AdaPolySI: Adaptive Polynomial Filtered Subspace Iteration for Hermitian Interior Eigenvalue Problems

Yuhui Ni*
National University of Defense Technology
Changsha, China
n.yh@foxmail.com

Xiaofei Xu*
Xiangtan University
Xiangtan, China
LittleFFFEI@126.com

Shengguo Li†
National University of Defense Technology
Changsha, China
nudtlsg@nudt.edu.cn

Juan Zhang†
Xiangtan University
Xiangtan, China
zhangjuan@xtu.edu.cn

Juan Chen†
National University of Defense Technology, China
Changsha, China
juanchen@nudt.edu.cn

Jianchun Wang
China Ship Scientific Research Center
Wuxi, China
wangjianchun@cssrc.com.cn

Jose E. Roman
Universitat Politècnica de València
Valencia, Spain
jroman@dsic.upv.es

Abstract

Interior eigenvalue problems arise in many scientific and engineering applications. This paper presents `AdaPolySI`, an adaptive polynomial-filtered subspace iteration eigensolver for large Hermitian interior eigenvalue problems, based on a Chebyshev approximation of the step function. AdaPolySI dynamically adjusts the polynomial degree as the iteration progresses, thereby reducing the number of sparse matrix–vector multiplications (SpMVs). We provide a theoretical convergence analysis and a practical strategy for selecting the polynomial degree. The performance is further improved through optimized sparse–dense matrix multiplication (SpMM) and efficient deflation techniques. Extensive experiments on Intel and ARM HPC systems demonstrate that AdaPolySI outperforms state-of-the-art solvers, including our implementation of the CJ-FEAST method as well as polynomial subspace method in EVSL and SLEPc. Both strong and weak scalability results are presented to demonstrate the parallel efficiency of AdaPolySI.

Keywords

Subspace iteration, Adaptive polynomial filtering, Deflation, Chebyshev polynomial, Parallel computing

ACM Reference Format:
Yuhui Ni, Xiaofei Xu, Shengguo Li, Juan Zhang, Juan Chen, Jianchun Wang, and Jose E. Roman. 2026. AdaPolySI: Adaptive Polynomial Filtered Subspace Iteration for Hermitian Interior Eigenvalue Problems. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3797905.3800553

*Both authors contributed equally to this research.
†Corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ICS '26, Belfast, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3800553

1 Introduction

Large-scale sparse symmetric (or complex Hermitian) eigenvalue problems constitute a fundamental class of problems [33] in modern numerical linear algebra. They arise ubiquitously in scientific and engineering computations, particularly in electronic structure calculations and quantum chemistry. In electronic structure simulations, discretized Hamiltonians derived from tight-binding or first-principles models often lead to large-scale sparse Hermitian matrices that describe interactions among atomic orbitals in nanoscale systems such as carbon nanotubes or graphene-based materials [18, 29]. These problems typically require computing eigenpairs near the Fermi level to analyze electronic properties such as the density of states and band structures. In most of these applications, only a small portion of the spectrum is of physical interest, and efficiently extracting the corresponding eigenpairs remains a major computational challenge.

For large-scale partial eigenvalue problems targeting extreme or interior eigenpairs, existing algorithms are generally categorized as projection-based iterative methods. The first class consists of *variable-dimension subspace* methods, which progressively expand the search subspace. Krylov-type methods such as Lanczos [2] and Arnoldi [3, 22] generate a subspace at iteration k of the form $\{v, Av, A^2v, \ldots, A^{k-1}v\}$, where A is the target matrix. Although parallelizable, these methods are inherently sequential and cannot fully exploit modern high-performance architectures. Non-Krylov methods such as Jacobi–Davidson (JD) [37] and LOBPCG [19] alleviate this limitation by employing effective preconditioners and Level-3 BLAS operations [8], making them better suited for large-scale parallel environments. However, both may suffer from numerical instability or breakdown if not carefully implemented [8, 12, 46]. In

practice, these methods are primarily used for computing extremal eigenvalues.

