The Brute-Force Aesthetics of Matrix Function Approximation

Jianlin Su

2026-06-25

Previously, in Efficient Computation of Matrix Square Roots and Inverse Square Roots and Efficient Computation of Matrix r-th Roots and Inverse r-th Roots, we discussed efficient computation of matrix powers, which extended the Newton–Schulz iteration originally used to compute msign to matrix powers. Recently, the arXiv preprint Muon^p: Muon with Fractional Spectral Powers provided another idea for constructing iterations to compute matrix powers, which also gave the author some inspiration.

However, whether it is the author's approach or the Muon^p approach, overall they are "usable but not general enough"—for example, they can only be used to compute rational powers of matrices, and the complexity grows with the numerator and denominator of the reduced fraction, which is clearly unscientific. To overcome these defects, this article proposes a general approximation framework that can theoretically fit arbitrary matrix functions.

Two Types of Functions

When speaking of matrix functions, there are actually two slightly different meanings; here we briefly introduce them.

The first is the matrix function operating on eigenvalues (EIG-type). Let the eigenvalue decomposition of matrix $\boldsymbol{M}$ be $\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$; then define $f[\boldsymbol{M}] \triangleq \boldsymbol{Q}f(\boldsymbol{\Lambda})\boldsymbol{Q}^{-1}$, where $f(\boldsymbol{\Lambda})$ means applying the function f element-wise to the diagonal. This is in fact what we usually call a matrix function; for example, matrix exponential and matrix logarithm both belong to this class, and they are usually defined via power series.

The second is the matrix function operating on singular values (SVD-type). Let the singular value decomposition of matrix $\boldsymbol{M}$ be $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$; then define $f\{\boldsymbol{M}\} \triangleq \boldsymbol{U}f(\boldsymbol{\Sigma})\boldsymbol{V}^\top$. The msign operation of Muon, and the mclip operation discussed in Computing Singular Value Clipping mclip via msign (Part I) and Computing Singular Value Clipping mclip via msign (Part II), all belong to this class.

EIG-type matrix functions only hold for square matrices, and eigenvalue decomposition is guaranteed to exist only over the complex numbers, so the definition of f usually needs

to be extended to complex numbers (unless the input matrix is restricted to real symmetric matrices); SVD-type matrix functions can be defined for matrices of arbitrary shape, any real matrix has a real singular value decomposition, and the singular values are always non-negative. These properties make the definition and analysis of SVD-type matrix functions relatively simpler.

These two classes of matrix functions each have their own application scenarios; their computational difficulty is not fundamentally different, but the details differ. For powers, the arbitrary positive integer power of an eigenvalue can be computed directly, but for singular values we can only efficiently compute odd powers:

$$\begin{aligned}[M]^n &= \boldsymbol{Q}\boldsymbol{\Lambda}^n\boldsymbol{Q}^{-1} = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1})^n = \boldsymbol{M}^n \\ \{\boldsymbol{M}\}^{2n+1} &= \boldsymbol{U}\boldsymbol{\Sigma}^{2n+1}\boldsymbol{V}^\top = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top(\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top)^n = \boldsymbol{M}(\boldsymbol{M}^\top\boldsymbol{M})^n\end{aligned} \tag{1}$$

That is, $[\boldsymbol{M}]^n$ is consistent with $\boldsymbol{M}^n$ as defined by matrix multiplication, while $\{\boldsymbol{M}\}^{2n+1}$ must be computed indirectly via $(\boldsymbol{M}^\top\boldsymbol{M})^n$. Therefore, when we want to approximate a matrix function via polynomials, EIG-type functions may choose a polynomial of any degree, whereas SVD-type functions may only choose odd-degree polynomials.

Existing Methods

This section uses the SVD-type fractional power $\{\boldsymbol{M}\}^{1/3}$ as an example to introduce two existing approximate computation schemes. For simplicity, assume the singular values of $\boldsymbol{M}$ lie in $[0, 1]$.

