{N+1}, A) \leq \epsilon_{\text{tol}}, \tag{4}$$

where $X_0 = A, Y_0 = I$ or $X_0 = I, Y_0 = A$ depending on f, $(X_{k+1}, Y_{k+1}) = f_{t_k}(X_k, Y_k, a_{t_k})$ and N is also optimized. Note that similar ideas have also appeared in optimal control Evans (1983) in the context of minimum-time control problems.

The optimal solution of Eq. (4) naturally corresponds to finding the optimal iterative algorithm

$$X_{k+1}, Y_{k+1} \leftarrow f_{t_k}(X_k, Y_k, a_{t_k}), \quad k \in [N].$$

that arrives as $\mathcal{L}(X, A) \leq \epsilon_{\text{tol}}$ as fast as it can.

3.2 The Environment

Here we elaborate how we formulate the problem in Eq. (4) to a sequential decision-making problem by describing $\mathcal{E} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, r, t)$, the 5-tuple specified in Section 2.2.

The simplest way to define the state and action variables is by setting each state as a tuple (X, Y) that corresponds to the matrices (X_k, Y_k), and setting each action as a $n_j + 1$ - tuple $(j, k_1, k_2, \cdots k_{n_j})$, where $j \in [m]$ denotes the iteration f_j and $k_1, k_2, \cdots k_{n_j}$ denotes the parameters for f_j. The state transition $\mathcal{T}$ simply becomes

$$\mathcal{T}(X, Y, j, k_1, k_2, \cdots k_{n_j}) = f_j(X, Y, k).$$

At each transition, we get rewarded by $-T_j$, the negative time needed to run the iteration. The terminal state is when $\mathcal{L}(X, A) \leq \epsilon_{\text{tol}}$. Our environment stems from this basic formulation, but we have important implementation details that we elaborate below.

Spectrum as state variables Having matrices each state can consume a lot of memory, and state transition may be slow. Instead, we use (s_1, s_2), where $s_1, s_2 \in \mathbb{R}^n$ corresponds to the eigenvalues of X, Y. Such parametrization is justified by the fact that both the transition $f_k(X, Y, A, a) = (X', Y')$ and the termination criteria $\mathcal{L}$ can be expressed only using the spectrum for our iterations of interest.

The core intuition is that when we write $A = UD_AU^T$, if $X = UD_XU^T$ and $Y = UD_YU^T$ for the same U and diagonal D_A, D_X, D_Y, the next states X', Y' are similar to A. Moreover, we can see that the spectrum of X', Y' only depends on the spectrum of X, Y and A. Induction finishes the proof. The similarity result and the congruent invariance of $\mathcal{L}$ shows that the termination criteria only depends on the spectrurm. We defer the proof to Section A.

Decoupled actions We have $n_j + 1$ tuple of possible actions each state. As they are continuous variables, the possible action space at the step becomes huge. To mitigate this, we decouple the state transition $\mathcal{T}$ into $n_j + 1$ stages: at stage 1, 2, $\cdots n_j$, only parameters $k_1, \cdots k_{n_j - 1}$ are chosen. At state $n_j + 1$, we get rewarded by $-T_j$, choose next iteration, and state transition happens.

Dealing with coupled iterations For computing matrix roots, some iterations are coupled (e.g., Denman-Beavers), while others like the scaled Visser iteration are not, making it challenging to mix them directly. Applying uncoupled iterations alone can break essential relationships (e.g., $Y_kX_k^{-1} = A$ for Denman-Beavers), potentially leading to incorrect results. To address this, we propose either augmenting uncoupled steps with coupled ones or tracking a boolean flag IsCoupled that governs when and how to restore consistency between variables before proceeding. See Section C for a detailed explanation.

3.3 THE SEARCHING STRATEGY

Here we describe the details of Monte-Carlo tree search in the predescribed environment $\mathcal{E}$. Let's note $n, v : \mathcal{S} \to \mathbb{N}$ the number of child nodes and visit count of that node, respectively. If all parameters $j, k_1, \cdots k_{n_j}$ are chosen and the state is ready for state transition, we call the state *transitionable*. We use progressive widening Couëtoux et al. (2011) to deal with the continuous parameter space: during the selection stage, if the node is transitionable and hasn't visited all possible children, or the number of child nodes $n(s) \leq Cv(s)^\alpha$ for some hyperparameters C, α, we go to the expansion step. If not, we choose the next node with UCT Kocsis & Szepesvári (2006). In the expansion stage, we add a child node. Choosing the iteration is discrete and we choose them depending on IsCoupled flag. To choose the parameters, we first sample randomly for E steps, then jitter around the best parameter found. In the rollout stage, we have predetermined baseline algorithms and run one of them to estimate the value of the state. At last, we backpropagate by using the Bellman equation

$$V_s = \max_{a \in \mathcal{A}} V_{\mathcal{T}(s,a)} + r(s, a).$$

Our search method is summarized in Algorithm 1. Details on the parameters for each experiment and a thorough description of Algorithm 1 can be found in Section C.

Algorithm 1 MatRL: Monte-Carlo Tree Search for Algorithm Discovery

Input: $C, \alpha, \epsilon_{\text{tol}}, E, T, RolloutList, \mathcal{L}, A \sim \mathcal{D}$
Initialize $c, n, t, cp \leftarrow 0, V \leftarrow -\text{INF}, cp[s_{root}] \leftarrow 1$, bestpath, bestrollout $\leftarrow 0$ // Each correspond to number of children, visit count, Transitionable, IsCoupled, and value estimation.
for $i = 1$ to T **do**
  $s \leftarrow s_{root}$
  while $Expandable(s)$ == False and $\mathcal{L}(s) \leq \epsilon_{\text{tol}}$ **do**
    // Expandable if s is transitionable and has a child node yet visited, or $c(s) \leq Cn(s)^\alpha$
    $s \leftarrow Best_{UCB}(s)$
  end while
  $s \leftarrow ExpandNode(s)$ // Here we expand after we look at $cp(s)$
  $r \leftarrow SampleRolloutList()$ // Here we sample from RolloutList, the baselines selected for rollout. If the baseline is coupled but $cp(s) = False$, we attach an additional coupling step at the front
  $s \leftarrow r(s)$
  bestpath, bestrollout $\leftarrow backpropagate(s)$ // Here we use bellman equation. If $V(s_{root})$ was updated, update bestpath and bestrollout
end for
return bestpath $\oplus$ bestrollout

4 RANDOM MATRICES AND GENERALIZATION GUARANTEES

The objective in Eq. (4) aims to find the optimal algorithm for a given matrix A sampled from $\mathcal{D}$. Here we show that under certain assumptions, the found iterative algorithm has a sense of generalization capability to a different matrix distribution $\mathcal{D}'$ with the same limiting distribution. Two main ideas for the proof is: first, the loss curve only depends on the spectrum. Second, if the limiting distributions are identical for $\mathcal{D}$ and $\mathcal{D}'$, the sampled matrices' spectrum will be similar if the matrix is sufficiently large. Hence, the loss curve will be similar for two matrices $X \sim \mathcal{D}$, $Y \sim \mathcal{D}'$ for sufficiently large matrices, and if the algorithm works well for X, it will work well for Y. The specific guarantee that we have is in Proposition 1. The proof is deferred to Section A.

Proposition 1. *(Generalization of the discovered algorithm) Say we have a sequence of symmetric random matrix distributions $\mathcal{P}_m, \mathcal{Q}_m$ defined in $\mathbb{R}^{m \times m}$, and denote random matrices sampled from $\mathcal{P}_m$, $\mathcal{Q}_m$ as X, Y. Let the empirical eigenvalue value distribution of $X \sim \mathcal{P}_m, Y \sim \mathcal{Q}_m$ be $\mu_m(X), \nu_m(Y)$, and their support be S_m, S'_m, respectively. Now, suppose*
(i) (Identical limiting distribution)

$$\mathbb{P}(\mu_m(X) \Rightarrow \mu^*) = \mathbb{P}(\nu_m(Y) \Rightarrow \mu^*) = 1,$$

i.e. both $\mu_m(X)$ and $\nu_m(Y)$ converges weakly to a common distribution μ^* with probability 1.
(ii) (Interval support of the limiting distribution) The support of μ^* is an interval $[a, b]$.

(iii) (Convergence of support) We have

$$\lim_{m\to\infty} \mathbb{P}(S_m \subseteq [a-\epsilon, b+\epsilon]) = \lim_{m\to\infty} \mathbb{P}(S'_m \subseteq [a-\epsilon, b+\epsilon]) = 1,$$

for all $\epsilon > 0$.
With the assumptions above, let f *be the matrix function we would like to compute,* f_k^* *the step* k *transformation of eigenvalues of the algorithm found by Algorithm 1, and* $L : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *be the loss. Assume* f, f_k^*, L *are continuous in* $[a-\epsilon_0, b+\epsilon_0]$ *for some* $\epsilon_0 > 0$. *Write the empirical loss of the random matrix* X *as*

$$\mathcal{L}_k(X) = \frac{1}{m}\sum_{i=1}^{m} L(f(\lambda_i), f_k^*(\lambda_i)),$$

where λ_i *are eigenvalues of* X.
Then, there exists $M_{\epsilon,\delta}$ *such that*

$$m \geq M_{\epsilon,\delta} \Rightarrow \mathbb{P}_{X\sim\mathcal{P}_m, Y\sim\mathcal{Q}_m}[|\mathcal{L}_k(X) - \mathcal{L}_k(Y)| < \epsilon] \geq 1-\delta.$$

Essentially, Proposition 1 states that if we find an algorithm $\{f_k^*(X, Y, a_k)\}_{k=1}^N$ using Algorithm 1 and it works well for a certain matrix A, as the loss value $\mathcal{L}_k(X)$ and $\mathcal{L}_k(Y)$ are similar for sufficiently large m, it will work well for any matrix within the same distribution, and also generalize to distributions with the same limiting distribution if m is sufficiently large.

5 Experiments

5.1 Discovered Algorithms

Different matrix functions Here we show loss curves of two different matrix functions, $\text{sign}(A)$ and $A^{1/2}$. Results for inverse and $A^{1/3}$ can be found in Section D. Wishart denotes $A = \frac{X^\top X}{3d} + \epsilon_{\text{stb}} I$ where $X \in \mathbb{R}^{d/4\times d}$, $X_{ij} \sim \mathcal{N}(0,1)$ i.i.d.. $\epsilon_{\text{stb}} = 1e-3$ exists for numerical stability. "Hessian of Quartic" is the indefinite Hessian of a d-dimensional quartic $\sum_i z_i^4/4 - z_i^2/4$ evaluated at a random point $z \sim \mathcal{N}(0, \mathbf{I_d})$. The loss function for matrix sign was $\|X^2 - I\|_F / \|A\|_F$, and the loss function for matrix square root was $\|X^2 - A\|_F / \|A\|_F$. For a complete list of baselines, see Section B.

The time vs loss curves for different matrix functions can be found in Fig. 1. One thing to notice is that for matrix sign function, the algorithm found by MatRL is strictly better than torch.linalg.eigh, and for matrix square root the algorithm returns an approximation within much smaller wall clock time. Depending on applications where exact matrix function is not neceessary, using algorithms found by MatRL can be an appealing choice.

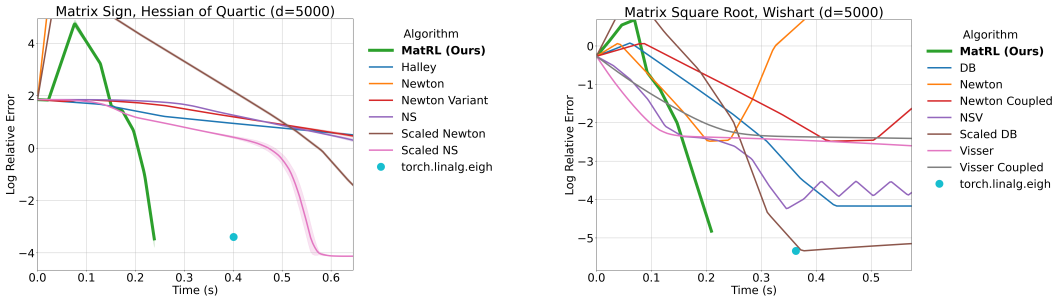


Figure 1: Time versus loss curve for matrix sign and square root. Each iterative algorithms are plotted as baselines, and the wall-clock time to run torch.linalg.eigh is also reported. For matrix sign we have a clear benefit. For matrix square root the found algorithm sacrifices accuracy for time.

The algorithm in Fig. 1, right is described in Algorithm 2. Here we see a tendancy that during the early iterations, the algorithm prefers Visser iteration, whereas for latter steps the algorithm prefers coupled Newton-Schulz iteration. An intuition for such mixed iteration is as follows: when we see

the loss curve, we can notice that the Visser iteration converges fast at first, but it quickly stabilizes and becomes very slow. Hence MatRL learns to use the cheap iterations at first to find a good initial start for NewtonSchulz, and run NewtonSchulz from that initial parameter for faster convergence.

Algorithm 2 Iterative SQRT for Wishart on GPU with $d = 5000$

Input: A
Initialize $X_0 = A, Y_0 = I$
Set $a \leftarrow [0.766, 1.876, 4.251, 0.924, 1], b \leftarrow [2.930, 0.819, 3.223, 0.460, 0.877]$,
// rounded off to three digits
for $i = 1$ to 5 **do**
  $X_i = a_{i-1}X_{i-1} + b_{i-1}(A - X_{i-1}^2)$
  $Y_i = a_{i-1}Y_{i-1} + b_{i-1}(I - Y_{i-1}X_{i-1})$
end for
for $i = 6$ to 8 **do**
  $X_i = 0.5(3I - X_{i-1}Y_{i-1})X_{i-1}$
  $Y_i = 0.5Y_{i-1}(3I - X_{i-1}Y_{i-1})$
end for
return X_8

Different setups yields different algorithms The mixing tendency appears for different matrix functions as well. Preference for a certain iteration over another emerges from two axes, the time it takes for an iteration and how effectively the iteration decreases the loss. Recalling the motivating example in Table 1, different computing environments, e.g. hardware (CPU/GPU), precision (Single/Double), or even the size of the matrix can decide the best algorithm. Here we only demonstrate how precision can change optimal algorithms. A full list of different setups and found algorithms are in Section D.

Algorithm 3 SIGN on GPU (FLOAT)

Input: A
Initialize $X_0 = A$
$a \leftarrow [35.536]$
$b \leftarrow [0.094, 1.607, 1.439, 1.191, 1.029, 1]$
for $i = 1$ to 1 **do**
  $X_i = 0.5(a_{i-1}X_{i-1} + (a_{i-1}X_{i-1})^{-1})$
end for
for $i = 2$ to 7 **do**
  $X_i = 1.5(b_{i-2}X_{i-1}) - 0.5(b_{i-2}X_{i-1})^3$
end for
return X_7

Algorithm 4 SIGN on GPU (DOUBLE)

Input: A
Initialize $X_0 = A$
$a \leftarrow [22.055, 0.244, 0.590, 0.938, 0.998, 1]$

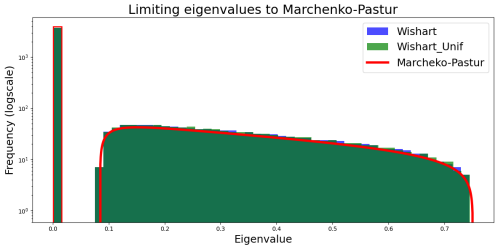
for $i = 1$ to 6 **do**
  $X_i = 0.5(a_{i-1}X_{i-1} + (a_{i-1}X_{i-1})^{-1})$
end for
return X_6

When the precision is double, the relative runtime ratio between inverse and matrix multiplication is not as high as that of single precision. Hence, the model prefers Newton's method more for double precision case, and Algorithm 4 only consists of Newton's iteration whereas Algorithm 3 contains initial Newton's iterations and latter NewtonSchulz iterations.

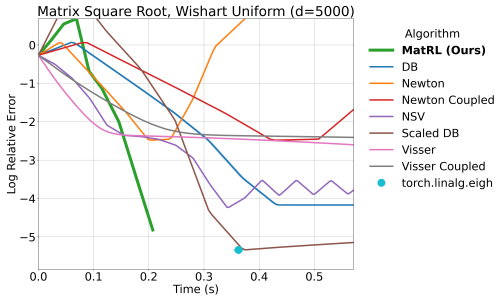
5.2 Generalization

In this section we verify the generalization guarantee that we had in Proposition 1. We test the algorithm learned in Algorithm 2 to a different matrix distribution with the same limiting spectrum, where each entries of X are sampled i.i.d. from Unif$[-\sqrt{3}, \sqrt{3}]$ instead of $\mathcal{N}(0, 1)$. We denote the distribution as WishartUnif. In this case, the two spectrum converges to the same spectrum in distribution due to Marchenko-Pastur (Fig. 2a).

Due to Proposition 1, we expect the learned algorithm in Algorithm 2 to work as well for matrices sampled from WishartUnif. Fig. 2b indeed shows that it is true: when we compare the plot in Fig. 1



(a) Same limiting eigenvalue distribution



(b) Testing Algorithm 2 on a different distribution

Figure 2: How generalization guarantee in Proposition 1 works. Here we have two different random matrix distributions with the same limiting spectrum. Even though they are sampled from different distributions, the limiting spectrum coincide and the learned algorithm generalizes.

and Fig. 2b, the two curves look almost identical - meaning the loss curve for two tests coincide, and generalization indeed happened.

5.3 A Real World Example: CIFAR-10 and ZCA Whitening

A common application of computing matrix square roots, and inverse-square, roots is ZCA whitening of images Bell & Sejnowski (1997). The ZCA whitening, also known as Mahalanobis whitening, decorrelates (or whitens) data samples using the inverse-square root of an empirical covariance matrix. We apply MatRL to learn an algorithm to compute square roots and inverse-square roots simultaneously using coupled iterations on empirical covariances of CIFAR-10 Krizhevsky et al. (2009) images. Specifically, the random input matrix is $\hat{\Sigma} = \frac{1}{n}(X-\mu)^T(X-\mu)$ where $X \in \mathbb{R}^{n\times d}$ is a random batch of n CIFAR-10 images, μ is the average of n images, followed by normalizing by the Frobenius norm and adding $\epsilon_{stb}I$ ($\epsilon_{stb} = 1e{-}3$). The algorithm discovered by MatRL converges significantly faster ($\sim$ 1.94x) than baselines in terms of wall-clock time, see Table 2 for the mean of the wall-clock time of 10 repeated runs. For a visual representation of the result see Section D.

Table 2: Time to reach 1e-4 relative error (in seconds) on CIFAR-10 ZCA whitening.