The first idea uses the identity $\{\boldsymbol{M}\}^{1/3} = \boldsymbol{M}(\boldsymbol{M}^\top\boldsymbol{M})^{-1/3}$, then substitutes the results of Efficient Computation of Matrix r-th Roots and Inverse r-th Roots to obtain

$$\boldsymbol{G}_0 = \boldsymbol{M}, \quad \boldsymbol{P}_0 = \boldsymbol{M}^\top\boldsymbol{M}$$

$$\boldsymbol{G}_{t+1} = \boldsymbol{G}_t(a_{t+1}\boldsymbol{I} + b_{t+1}\boldsymbol{P}_t + c_{t+1}\boldsymbol{P}_t^2) \tag{2}$$

$$\boldsymbol{P}_{t+1} = (a_{t+1}\boldsymbol{I} + b_{t+1}\boldsymbol{P}_t + c_{t+1}\boldsymbol{P}_t^2)^3\boldsymbol{P}_t \tag{3}$$

Finally we have $\lim_{t\to\infty}\boldsymbol{G}_t = \{\boldsymbol{M}\}^{1/3}$. Note that the iteration for $\boldsymbol{P}_t$ is independent; the principle of this scheme is to choose suitable a_t, b_t, c_t so that $\boldsymbol{P}_t \to \boldsymbol{I}$, and then the product of the separated terms $a_{t+1}\boldsymbol{I} + b_{t+1}\boldsymbol{P}_t + c_{t+1}\boldsymbol{P}_t^2$ tends to $\boldsymbol{P}_0^{-1/3}$. Its main issue is that one must first explicitly compute $\boldsymbol{M}^\top\boldsymbol{M}$, causing the condition number to be squared and the numerical accuracy to drop accordingly.

The second idea comes from Muon^p: Muon with Fractional Spectral Powers. It views computing $m^{1/3}$ as finding the root of the equation $x^3 - m = 0$; then as long as we design an odd-degree polynomial iteration to find the root of this equation, we can write the corresponding matrix iteration. The paper considers the fixed-point iteration

$$x_{t+1} = x_t + c(m - x_t^3) \qquad \Leftrightarrow \qquad \boldsymbol{X}_{t+1} = \boldsymbol{X}_t + c(\boldsymbol{M} - \boldsymbol{X}_t\boldsymbol{X}_t^\top\boldsymbol{X}_t) \tag{4}$$

By simple analysis one can prove that the condition guaranteeing convergence for any $m \in [0,1]$ is $c \leq 2/3$; if a fixed value is used one may take $c = 2/3$. What inspired the author most about this idea is that the input $\boldsymbol{M}$ is incorporated into every step of the iteration, rather than relying solely on the previous result $\boldsymbol{X}_t$, but it still leaves many problems unsolved. For example, when the iteration format has multiple parameters, how to determine them one by one, and whether one can choose different parameters at each step to improve the convergence speed—the answers to these questions remain unknown.

In addition, the two ideas above share a common problem: their computational cost strongly depends on the reduced-fraction form of the desired power. For example, if we want to compute the 0.33 power, the reduced fraction is 33/100, so both ideas would require us to compute the 100th power of some matrix, which is a considerable cost, but in fact 0.33 is scarcely different from 1/3, so such a huge difference is unreasonable.

Brute-Force Aesthetics

Therefore, we need a more general framework that can derive a usable approximation for a specified matrix function, and can automatically produce similar results for similar matrix functions; ideally it would also support arbitrary functions beyond power functions. This is the goal of the present article.

Without loss of generality, consider the SVD-type matrix function $f\{\boldsymbol{M}\}$. It maps a singular value m to $f(m)$; we need to construct an odd-degree polynomial iteration to approximate $f(m)$. Assume each step depends on the current value x_t and the input m (in general, we could also include $x_1, \ldots, x_{t-1}$ in the iteration), and that the degree of each step does not exceed 3; then we can construct the general iteration format

$$x_{t+1} = c_{t+1,1}x_t + c_{t+1,2}m + c_{t+1,3}x_t^3 + c_{t+1,4}m^3 + c_{t+1,5}x_t^2 m + c_{t+1,6}x_t m^2 \tag{5}$$

where $\boldsymbol{c}_{t+1} = (c_{t+1,1}, \ldots, c_{t+1,6})$ are the parameters to be determined, six per step, and different values are allowed at each step in order to increase the approximation accuracy. The idea here is actually very direct: just add in every term we can think of, without forcing interpretability, and let the subsequent fitting result decide whether they are useful. Although this lacks the elegant feeling of theoretically obtaining an analytic solution, it possesses a certain brute-force aesthetic of universal computation.

How should we then determine $\boldsymbol{c}_{t+1}$? A naive approach, as in the previous articles Appreciation of the Muon Optimizer: The Essential Leap from Vectors to Matrices and Newton–Schulz Iteration for the msign Operator (Part I), is to first fix a total number of iterations T. Then the whole iteration can be viewed as a model; next choose a regression target, and use a gradient optimizer to train end-to-end.

However, although this approach theoretically has a chance to obtain a solution closer to the global optimum, in practice it often faces many difficulties: the nonlinearity and non-convexity after multiple iterations, as well as the randomness of initialization, all noticeably affect the results.

Greedy Strategy

In order to stabilize the results, the author proposes a layer-by-layer greedy solving strategy—at each step assuming $\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_t$ are already known, and then solving for $\boldsymbol{c}_{t+1}$ with optimization objective

$$\boldsymbol{c}_{t+1}^* = \arg\min_{\boldsymbol{c}_{t+1}} d(x_{t+1}, f) \qquad \text{s.t.} \qquad \|\boldsymbol{c}_{t+1}\|_\infty \leq B \tag{6}$$

Here $\|\boldsymbol{c}_{t+1}\|_\infty \leq B$ constrains the maximum absolute value of any component of $\boldsymbol{c}_{t+1}$ not to exceed B, avoiding precision loss caused by extreme amplification followed by shrinkage; we could also consider choosing different B_i for each $c_{t+1,i}$. $d(x_{t+1}, f)$ is the cost function; we have two choices:

$$L_2: \qquad \int_a^b (x_{t+1} - f(m))^2 \, dm \tag{7}$$

$$L_\infty: \qquad \max_{m \in [a,b]} |x_{t+1} - f(m)| \tag{8}$$

where $[a, b]$ is the interval of singular values we care about. Clearly, L_2 cares about the average error, while L_∞ cares about the maximum error; below we will see that under either choice we can solve exactly.

Although the greedy strategy cannot guarantee optimality, it has its own unique advantages. First, Polar Express proved that if $f(m) = 1$ (i.e. computing msign), then the greedy solution is the optimal solution, indicating that in some cases the greedy solution can also be very close to optimal; second, each step of the greedy solution is exact and progressive, ensuring that every step moves closer to the target. We can precompute a sufficient number of $\boldsymbol{c}^*$, and then truncate the first several steps as needed, obtaining a good approximation.

Finally, because each step of the greedy solution approximates the target, its fluctuations are still controllable, and the phenomenon of violent amplification followed by shrinkage will not appear. This is particularly important for numerical computation, especially low-precision computation. If necessary, we can add more constraints to achieve desired properties; for example, $\|\boldsymbol{c}_{t+1}\|_\infty \leq B$ is a very basic constraint.

Solving One by One

The main reason for choosing these two cost functions is that their corresponding optimization problems can be solved exactly; next we demonstrate their solution processes.

First is L_2; it requires computing an integral. After discretizing the integral, it becomes a regression problem with a finite sample. Noting that x_{t+1} is linear in $\boldsymbol{c}_{t+1}$, this is simply a linear regression problem, exactly solvable! Even if we add some linear constraints, it

is merely a convex quadratic program, still exactly solvable and with very mature solver tools.

Next is L_∞; it requires taking the max of the error over the whole interval. We also discretize this for approximation. Then, through an elegant transformation, we can convert it into a linear programming problem: introduce a new variable z, giving

$$\min_{\boldsymbol{c}_{t+1}} \max_{m\in[a,b]} |x_{t+1} - f(m)| = \min_{\boldsymbol{c}_{t+1},z} \left\{ z \mid -z \le x_{t+1} - f(m) \le z, \forall m \in [a,b] \right\} \tag{9}$$

If $[a,b]$ is discretized into N points, then these samples transform into $2N$ linear inequality constraints on 7 unknowns, among which z is minimized. Linear programming is simpler and more mature than quadratic programming, so naturally there is no difficulty.

The successful conversion of the L_∞ case into a linear program is also one of the advantages of the greedy strategy. If we tried to use a gradient-descent-like optimizer to jointly optimize the various $\boldsymbol{c}$, the max of L_∞ would be a core obstacle to optimization, because gradients cannot propagate effectively through max, and combined with the extreme nonlinearity after multiple iterations, efforts to find a global solution would be difficult to succeed. But through the "greedy solution + linear programming" conversion, we can stably obtain an effective solution.

Reference Implementation

If restricted to NumPy and SciPy, for bounded linear regression we can use `scipy.optimize.lsq_linear`, and for general linear programming we can use `scipy.optimize.linprog`. But considering code generality and simplicity, as well as the possibility of adding custom constraints at any time, we recommend solving based on dedicated convex optimization tools such as CVXPY.