MatRL (Ours)	DB	Scaled DB	torch.eigh	Newton
0.0551	0.1571	0.1791	0.1071	N/A

6 Conclusion

In this paper we propose MatRL, an MCTS-based automated solution to find iterative algorithms for matrix function computation. We showed that we can generate an algorithm specifically tailored for a specific input matrix distribution and compute environment, and the found algorithm is guaranteed to generalize to different matrix distributions with the same limiting spectrum under certain assumptions. We verify our findings with experiments, showing MatRL found algorithms that are faster than existing baselines and competitive to standard torch library.

Our work has many future directions. One direction is overcoming the current limitations of our work, e.g. extending the result to rectangular and sparse matrices. Another interesting direction is not fixing the matrix iterations beforehand, but making the RL agent discover novel iterations that can ensure stability, such as the Denman-Beavers iteration. At last, automatic discovery of optimal parameters of optimization algorithms such as Muon Jordan et al. (2024) with a different reward (e.g. validation error) could be an interesting direction.

Ethical Statement

The authors have no ethical concerns regarding the paper.

Reproducibility Statement

Also, the experimental results given in Section 5 and Section C are reproducible by running the code in the supplementary material.

References

Kwangjun Ahn and Byron Xu. Dion: A communication-efficient optimizer for large models. *arXiv preprint arXiv:2504.05295*, 2025.

Anthony J Bell and Terrence J Sejnowski. The "independent components" of natural scenes are edge filters. *Vision research*, 37(23):3327–3338, 1997.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Ralph Byers. Solving the algebraic riccati equation with the matrix sign function. *Linear Algebra and its Applications*, 85:267–279, 1987.

Ralph Byers and Hongguo Xu. A new scaling for newton's iteration for the polar decomposition and its backward stability. *SIAM Journal on Matrix Analysis and Applications*, 30(2):822–843, 2008.

Guillaume Chaslot, Steven De Jong, Jahn-Takeshi Saito, and Jos Uiterwijk. Monte-carlo tree search in production management problems. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, volume 9198, 2006.

Guillaume M Jb Chaslot, Mark HM Winands, H Jaap van den Herik, Jos WHM Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357, 2008.

Jie Chen and Edmond Chow. A newton-schulz variant for improving the initial convergence in matrix sign computation. *Preprint ANL/MCS-P5059-0114, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL*, 60439, 2014.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers 5*, pp. 433–445. Springer, 2011.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29, 2016.

Eugene D Denman and Alex N Beavers Jr. The matrix sign function and computations in systems. *Applied mathematics and Computation*, 2(1):63–94, 1976.

Lawrence C Evans. An introduction to mathematical optimal control theory version 0.2. *Lecture notes available at http://math. berkeley. edu/evans/control. course. pdf*, 1983.

Alhussein Fawzi, Giacomo De Palma, Ankit Goyal, Gary Becigneul, Mohammad Barekatain, Sam Bond-Taylor, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022. doi: 10.1038/s41586-022-05172-4.

Evan S Gawlik. Zolotarev iterations for the matrix square root. *SIAM journal on matrix analysis and applications*, 40(2):696–719, 2019.

Chun-Hua Guo. On newton's method and halley's method for the principal pth root of a matrix. *Linear algebra and its applications*, 432(8):1905–1922, 2010.

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

Nicholas J Higham. Stable iterations for the matrix square root. *Numerical Algorithms*, 15:227–242, 1997.

Nicholas J Higham. *Functions of matrices: theory and computation*. SIAM, 2008.

Nicholas J Higham and Robert S Schreiber. Fast polar decomposition of an arbitrary matrix. *SIAM Journal on Scientific and Statistical Computing*, 11(4):648–655, 1990.

Jean-Baptiste Hoock, Chang-Shing Lee, Arpad Rimmel, Fabien Teytaud, Mei-Hui Wang, and Oliver Teytaud. Intelligent agents for the game of go. *IEEE Computational Intelligence Magazine*, 5(4): 28–42, 2010.

WD Hoskins and DJ Walton. A faster, more stable method for computing the pth roots of positive definite matrices. *Linear Algebra and Its Applications*, 26:139–163, 1979.

Bruno Iannazzo. A note on computing the matrix square root. *Calcolo*, 40(4):273–283, 2003.

Bruno Iannazzo. On the newton method for the matrix p th root. *SIAM journal on matrix analysis and applications*, 28(2):503–523, 2006.

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

Pierre-Alexandre Kamienny, Guillaume Lample, Sylvain Lamprier, and Marco Virgolin. Deep generative symbolic regression with monte-carlo-tree-search. In *International Conference on Machine Learning*, pp. 15655–15668. PMLR, 2023.

Mikhail Khodak, Edmond Chow, Maria-Florina Balcan, and Ameet Talwalkar. Learning to relax: Setting solver parameters across a sequence of linear system instances. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. URL `https://arxiv.org/abs/2310.02246`.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Hou-Biao Li, Ting-Zhu Huang, Yong Zhang, Xing-Ping Liu, and Tong-Xiang Gu. Chebyshev-type methods and preconditioning techniques. *Applied Mathematics and Computation*, 218(2):260–270, 2011.

Ke Li and Jitendra Malik. Learning to optimize neural nets. *arXiv preprint arXiv:1703.00441*, 2017.

Kexin Li, Qianwang Deng, Like Zhang, Qing Fan, Guiliang Gong, and Sun Ding. An effective mcts-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem. *Computers & Industrial Engineering*, 155:107211, 2021.

Peihua Li, Jiangtao Xie, Qilong Wang, and Zilin Gao. Towards faster training of global covariance pooling networks by iterative matrix square root normalization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 947–955, 2018.

Lin Lin, Jianfeng Lu, Lexing Ying, Roberto Car, and Weinan E. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. 2009.

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

Sohir Maskey, Raffaele Paolino, Aras Bacho, and Gitta Kutyniok. A fractional graph laplacian approach to oversmoothing. *Advances in Neural Information Processing Systems*, 36:13022–13063, 2023.

Shimpei Matsumoto, Noriaki Hirosue, Kyohei Itonaga, Kazuma Yokoo, and Hisatomo Futahashi. Evaluation of simulation strategy on single-player monte-carlo tree search and its discussion for a practical scheduling problem. In *World Congress on Engineering 2012. July 4-6, 2012. London, UK.*, volume 2182, pp. 2086–2091. International Association of Engineers, 2010.

Yuji Nakatsukasa and Roland W Freund. Computing fundamental matrix decompositions accurately via the matrix sign function in two iterations: The power of zolotarev's functions. *siam REVIEW*, 58(3):461–493, 2016.

Yuji Nakatsukasa, Zhaojun Bai, and François Gygi. Optimizing halley's iteration for computing the matrix polar decomposition. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2700–2720, 2010.

Victor Pan and Robert Schreiber. An improved newton iteration for the generalized inverse of a matrix, with applications. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1109–1130, 1991.

Pierre Harvey Richemond, Allison Tam, Yunhao Tang, Florian Strub, Bilal Piot, and Felix Hill. The edge of orthogonality: A simple view of what makes byol tick. In *International Conference on Machine Learning*, pp. 29063–29081. PMLR, 2023.

Arpad Rimmel, Fabien Teytaud, and Tristan Cazenave. Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows. In *Applications of Evolutionary Computation: EvoApplications 2011: EvoCOMNET, EvoFIN, EvoHOT, EvoMUSART, EvoSTIM, and EvoTRANSLOG, Torino, Italy, April 27-29, 2011, Proceedings, Part II*, pp. 501–510. Springer, 2011.

Ashish Sabharwal, Horst Samulowitz, and Chandra Reddy. Guiding combinatorial optimization with uct. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*, pp. 356–361. Springer, 2012.

Günther Schulz. Iterative berechung der reziproken matrix. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik*, 13(1):57–59, 1933.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Dennis Soemers. Tactical planning using mcts in the game of starcraft. *Master's thesis, Maastricht University*, 2014.

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*, pp. 1115–1123, 2021.

Yue Song, Nicu Sebe, and Wei Wang. Fast differentiable matrix square root. *arXiv preprint arXiv:2201.08663*, 2022.

Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A monte-carlo aixi approximation. *Journal of Artificial Intelligence Research*, 40:95–142, 2011.

Qilong Wang, Jiangtao Xie, Wangmeng Zuo, Lei Zhang, and Peihua Li. Deep cnns meet global covariance pooling: Better representation and generalization. *IEEE transactions on pattern analysis and machine intelligence*, 43(8):2582–2597, 2020.

Frederick V Waugh and Martin E Abel. On fractional powers of a matrix. *Journal of the American Statistical Association*, 62(319):1018–1021, 1967.

A TECHNICAL PROOFS

We first show that the environment can be parametrized by the spectrum.

Proposition A.2. *Let $\{f_k(X, Y, A)\}_{k=1}^m$ is a set of congruence invariant diagonal preserving matrix functions that take X, Y, A as input and outputs (X', Y'), $\mathcal{L}(X, A)$ be a congruence invariant matrix function that takes X, A as input and outputs a scalar, and $X_0, Y_0, \{X_k\}_{k=1}^{m+1}$ satisfy*

$$(X_0, Y_0) = (A, I) \quad or \quad (I, A)$$

and

$$(X_{k+1}, Y_{k+1}) = f_k(X_k, Y_k, A), \quad k \in [m].$$

At last, write $A = UD_AU^T$. Then, the following properties hold:
i) $X_k, Y_k \sim A$ for all $k \in [m+1]$.
ii) Write $X_k = UP_kU^T$, $Y_k = UQ_kU^T$. Then, P_{k+1}, Q_{k+1} depends only on P_k, Q_k and D_A.
iii) The loss $\mathcal{L}(X_k, A)$ only depends on P_k, D_A.