Below, taking $f(m) = m^{1/3}$ as an example, we give a reference implementation based on CVXPY:

```
import numpy as np
import cvxpy as cp

N = 10000  # Number of discretization points
B = 10  # Parameter bound
m = np.linspace(0, 1.01, N)  # We care about singular values in 0~1,
                             # but use a slightly larger range for
                             # parameter estimation to ensure stability

x, f = m, m**(1 / 3)
coefs = []
for t in range(10):
    A = np.array([x, m, x**3, m**3, x**2 * m, x * m**2]).T
    c = cp.Variable(6)
```

```
    objective = cp.Minimize(cp.sum_squares(A @ c - f))  # L2
    # objective = cp.Minimize(cp.max(cp.abs(A @ c - f)))  # L-infinity
    constraints = [cp.abs(c) <= B]
    problem = cp.Problem(objective, constraints)
    result = problem.solve()
    coefs.append(c.value.round(3))
    x = A @ c.value.round(3)
    print(f'iter {t + 1}, max error:', np.abs(x - f).max())
    print(f'iter {t + 1}, mse error:', np.square(x - f).mean())

coefs = np.array(coefs)
```

Effect Comparison

Below we compare the L_2 greedy solution, the L_∞ greedy solution, and the fixed-step-size iteration of Muon^p, $x_{t+1} = x_t + \frac{2}{3}(m - x_t^3)$, on the cube root $f(m) = m^{1/3}$. All methods start from $x_0 = m$, the discretization interval is $[0, 1.01]$, the parameter bound is $B = 10$, and T is the total number of iterations.

> L_2 **greedy solution, 10 iterations**: maximum error approximately 8.5×10^{-2}, mean squared error approximately 5.4×10^{-5};
>
> L_∞ **greedy solution, 10 iterations**: maximum error approximately 4.6×10^{-2}, mean squared error approximately 8.2×10^{-4};
>
> **Muon^p, 10 iterations**: maximum error approximately 1.4×10^{-1}, mean squared error approximately 6.3×10^{-4};
>
> **Muon^p, 20 iterations**: maximum error approximately 1.0×10^{-1}, mean squared error approximately 1.3×10^{-4}.

The comparison plot is as follows:

SVG image not supported by pdflatex.
See clickable source: assets/11787/3330219580.svg
Comparison of Cube Root Approximation Methods

From the numerical results, the L_∞ greedy solution has an obvious advantage in maximum error, L_2 is second, and the fixed-step-size Muon^p scheme at $T = 20$ has a maximum error still worse than the L_2 greedy solution at $T = 10$; from the figure one can see that the Muon^p scheme's weak region is mainly near 0, because $c = 2/3$, although taking into account the whole interval $[0, 1]$, is too small near 0. This is the main defect of a static step size.

By replacing 1/3 with 1/5 in the code, we can obtain an approximation to the 5th root without changing the iteration degree, but if using the Muon^p scheme or the author's previous r-th root algorithm, one would need to raise the iteration degree to at least 5. This demonstrates the generality of the framework proposed in this article.

Some Results

Below we give the L_∞ greedy iteration coefficients for computing $\{\boldsymbol{M}\}^0$ (msign), $\{\boldsymbol{M}\}^{1/2}$, $\{\boldsymbol{M}\}^{1/3}$, and $\{\boldsymbol{M}\}^{1/4}$. All coefficients are kept to three decimal places, with parameter bound $B = 10$. The fitting intervals for $m^{1/2}$, $m^{1/3}$, $m^{1/4}$ are $[0, 1.01]$; the fitting interval for m^0 (i.e. the constant 1) is $[0.001, 1.01]$.

The per-step iteration format is

$$\begin{aligned}\boldsymbol{X}_{t+1} =& c_{t+1,1}\boldsymbol{X}_t + c_{t+1,2}\boldsymbol{M} + c_{t+1,3}\boldsymbol{X}_t\boldsymbol{X}_t^\top\boldsymbol{X}_t \\ &+ c_{t+1,4}\boldsymbol{M}\boldsymbol{M}^\top\boldsymbol{M} + c_{t+1,5}\boldsymbol{X}_t\boldsymbol{X}_t^\top\boldsymbol{M} + c_{t+1,6}\boldsymbol{X}_t\boldsymbol{M}^\top\boldsymbol{M}\end{aligned} \tag{10}$$