Proof. i) We prove by induction.
If $k = 0$, we know that $X_0, Y_0 = A, I$ or I, A hence they are similar with A.
Say X_t, Y_t are similar with A. Then $X_t = UD_1U^T, Y_t = UD_2U^T$ for diagonal D_1, D_2. Now we can see that

$$f_t(X_t, Y_t, A) = f_t(UD_1U^T, UD_2U^T, UD_AU^T) = Uf_t(D_1, D_2, D_A)U^T = (UD_1'U^T, UD_2'U^T),$$

for some diagonal D_1', D_2'. The second equality comes from congruent invariance, and the third equality follows from diagonal preservence. Hence, $X_{t+1} = UD_1'U^T$ and X_{t+1} and A are similar, $Y_{t+1} = UD_2'U^T$ and Y_{t+1} and A are also similar.
ii) From the proof of i) we know that $f_t(D_1, D_2, D_A) = (D_1', D_2')$, where $D_1, D_2, D_A, D_1', D_2'$ are the spectrum of $X_t, Y_t, A, X_{t+1}, Y_{t+1}$, respectively.
iii) We know X_k and A are similar. Let $X_k = UP_kU^T$ and $A = UD_AU^T$. $\mathcal{L}(UP_kU^T, UD_AU^T) = \mathcal{L}(P_k, D_A)$ by congruent invariance. □

Next we show that all iterations that we deal with in the paper is congruent invariant diagonal preserving. See Section B.5 and Table 7 for the iterations of interest.

Lemma A.1. *Assume $f(X_1, X_2, \cdots X_k)$, $g(X_1, X_2, \cdots X_k)$ are congruent invariant diagonal preserving. Then, $f + g$, fg, f^{-1} are all congruent invariant diagonal preserving.*

Proof. Diagonal preserving is simple, as sum, multiple, inverse of diagonal matrices are diagonal. Let's show congruence invariance.

$$f(QX_1Q^T, \cdots QX_kQ^T) + g(QX_1Q^T, \cdots QX_kQ^T) = Qf(X_1, \cdots X_k)Q^T + Qg(X_1, \cdots X_k)Q^T$$

and

$$Qf(X_1, \cdots X_k)Q^T + Qg(X_1, \cdots X_k)Q^T = Q(f(X_1, \cdots X_k) + g(X_1, \cdots X_k))Q^T.$$

$$f(QX_1Q^T, \cdots QX_kQ^T)g(QX_1Q^T, \cdots QX_kQ^T) = Qf(X_1, \cdots X_k)Q^TQg(X_1, \cdots X_k)Q^T$$

and

$$Qf(X_1, \cdots X_k)Q^TQg(X_1, \cdots X_k)Q^T = Q(f(X_1, \cdots X_k)g(X_1, \cdots X_k))Q^T.$$

$$f(QX_1Q^T, \cdots, QX_kQ^T)^{-1} = (Qf(X_1, \cdots X_k)Q^T)^{-1} = Qf(X_1, \cdots X_k)^{-1}Q^T.$$

□

Proposition A.3. *Say $f(X_1, X_2, \cdots X_k)$ is a rational function of $X_1, \cdots X_k$, i.e. $f(X_1, X_2, \cdots X_k) = P(X_1, \cdots X_k)Q(X_1, \cdots X_k)^{-1}$ for polynomials P, Q. Then f is congruent invariant diagonal preserving.*

Proof. We know that P, Q, and hence PQ^{-1} is congruent invariant diagonal preserving from lemma 1. □

As all iterations in Table 7 or Section B.5 are rational functions, the iterations of interest are congruent invariant diagonal preserving.

At last we show Proposition 1: our found algorithm will generalize to the matrices drawn from the distribution with the same limiting eigenvalue spectrum.

Proposition A.4. *(Generalization of the discovered algorithm, Proposition 1 of the main paper) Say we have a sequence of symmetric random matrix distributions $\mathcal{P}_m, \mathcal{Q}_m$ defined in $\mathbb{R}^{m\times m}$, and denote random matrices sampled from $\mathcal{P}_m$, $\mathcal{Q}_m$ as X, Y. Let the empirical eigenvalue value distribution of $X \sim \mathcal{P}_m, Y \sim \mathcal{Q}_m$ be $\mu_m(X), \nu_m(Y)$, and their support be S_m, S'_m, respectively. Now, suppose*
(i) (Identical limiting distribution)

$$\mathbb{P}(\mu_m(X) \Rightarrow \mu^*) = \mathbb{P}(\nu_m(Y) \Rightarrow \mu^*) = 1,$$

i.e. both $\mu_m(X)$ and $\nu_m(Y)$ converges weakly to a common distribution μ^* with probability 1.
(ii) (Interval support of the limiting distribution) The support of μ^* is an interval $[a, b]$.
(iii) (Convergence of support) We have

$$\lim_{m\to\infty} \mathbb{P}(S_m \subseteq [a-\epsilon, b+\epsilon]) = \lim_{m\to\infty} \mathbb{P}(S'_m \subseteq [a-\epsilon, b+\epsilon]) = 1,$$

for all $\epsilon > 0$.
With the assumptions above, let f be the matrix function we would like to compute, f_k^* the step k transformation of eigenvalues of the algorithm found by Algorithm 1, and $\mathcal{L}$ be the loss. Assume f, f_k^*, L are continuous in $[a-\epsilon_0, b+\epsilon_0]$ for some $\epsilon_0 > 0$. Write the empirical loss of the random matrix X as

$$\mathcal{L}_k(X) = \frac{1}{m}\sum_{i=1}^{m} L(f(\lambda_i), f_k^*(\lambda_i)),$$

where λ_i are eigenvalues of X.
Then, there exists $M_{\epsilon,\delta}$ such that

$$m \geq M_{\epsilon,\delta} \Rightarrow \mathbb{P}_{X\sim\mathcal{P}_m, Y\sim\mathcal{Q}_m}[|\mathcal{L}_k(X) - \mathcal{L}_k(Y)| < \epsilon] \geq 1-\delta.$$

Proof. We write

$$\mathcal{L}^* = \int L(f(\sigma), f_k(\sigma)) d\mu^*(\sigma).$$

We would like to show that for sufficiently large m, $|\mathcal{L}_k(X) - \mathcal{L}^*| < \epsilon/2$ with high probability. As $\mathbb{P}(\mu_m(X) \Rightarrow \mu^*) = 1$, we know that with probability 1,

$$\lim_{m\to\infty} \int f d\mu_m(X) = \int f d\mu^*$$

for all continuous bounded f. We shall extend $L(f(x), f_k(x))$ in a way that it is continuous bounded in $\mathbb{R}$ and the "difference" is small.
First, we know that $L(f(x), f_k(x))$ is continuous in $[a-\epsilon_0, b+\epsilon_0]$. Say

$$A = \max_{x\in[a-\epsilon_0, b+\epsilon_0], y\in[a,b]} |L((f(x), f_k(x))| + |L((f(y), f_k(y))|.$$

Now, choose $\epsilon' = \min\{\epsilon_0, \epsilon/8A\}$ (when $A = 0$ we just have $\epsilon' = \epsilon_0$). With the chosen ϵ', define $\tilde{L}$ as

$$\tilde{L}(x) = \begin{cases} L(f(x), f_k(x)) \quad if \quad x \in [a,b] \\ (x-a+\epsilon')\frac{L(f(a), f_k(a))}{\epsilon'} \quad if \quad x \in [a-\epsilon', a] \\ (-x+b+\epsilon')\frac{L(f(b), f_k(b))}{\epsilon'} \quad if \quad x \in [b, b+\epsilon'] \\ 0 \quad if \quad x \in (-\infty, a-\epsilon'], [b+\epsilon', \infty), \end{cases}$$

which is a bounded continuous function in $\mathbb{R}$. At last, choose M_1 sufficiently large so that $m \geq M_1$ implies

$$|\int \tilde{L} d\mu_m(X) - \int \tilde{L} d\mu^*| < \epsilon/4$$

with probability at least $1-\delta/4$ and

$$\mathbb{P}(S_m \subseteq [a-\epsilon', b+\epsilon']) \geq 1-\delta/4.$$

Such m exists because of assumptions (i) and (iii). Now, we know that

$$\mathcal{L}_k(X) = \int L(f(x), f_k(x))\mu_m(X).$$

Moreover, the function $|\tilde{L}(x) - L(f(x), f_k(x))| \leq A$ for $x \in S_m$. This is because

$$|\tilde{L}(x)| \leq \max_{x\in[a,b]} |L(f(x), f_k(x))|, \quad S_m \subseteq [a-\epsilon', b+\epsilon'] \subseteq [a-\epsilon_0, b+\epsilon_0].$$

As $\tilde{L}(x) - L(f(x), f_k(x)) = 0$ for $x \in [a, b]$, the value

$$|\int \tilde{L} d\mu_m(X) - \int L(f(x), f_k(x)) d\mu_m(X)| \leq 2\epsilon' A \leq \epsilon/4$$

with probability at least $1 - \delta/2$. Hence, when $m \geq M_1$,

$$|\mathcal{L}_k(X) - \int \tilde{L} d\mu^*| = |\mathcal{L}_k(X) - \mathcal{L}^*| < \epsilon/2$$

with probability at least $1 - \delta/2$. We can do the same argument for Y to find M_2. Take $M_{\epsilon,\delta} = \max\{M_1, M_2\}$. Using union bound, we can see the probability that both $|\mathcal{L}_k(X) - \mathcal{L}^*| < \epsilon/2$ and $|\mathcal{L}_k(Y) - \mathcal{L}^*| < \epsilon/2$ happens is at least $1 - \delta$. Hence, $\mathbb{P}_{X\sim\mathcal{P}_m, Y\sim\mathcal{Q}_m}[|\mathcal{L}_k(X) - \mathcal{L}_k(Y)| < \epsilon] \geq 1 - \delta$. □

B List of Used Matrix Iterations

We first present a table that shows different types of baseline algorithms used in the paper with references.

Table 3: List of baselines

Matrix function	List of baselines
Inverse	NS Pan & Schreiber (1991), Chebyshev Li et al. (2011)
Sign	Newton Higham (2008), NS Schulz (1933), ScaledNewton Byers & Xu (2008), ScaledNS Chen & Chow (2014), HalleyNakatsukasa et al. (2010)
Square root	DBDenman & Beavers Jr (1976), NSVHigham (1997)(2.6), Scaled DBHigham (1997), VisserHigham (2008), NewtonHoskins & Walton (1979)
1/3 - root	Iannazzo Iannazzo (2006), Visser Higham (2008), Newton Iannazzo (2006)(1.2)

B.1 Iterative methods associated with inverse

We have two different baselines for inverse. One is Newton's method proposed by Schulz, which is the iteration

$$\textbf{(InvNewton)} \qquad X_{k+1} = 2X_k - X_k A X_k.$$

For an appropriate initialization, the norm $\|I - AX_k\|_2$ will converge quadratically to zero. This is because we can write

$$I - AX_{k+1} = I - 2AX_k + AX_k AX_k = (I - AX_k)^2.$$

Another baseline is applying Chebyshev's iteration to the function $X^{-1} - A$. We have

$$\textbf{(InvChebyshev)} \qquad X_{k+1} = 3X_k - 3X_k AX_k + X_k AX_k AX_k.$$

With similar logic we can obtain $I - AX_{k+1} = (I - AX_k)^3$. Hence at each iteration the error decreases cubically. The drawback is that Chebyshev's method needs at least three matrix-matrix multipications each iteration.

B.2 Iterative methods associated with sign

The simplest method to compute matrix sign is Newton's method, where the iteration is given as

$$\textbf{(SignNewton)} \qquad X_{k+1} = \frac{1}{2}(X_k + X_k^{-1}).$$

The NewtonSchulz variant avoids computing inverse by using the iteration

$$X_{k+1} = \frac{1}{2}(3X_k - X_k X_k^T X_k),$$

hence for symmetric matrices

$$\textbf{(SignNewtonSchulz)} \qquad X_{k+1} = \frac{1}{2}(3X_k - X_k^3).$$

Newton's method has scaled variants, where we do

$$\textbf{(SignScaledNewton)} \qquad X_{k+1} = \frac{1}{2}(\mu_k X_k + (\mu_k X_k)^{-1}),$$

for specific μ_k. Our baseline is the one proposed in Byers & Xu (2008). Here μ_k is defined as the following: we let a, b be constants that satisfy $a \leq \sigma_n \leq \sigma_1 \leq b$ for the singular values of A. Then

$$\mu_0 = \frac{1}{\sqrt{ab}}, \quad \mu_1 = \sqrt{\frac{2}{\sqrt{a/b} + \sqrt{b/a}}}, \quad \mu_k = \sqrt{\frac{2}{\mu_{k-1} + \mu_{k-1}^{-1}}}, k \geq 2.$$

a and b can be obtained by computing $\|A|_2$ and $\|A^{-1}|_2^{-1}$. NewtonSchulz method may also have variants: a recent variant in Chen & Chow (2014) scales each X_k as

$$\textbf{(SignScaledNewtonSchulz)} \qquad X_{k+1} = \frac{3}{2}\rho_k X_k - \frac{1}{2}(\rho_k X_k)^3,$$

where $X_0 = A/\lambda_{|max|}(A)$, $x_0 = \lambda_{|min|}(A)/\lambda_{|max|}(A)$ and

$$\rho_k = \sqrt{\frac{3}{1 + x_0 + x_0^2}}, \quad x_{k+1} = \frac{1}{2}\rho_k x_k(3 - \rho_k^2 x_k^2).$$

Halley's method uses a rational approximation of sign function to compute the matrix sign. The iteration is written as

$$\textbf{(SignHalley)} \qquad X_{k+1} = X_k(a_k I + b_k X_k^2)(I + c_k X_k^2)^{-1},$$

where default Halley's iteration uses $a = c = 3, b = 1$ and the scaled Halley in Nakatsukasa et al. (2010) uses certain optimal coefficients.

Newton variant is essentially a variant of Newton's method where we do

$$\textbf{(SignNewtonVariant)} \qquad X_{k+1} = 2X_k(I + X_k^2)^{-1}.$$

This is the inverse of SignNewton, and X_k converges to $\text{sign}(A)^{-1}$, which is $\text{sign}(A)$ when A is invertible.

B.3 Iterative methods associated with square root

The simplest method in this case is also the Newton's method,

$$\textbf{(SqrtNewton)} \qquad X_{k+1} = \frac{1}{2}(X_k + X_k^{-1}A).$$

The above method can be unstable, which led to the development of coupled iterations. Denman-Beavers iteration uses the following coupled iterationn of X_k and Y_k: Denman-Beavers is initialized with $X_0 = A, Y_0 = I$ and iteratively applies

$$\textbf{(SqrtDenmanBeavers)} \qquad \begin{cases} X_{k+1} = \frac{1}{2}(X_k + Y_k^{-1}) \\ Y_{k+1} = \frac{1}{2}(Y_k + X_k^{-1}). \end{cases}$$

Here, $X_k \to A^{1/2}$ and $Y_k \to A^{-1/2}$. There is a variant of Denman-Beavers that avoids computing matrix inverse - introduced in Higham (1997), the iteration writes

$$\textbf{(SqrtNewtonSchulzVariant)} \qquad \begin{cases} X_{k+1} = \frac{1}{2}(3X_k - X_k Y_k X_k) \\ Y_{k+1} = \frac{1}{2}(3Y_k - Y_k X_k Y_k). \end{cases}$$

Like ScaledNewton, we have a scaled variant of Denman-Beavers. The scaling we use is a variant of Byer's scaling Byers (1987) introduced in Higham (1997). The iteration is given as

$$\textbf{(SqrtScaledDenmanBeavers)} \qquad \begin{cases} \gamma_k = |\det X_k \det Y_k|^{-1/2n} \\ X_{k+1} = \frac{1}{2}(\gamma_k X_k + (\gamma_k Y_k)^{-1}) \\ Y_{k+1} = \frac{1}{2}(\gamma_k Y_k + (\gamma_k X_k)^{-1}). \end{cases}$$

The cost of computing γ_k is negligible when we use decomposition methods such as LU decomposition or Cholesky to compute matrix inverse.

At last, there is the fixed-point iteration, which we will denote as the Visser iteration Higham (2008). The Visser iteration is given as

$$\textbf{(SqrtVisser)} \qquad X_{k+1} = X_k + \frac{1}{2}(A - X_k^2).$$

B.4 ITERATIVE METHODS ASSOCIATED WITH 1/3-ROOT

There are a number of stable methods to compute matrix p-th root (see Iannazzo (2006) for different methods). We use the following method as a baseline: initialize $X_0 = I, Y_0 = A$ and

$$\textbf{(prootIannazzo)} \qquad \begin{cases} X_{k+1} = X_k(\frac{2I+Y_k}{3}) \\ Y_{k+1} = (\frac{2I+Y_k}{3})^{-3} Y_k \end{cases}$$

With this iteration, $X_k \to A^{1/3}$ and $Y_k \to I$. We have Newton's method and Visser's iteration as we had for square root:

$$\textbf{(prootNewton)} \qquad X_{k+1} = (2X_k + X_k A^{-2})/3$$

is the Newton's method, and

$$\textbf{(prootVisser)} \qquad X_{k+1} = X_k + \frac{1}{3}(A - X_k^3)$$

becomes Visser's iteration.

B.5 SUMMARY

We present a table of the matrix functions and iterations that we used.

Table 4: Iterative methods for computing matrix inverse, sign, square root, and 1/3-root.