where $\boldsymbol{X}_0 = \boldsymbol{M}$, and we assume the singular values of $\boldsymbol{M}$ have already been normalized into $[0, 1]$. Note that $\boldsymbol{M}^\top\boldsymbol{M}$ and $\boldsymbol{M}\boldsymbol{M}^\top\boldsymbol{M}$ must be computed in the first step and can be stored; $c_{t+1,3}\boldsymbol{X}_t\boldsymbol{X}_t^\top\boldsymbol{X}_t$ and $c_{t+1,5}\boldsymbol{X}_t\boldsymbol{X}_t^\top\boldsymbol{M}$ can be merged into $\boldsymbol{X}_t\boldsymbol{X}_t^\top(c_{t+1,3}\boldsymbol{X}_t + c_{t+1,5}\boldsymbol{M})$, so although this iteration has six terms, in practice it only adds one matrix multiplication $\boldsymbol{X}_t\boldsymbol{M}^\top\boldsymbol{M}$ compared with the Muon^p iteration.

The coefficients are as follows:

Summary

This article proposed a general matrix function approximation framework based on a greedy strategy—not pursuing strict interpretability, but directly constructing simple polynomial iterations. At each step it regresses toward the target following a greedy strategy, chooses an appropriate cost function, converts the problem into a quadratic program or a linear program, solves the iteration parameters exactly and stably, and finally obtains an effective computational scheme.

	t	$c_{t,1}$	$c_{t,2}$	$c_{t,3}$	$c_{t,4}$	$c_{t,5}$	$c_{t,6}$
	1	2.564	2.564	−1.256	−1.256	−1.256	−1.256
	2	2.668	1.243	−1.987	0.715	7.158	−10.000
	3	2.670	−0.793	−0.791	3.138	1.727	−4.170
	4	2.487	0.018	−0.635	0.033	0.019	−0.060
$\{\boldsymbol{M}\}^0$	5	2.350	0.017	−0.616	−0.013	−0.003	0.002
	6	2.095	0.001	−0.582	0.001	0.000	−0.002
	7	1.761	−0.000	−0.537	0.001	0.001	−0.002
	8	1.547	−0.001	−0.508	0.001	0.001	−0.001
	9	1.503	−0.000	−0.501	−0.000	0.000	0.000
	10	1.499	−0.000	−0.499	0.000	0.000	−0.000
	1	0.931	0.931	−0.245	−0.245	−0.245	−0.245
	2	1.366	0.368	−5.209	0.216	10.000	−5.723
	3	1.495	−0.070	−5.241	0.379	10.000	−5.557
	4	1.261	0.402	−5.486	−0.066	10.000	−5.143
$\{\boldsymbol{M}\}^{1/2}$	5	1.145	0.370	−4.675	1.648	10.000	−7.530
	6	1.112	0.429	−4.937	1.000	10.000	−6.635
	7	1.071	0.535	−5.013	1.129	10.000	−6.762
	8	1.060	0.427	−4.697	1.472	10.000	−7.288
	9	1.035	0.651	−5.296	0.695	10.000	−6.123
	10	1.027	0.506	−4.613	1.888	10.000	−7.839
	1	1.199	1.199	−0.405	−0.405	−0.405	−0.405
	2	1.168	1.717	−4.360	0.767	10.000	−8.393
	3	1.897	−0.485	−4.022	2.809	10.000	−9.167
	4	1.702	−0.140	−4.187	2.457	10.000	−8.854
$\{\boldsymbol{M}\}^{1/3}$	5	1.525	0.087	−4.142	2.415	10.000	−8.921
	6	1.392	0.406	−4.142	2.750	10.000	−9.500
	7	1.292	0.384	−3.870	3.122	10.000	−10.000
	8	1.245	0.809	−4.187	3.013	10.000	−10.000
	9	1.173	0.517	−3.433	3.409	9.255	−10.000
	10	1.180	1.091	−4.355	2.894	10.000	−9.933
	1	1.385	1.385	−0.517	−0.517	−0.517	−0.517
	2	1.398	1.910	−3.967	1.031	10.000	−9.554
	3	2.155	−0.986	−3.272	3.804	9.311	−10.000
	4	1.957	−0.663	−3.329	3.625	9.393	−10.000
$\{\boldsymbol{M}\}^{1/4}$	5	1.789	−0.099	−3.540	3.340	9.381	−10.000
	6	1.640	0.020	−3.559	3.267	9.537	−10.000
	7	1.517	0.324	−3.590	3.190	9.429	−10.000
	8	1.405	0.343	−3.395	3.280	9.260	−10.000
	9	1.345	0.651	−3.514	3.224	9.145	−10.000
	10	1.258	0.748	−3.030	3.640	8.201	−10.000