Method	**Iteration Formula**
	Methods for Inverse
Newton (Schulz)	$X_{k+1} = 2X_k - X_k A X_k$
Chebyshev	$X_{k+1} = 3X_k - 3X_k A X_k + X_k A X_k A X_k$
	Methods for Sign
Newton	$X_{k+1} = \frac{1}{2}(X_k + X_k^{-1})$
NewtonSchulz	$X_{k+1} = \frac{1}{2}(3X_k - X_k^3)$
ScaledNewton	$X_{k+1} = \frac{1}{2}(\mu_k X_k + (\mu_k X_k)^{-1})$
ScaledNewtonSchulz	$X_{k+1} = \frac{3}{2}\rho_k X_k - \frac{1}{2}(\rho_k X_k)^3$
Halley	$X_{k+1} = X_k(a_k I + b_k X_k^2)(I + c_k X_k^2)^{-1}$
NewtonVariant	$X_{k+1} = 2X_k(I + X_k^2)^{-1}$
	Methods for SquareRoot
Newton	$X_{k+1} = \frac{1}{2}(X_k + X_k^{-1}A)$
DenmanBeavers	$\begin{cases} X_{k+1} = \frac{1}{2}(X_k + Y_k^{-1}) \\ Y_{k+1} = \frac{1}{2}(Y_k + X_k^{-1}) \end{cases}$
NewtonSchulzVariant	$\begin{cases} X_{k+1} = \frac{1}{2}(3X_k - X_k Y_k X_k) \\ Y_{k+1} = \frac{1}{2}(3Y_k - Y_k X_k Y_k) \end{cases}$
ScaledDenmanBeavers	$\begin{cases} \gamma_k = \lvert\det X_k \det Y_k\rvert^{-1/2n} \\ X_{k+1} = \frac{1}{2}(\gamma_k X_k + (\gamma_k Y_k)^{-1}) \\ Y_{k+1} = \frac{1}{2}(\gamma_k Y_k + (\gamma_k X_k)^{-1}) \end{cases}$
Visser	$X_{k+1} = X_k + \frac{1}{2}(A - X_k^2)$
	Methods for 1/3-th Root
Iannazzo	$\begin{cases} X_{k+1} = X_k \left(\frac{2I + Y_k}{3}\right) \\ Y_{k+1} = \left(\frac{2I + Y_k}{3}\right)^{-3} Y_k \end{cases}$
Newton	$X_{k+1} = \frac{1}{3}(2X_k + X_k A^{-2})$
Visser	$X_{k+1} = X_k + \frac{1}{3}(A - X_k^3)$

C EXPERIMENTAL DETAILS

C.1 DETAILED EXPLANATION OF ALGORITHM 1

Algorithm 1 is summarized in Section 3.3. Here we explain how each subroutine $Expandable$, $Best_{UCB}$, $ExpandNode$, $SampleRolloutList$, and $backpropagate$ is implemented.

To begin with, we have two important flags at each state. One flag is IsTransitionable: if the iteration type and all parameters for that iteration is fixed, we set IsTransitionable(s) = True. Else, IsTransitionable(s) = False. Another flag is IsCoupled: for the root state, IsCoupled = True. If you use a coupled iteration at a state where IsCoupled = True, IsCoupled = True at the next state also. If you use an iteration that is not coupled, we set IsCoupled = False. When IsCoupled = True, you can do either coupled or uncoupled iteration. When IsCoupled = False, you can either do the "coupling" iteration (that would be specified later for each matrix function) or an iteration that is not coupled. Whenn you do the coupling iteration, IsCoupled = True for the next state, else IsCoupled = False. A table that summmarizes the transition of IsCoupled is as below.

Table 5: State transition of IsCoupled

Current IsCoupled	Iteration Type	Next IsCoupled
True	Coupled iteration	True
True	Uncoupled iteration	False
False	Coupling iteration	True
False	Non-coupling iteration	False

$Expandable(s)$ is a method that determines whether it is possible to expand a child node from current node s. If $IsTransitionable(s) = True$, the possible choice of next action becomes a discrete set of iterations. Hence $Expandable(s) = True$ if $IsCoupled(s) = False$ and number of children of $s <$ number of iterations that are not coupled, or $IsCoupled(s) = True$ andd number of children of $s <$ number of iterations - 1. We subtract 1 because we will not expand with coupling iteration. If $IsTransitionable = False$, we expand with a continuous variable hence we do progressive widening. If number of children of $s < C_{pw}N(s)^{\alpha_{pw}}$ where $n(s)$ is the number of visits for node s, we return True and else return False.

Algorithm 5 $Expandable(s)$

if $IsTransitionable(s)$ **then**
  if $IsCoupled(s) =$ False **then**
    if number of children of $s <$ number of *non-coupled* iterations **then**
      return True
    else
      return False
    end if
  else
    if num_children$(s) <$ num_iterations $- 1$ **then**
      return True
    else
      return False
    end if
  end if
else
  if number of children of $s < C_{pw}n(s)^{\alpha_{pw}}$ **then**
    return True
  else
    return False
  end if
end if

$Best_{UCB}(s)$ is simple: Choose the child node c with the maximal value of $V(c)+C_{ucb}\sqrt{\frac{\log n(s)+1}{n(c)}}$.

Algorithm 6 $Best_{UCB}(s)$

Input: Node s with children set $\mathcal{C}(s)$
Parameters: Exploration constant C_{ucb}
$best\_value \leftarrow -\infty$
$best\_child \leftarrow$ null
for all $c \in \mathcal{C}(s)$ **do**
  $score \leftarrow V(c) + C_{ucb}\sqrt{\frac{\log(n(s))+1}{n(c)}}$
  if $score > best\_value$ **then**
    $best\_value \leftarrow score$
    $best\_child \leftarrow c$
  end if
end for
return $best\_child$

$ExpandNode(s)$ depends on $IsTransitionable$. If $IsTransitionable(s) = True$, simply adding a node that hasn't been visited is enough, because the children are discrete. If $IsTransitionable(s) = False$, the children can take continuous parameters. If $num\_child(s) \leq E$ for hyperparameter E, do random sampling in range $[lo, hi]$ that is prespecified. Else, find the child with the best value and sample near that parameter p. Specifically, with probability 0.05, sample uniformly at random from $[lo, hi]$. Else sample random uniform at a new interval $[lo, hi] \cap [p - stddev\_scale * (hi - lo)/2.0, p + stddev\_scale * (hi - lo)/2.0]$. $stddev\_scale = 1/\log(2 + n(s))$ decays logarithmically with $n(s)$, the visit count.

Algorithm 7 $ExpandNode(s)$

if $IsTransitionable(s)$ **then**
  Add a new discrete child node to s
else
  if $num\_child(s) \leq E$ **then**
    Sample $x \sim \mathcal{U}[lo, hi]$
    Add child node with parameter x
  else
    $p \leftarrow$ parameter of best-value child of s
    $w \leftarrow (hi - lo)/2$
    $stddev\_scale \leftarrow 1/\log(2 + n(s))$
    $r \sim \mathcal{U}[0, 1]$
    if $r < 0.05$ **then**
      Sample $x \sim \mathcal{U}[lo, hi]$
    else
      Define interval $I = [lo, hi] \cap [p - stddev\_scale \cdot w,\ p + stddev\_scale \cdot w]$
      Sample $x \sim \mathcal{U}[I]$
    end if
    Add child node with parameter x
  end if
end if

$SampleRolloutList()$ samples a baseline rollout algorithm that is consisted of mutiple iterations of well-working baselines such as scalednewton for sign or scaled Denman-Beavers for matrix square root. If the rollout is coupled iteration but the current state is not coupled, we append the coupling iteration at the front of rollout.

At last, $Backpropogate(s)$ uses Bellman equation to update $V(s)$ in the path from root to s and if $V(s_0)$ is updated, we update bestpath and bestrollout accordingly.

C.2 Experimental Environment

All GPU based experiments were done in NVIDIA RTX A-6000 and CPU based experiments were done in AMD EPYC 7713 64-Core Processor. We repeated the experiments five times and picked the best algorithm, and if the method diverged for five times we ran additional experiments to find a good algorithm.

C.3 Hyperparameters

Here we detail the hyperparameters in Algorithm 1: this includes basic parameters such as α in progressive widening, list of possible actions for each matrix function, and $RolloutList$ for each matrix function.

We set $C_{pw} = 2, \alpha_{pw} = 0.3, C_{ucb} = 5, E = 5, \epsilon_{tol} = 1e-6$ and $1e-11$ for the experiments. The loss function and RolloutList for each matrix function is as below:

Table 6: Loss function, action list, and rollout list for each matrix function

Function	Loss Function	ActionList	RolloutList
Inv	$\frac{\|AX-I\|_F}{\|A\|_F}$	`[Inv_NS, Inv_Chebyshev]`	`[Inv_NS, Inv_Chebyshev]`
Sign	$\frac{\|X^2-I\|_F}{\|A\|_F}$	`[Sign_NS, Sign_Newton, Sign_Quintic, Sign_Halley]`	`[Sign_ScaledNS, Sign_ScaledNewton, Sign_Halley]`
Sqrt	$\frac{\|X^2-A\|_F}{\|A\|_F}$	`[Sqrt_DB, Sqrt_NSV, Sqrt_Visser, Sqrt_VisserCoupled, Sqrt_Coupling]`	`[Sqrt_ScaledDB, Sqrt_NSV]`
Proot	$\frac{\|X^3-A\|_F}{\|A\|_F}$	`[Proot_Newton, Proot_Visser, Proot_Iannazzo, Proot_Coupling]`	`[Proot_Newton, Proot_Visser, Proot_Iannazzo]`

Each iteration in ActionList is parametrized to have tunable parameters. A full table denoting how each action is parameterized is as Table 7.

C.4 List of Distributions

The list of distributions we used throughout the experiments are as follows:
1. **Wishart** denotes $A = \frac{X^\top X}{3d} + \epsilon_{\text{stb}} I$ where $X \in \mathbb{R}^{d/4 \times d}$, $X_{ij} \sim \mathcal{N}(0,1)$ i.i.d.. $\epsilon_{\text{stb}} = 1e-3$ exists for numerical stability.
2. **Uniform** denotes $A = QDQ^T$ where Q is sampled from a Haar distribution and D is a diagonal matrix where its entries are sampled from uniform $[-1, 1]$. We cap the diagonal entries with absolute value $<$ 1e-3 to 1e-3.
3. **Hessian of Quartic** is the indefinite Hessian of a d-dimensional quartic $\sum_i z_i^4/4 - z_i^2/4$ evaluated at a random point $z \sim \mathcal{N}(0, \mathbf{I_d})$. We cap the eigenvalues with absolute value ¡ 1e-3 to 1e-3, and normalize with Frobenius norm.
4. **CIFAR-10** is the random input matrix is $\hat{\Sigma} = \frac{1}{n}(X-\mu)^T(X-\mu)$ where $X \in \mathbb{R}^{n \times d}$ is a random batch of n flattened CIFAR-10 images and μ is the average of the n images. We normalize with the Frobenius norm and add $\epsilon_{stb} I$ for $\epsilon_{stb} = 1e-3$.
5. **Erdos-Renyi** is the normalized graph Laplacian of a random Erdos-Renyi graph. We set $p = 0.4$ and $d = 5000$ for the experiments.

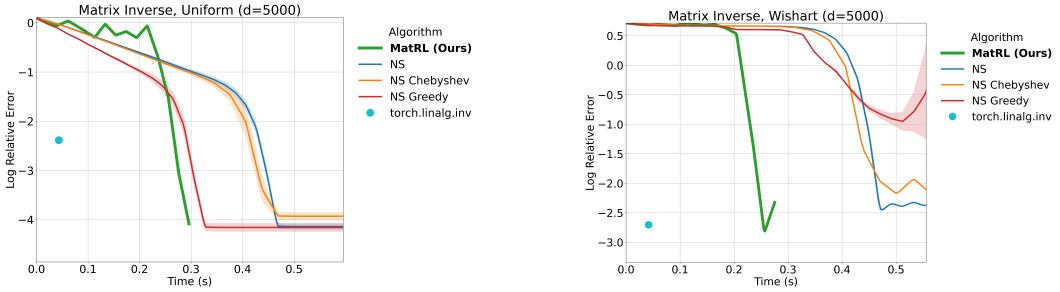
Table 7: How actions are parametrized

Method	Iteration Formula	Parameter Range
	Actions for Inverse	
Newton (Schulz)	$X_{k+1} = a_k X_k - b_k X_k A X_k$	$a_k, b_k \in [0, 5]$
Chebyshev	$X_{k+1} = a_k X_k - b_k X_k A X_k + c_k X_k A X_k A X_k$	$a_k, b_k, c_k \in [0, 5]$
	Actions for Sign	
Newton	$X_{k+1} = \frac{1}{2}(a_k X_k + (a_k X_k)^{-1})$	$a_k \in [0, 40]$
NewtonSchulz	$X_{k+1} = X_k + a_k(b_k X_k - (b_k X_k)^3)$	$a_k, b_k \in [0, 5]$
Quintic	$X_{k+1} = a_k X_k + b_k X_k^3 + c_k X_k^5$	$a_k, b_k, c_k \in [0, 5]$
Halley	$X_{k+1} = X_k(a_k I + b_k X_k^2)(I + c_k X_k^2)^{-1}$	$a_k, b_k, c_k \in [0, 40]$
	Actions for SquareRoot	
DenmanBeavers	$\begin{cases} X_{k+1} = \frac{1}{2}(a_k X_k + (b_k Y_k)^{-1}) \\ Y_{k+1} = \frac{1}{2}(b_k Y_k + (a_k X_k)^{-1}) \end{cases}$	$a_k, b_k \in [0, 50]$
NewtonSchulzVariant	$\begin{cases} X_{k+1} = \frac{1}{2}(a_k X_k - b_k X_k Y_k X_k) \\ Y_{k+1} = \frac{1}{2}(a_k Y_k - b_k Y_k X_k Y_k) \end{cases}$	$a_k, b_k \in [0, 5]$
Visser	$X_{k+1} = a_k X_k + b_k(A - X_k^2)$	$a_k, b_k \in [0, 10]$
Visser_Coupled	$\begin{cases} X_{k+1} = a_k X_k + b_k(A - X_k^2) \\ Y_{k+1} = a_k Y_k + b_k(I - X_k Y_k) \end{cases}$	$a_k, b_k \in [0, 10]$
Coupling	$Y_k = X_k A^{-1}$	–
	Actions for 1/3-th Root	
Iannazzo	$\begin{cases} X_{k+1} = X_k \left(\frac{a_k I + b_k Y_k}{3}\right) \\ Y_{k+1} = \left(\frac{a_k I + b_k Y_k}{3}\right)^{-3} Y_k \end{cases}$	$a_k, b_k \in [0, 10]$
Newton	$X_{k+1} = \frac{1}{3}(a_k X_k + b_k X_k A^{-2})$	$a_k, b_k \in [0, 10]$
Visser	$X_{k+1} = a_k X_k + b_k(A - X_k^3)$	$a_k, b_k \in [0, 10]$
Coupling	$Y_k = A X_k^{-3}$	–

D LIST OF ALL EXPERIMENTAL RESULTS

D.1 DIFFERENT MATRIX FUNCTIONS

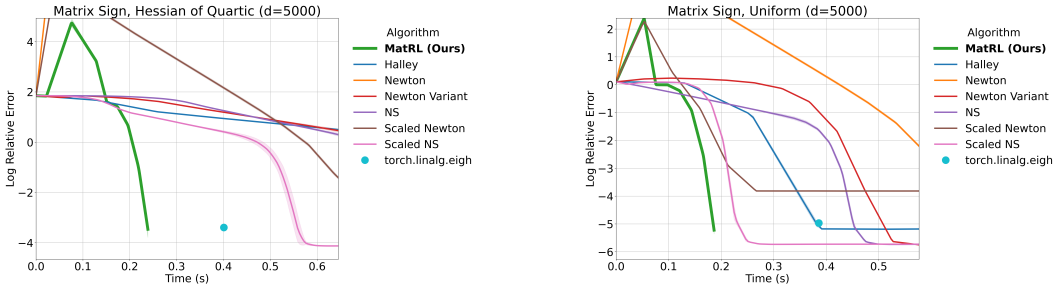
Inverse We learn to compute matrix inverse for two different distributions, Wishart and Uniform. Unfortunately, in our experiments, using Newtonschulz to compute matrix inverse was much slower than directly using torch.linalg.inv. However for uniform distribution, we had a more precise approximation of the inverse in terms of the loss than torch.linalg.inv.



(a) Uniform distribution (b) Wishart distribution

Figure 3: Computing matrix inverse with NewtonSchulz and variants

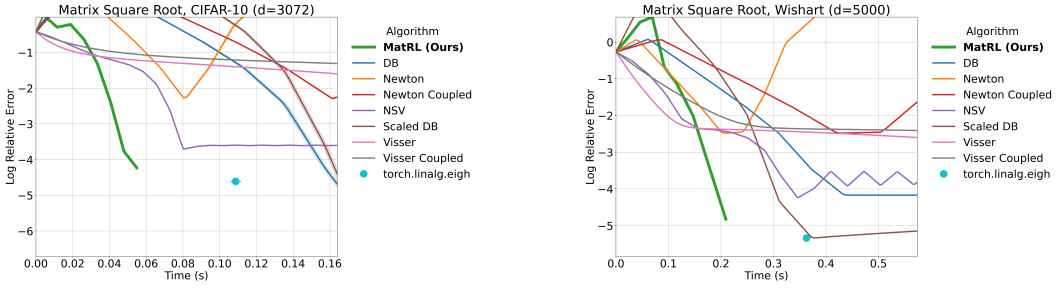
Matrix sign We learn matrix sign for Quartic Hessian and for matrices with Uniform [-1, 1] diagonal entries. Here $d = 5000$.



(a) Hessian of Quartic (b) Uniform distribution

Figure 4: Computing matrix sign with NewtonSchulz and variants

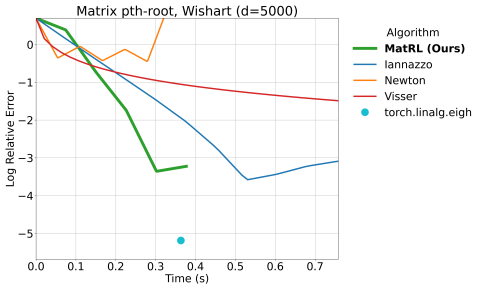
Matrix sqrt We learn matrix sqrt for CIFAR-10 and Wishart matrices with $d = 5000$. For CIFAR-10, the matrix is the empirical covariance matrix added by epsilon.



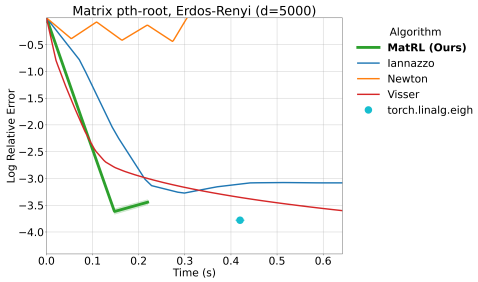
(a) CIFAR-10 (b) Wishart distribution

Figure 5: Computing matrix square root with NewtonSchulz and variants

Matrix 1/3-root We learn matrix 1/3-root for Wishart matrices and Erdos-Renyi graph. For Wishart matrices the method is not very effective: torch.linalg.eigh can find matrix 1/3-root with better accuracy with the same amount of time. However, for normalized graph Laplacians of Erdos-Renyi graph, it finds a faster algorithm with almost similar accuracy.



(a) Wishart distribution



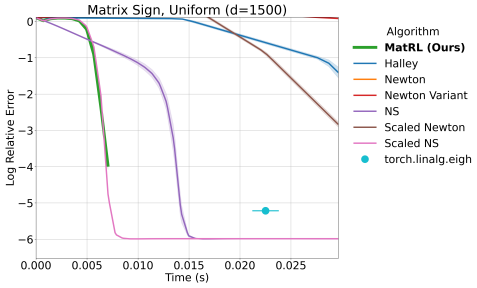
(b) Normalized graph Laplacian of Erdős-Rényi

Figure 6: Examples of matrix distributions: structural or spectral views.

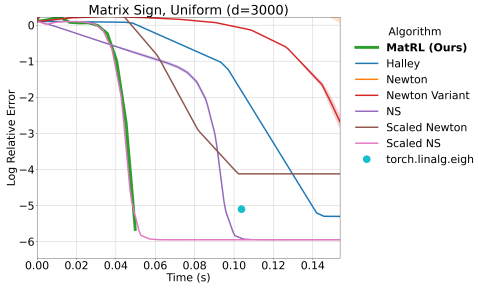
D.2 Algorithm 1 Adapting: sizes, precision, compute

We demonstrate that different d, precision (float or double), and compute (GPU/CPU) can lead to different algorithms with matrix sign computation. We show both the loss curve and the found algorithm in each case.

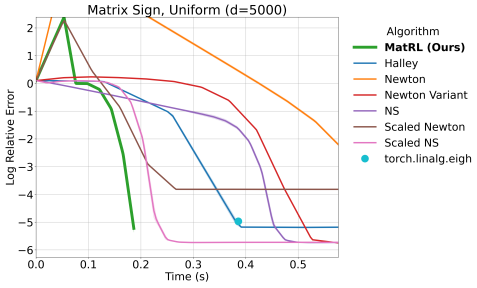
Different problem sizes Here we show the results to compute matrix sign on random matrices with spectrum $Unif[-1, 1]$. $d = 1500, 3000, 5000, 10000$. One trend that we see is that for small d, we tend to use NewtonSchulz more, whereas for larger d we tend to use Newton step more. It is related with the relative cost between Newton step and Newtonschulz step.



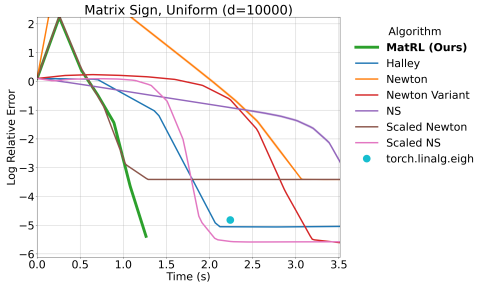
(a) $d = 1500$



(b) $d = 3000$



(c) $d = 5000$



(d) $d = 10000$

Figure 7: Computing matrix sign for different matrix sizes

The found algorithms for $d = 1500, 3000, 5000, 10000$ are as follows.

Algorithm 8 Iterative SIGN for Uniform on GPU with $d = 1500$

Input: A
Initialize $X_0 = A$
Set $a \leftarrow [1.731, 1.729, 1.724, 1.712, 1.682, 1.606, 1.439, 1.190, 1.029, 1.000]$,
// rounded off to three digits
for $i = 1$ to 10 **do**
  $X_i = a_{i-1}X_{i-1} + 0.5(a_{i-1}X_{i-1} - (a_{i-1}X_{i-1})^3)$
end for
return X_9

Algorithm 9 Iterative SIGN for Uniform on GPU with $d = 3000$

Input: A
Initialize $X_0 = A$
Set $a \leftarrow [2.179, 1.000], b \leftarrow [1.135, 1.5, 0.5]$
$c \leftarrow [4.308, 1.711, 1.679, 1.599, 1.424, 1.176, 1.025, 1.000]$,
// rounded off to three digits
$X_1 = a_1X_0 + a_0(a_1X_0 - (a_1X_0)^3)$
$X_2 = b_0X_1 - b_1X_1^3 + b_2X_1^5$
for $i = 3$ to 10 **do**
  $X_i = 1.5(c_{i-3}X_{i-1}) - 0.5(c_{i-3}X_{i-1})^3$
end for

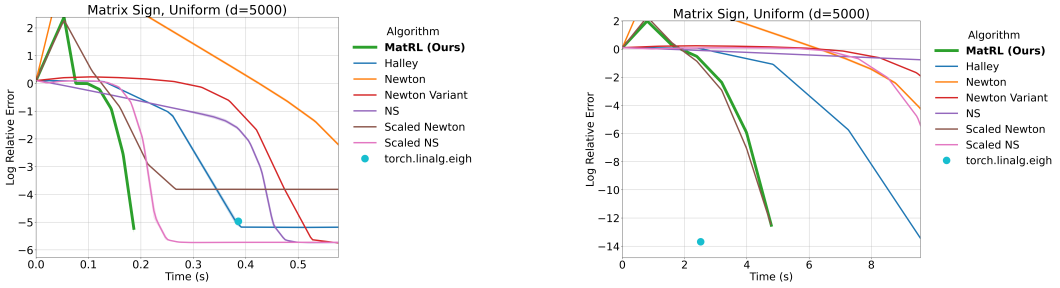
Algorithm 10 Iterative SIGN for Uniform on GPU with $d = 5000$

Input: A
Initialize $X_0 = A$
$a \leftarrow [35.536]$
$b \leftarrow [0.094, 1.607, 1.439, 1.191, 1.029, 1]$
for $i = 1$ to 1 **do**
  $X_i = 0.5(a_{i-1}X_{i-1} + (a_{i-1}X_{i-1})^{-1})$
end for
for $i = 2$ to 7 **do**
  $X_i = 1.5(b_{i-2}X_{i-1}) - 0.5(b_{i-2}X_{i-1})^3$
end for
return X_7

Algorithm 11 Iterative SIGN for Uniform on GPU with $d = 10000$

Input: A
Initialize $X_0 = A$
Set $a \leftarrow [29.691, 0.243], b \leftarrow [0.616, 1.104, 1.008, 1]$,
// rounded off to three digits
for $i = 1$ to 2 **do**
  $X_i = 0.5(a_{i-1}X_{i-1} + (a_{i-1}X_{i-1})^{-1})$
end for
for $i = 3$ to 6 **do**
  $X_i = b_{i-1}X_{i-1} + 0.5(b_{i-1}X_{i-1} - (b_{i-1}X_{i-1})^3)$
end for
return X_6

Different precision Here we show the results to compute matrix sign on random matrices with spectrum $Unif[-1, 1]$ for float and double precision. For double precision when $d = 5000$, NewtonSchulz becomes as expensive as Newton step whereas Newton step is more effective - hence we use Newton until the end. For float precision the method finds a mixture of Newton and NewtonSchulz. For double we used $\epsilon_{tol} = 1e - 11$.



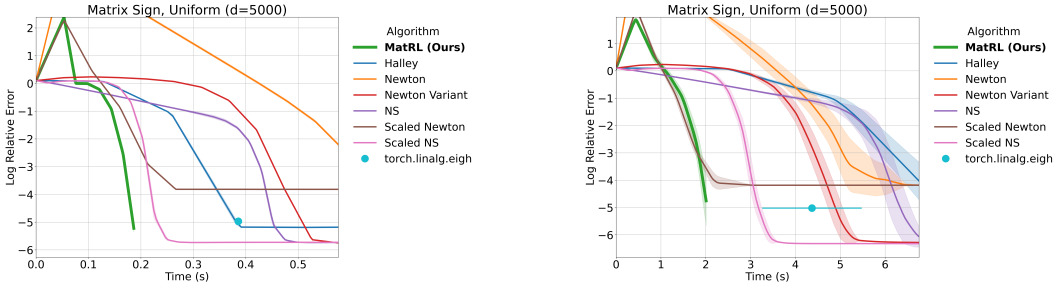
(a) Float (b) Double

Figure 8: Computing matrix sign for different precision

Algorithm 12 Iterative SIGN for Uniform on GPU with $d = 5000$, DOUBLE

Input: A
Initialize $X_0 = A$
$a \leftarrow [22.055, 0.244, 0.590, 0.938, 0.998, 1]$
for $i = 1$ to 6 **do**
  $X_i = 0.5(a_{i-1}X_{i-1} + (a_{i-1}X_{i-1})^{-1})$
end for
return X_6

Different compute We also run MatRL on GPU and on CPU. The difference that occurs here is also similar in vein: on a GPU, Newton step is $\approx$ x2.28 more costly than a Newtonschulz step, whereas on a CPU it is $\approx$ x1.62 more costly. This makes the algorithm found on CPU use Newton step more.



(a) GPU (NVIDIA RTX A-6000) (b) CPU (AMD EPYC 7713 64-Core Processor)

Figure 9: Computing matrix sign on different machines

Algorithm 13 Iterative SIGN for Uniform on CPU with $d = 5000$

Input: A
Initialize $X_0 = A$
Set $a \leftarrow [19.751, 0.205], b \leftarrow [0.524, 1.161, 1.020, 1.000]$,
// rounded off to three digits
for $i = 1$ to 2 **do**
  $X_i = 0.5(a_{i-1}X_{i-1} + (a_{i-1}X_{i-1})^{-1})$
end for
for $i = 3$ to 6 **do**
  $X_i = b_{i-3}X_{i-1} + 0.5(b_{i-3}X_{i-1} - (b_{i-3}X_{i-1})^3)$
end for
return X_6