The second class comprises *fixed-dimension subspace* methods, which maintain a constant subspace size throughout iterations. Representative approaches include trace minimization [34, 36] and modern filtered subspace iteration (FSI) methods such as FEAST [26, 30], the Sakurai–Sugiura (SS) method [34], EVSL [23, 24], SLEPc [11, 28] and ChASE [39, 40]. FSI methods are often combined with spectrum slicing, which partitions the spectrum into nonoverlapping intervals that can be processed independently in parallel. As a result, a large number of eigenpairs can be computed in a truly parallel manner. The spectral filters are typically constructed using polynomial functions (as in EVSL and ChASE) or rational functions (as in FEAST and SS). In this work, we focus on polynomial filtering, which avoids solving linear systems required by rational filters and is thus better suited for large-scale parallel environments.

Chebyshev polynomials are widely used to construct spectral filters, as in ChebFD [20, 27], CJ-FEAST [15, 16] and EVSL [23]. Nearly all existing methods rely on fixed-degree filters, with the exception of ChASE [40], which adaptively tunes the polynomial degree per eigenvector according to its residual norm, thereby reducing the overall floating-point operations by up to 20%. Chebyshev filtering in ChASE leverages the minimax property of Chebyshev polynomials, which suppresses values inside a target interval while causing rapid growth beyond it. This makes RACE effective for computing extremal eigenvalues but *cannot* be applied to interior eigenvalue problems without additional modifications.

In this work, we propose `AdaPolySI`, an adaptive polynomial filtered subspace iteration framework for large-scale sparse Hermitian interior eigenvalue problems. Our filter is based on a Chebyshev approximation to the *step function*, similar to CJ-FEAST, but replaces the Jackson damping with a more flexible exponent-μ Lanczos damping. Note that CJ-FEAST was originally proposed for computing the SVD [16], and no high-performance implementations are available. In this work, we extend CJ-FEAST to Hermitian eigenvalue problems and develop a parallel implementation. Beyond CJ-FEAST, we introduce a new criterion for dynamically selecting the polynomial degree, which differs from the strategy used in ChASE. In particular, we adopt exponent-μ Lanczos damping instead of the Jackson damping used in [15], providing a sharper filter (see Figure 2) and integrating more naturally with the adaptive mechanism. Proposition 3.1 establishes the approximation properties of AdaPolySI. We further conduct a series of experiments to show that AdaPolySI is insensitive to the parameter μ and that, compared with EVSL's fixed-degree filters, the adaptive filters require significantly fewer iterations and sparse matrix–vector multiplications (SpMVs). Since the dominant cost of FSI methods is the repeated multiplication of a sparse matrix with multiple vectors—effectively a sparse–dense matrix multiplication (SpMM)—we implement a high-performance SpMM kernel in both CJ-FEAST and AdaPolySI, rather than invoking multiple SpMVs as in EVSL.

AdaPolySI is implemented based on the YHAMG [43] library[1] and therefore naturally supports distributed-memory parallelism. ParMETIS [17] can also be seamlessly integrated to reduce interprocess communication. We compare AdaPolySI with CJ-FEAST, EVSL[2] on shared-memory platforms, and further with SLEPc [28] on distributed-memory systems. Numerical experiments are conducted on two high-performance computing platforms equipped with Intel and ARM processors, respectively. Using real-world matrices from the SuiteSparse [5] and ELSES [13] collections, along with several synthetic test matrices, AdaPolySI achieves a 2.29× average speedup over the fixed-degree CJ-FEAST algorithm and a 5.04× average speedup over EVSL-PSI. On distributed environments, AdaPolySI and CJ-FEAST exhibit strong scalability comparable to that of SLEPc-KS.

The main contributions of this work are summarized as follows:

- **Algorithmic innovation:** We propose an adaptive Chebyshev-filtered subspace iteration method for large-scale Hermitian interior eigenvalue problems, with theoretical analysis of convergence and adaptivity criterion.
- **High-performance Implementation:** A distributed parallel implementation is developed, featuring optimized sparse-dense matrix multiplication (SpMM), implicit deflation to accelerate convergence and spurious eigenvalue detection to avoid redundant iterations.
- **Comprehensive evaluation:** Extensive experiments on both synthetic and real-world sparse matrices demonstrate the accuracy and superior performance of AdaPolySI compared with state-of-the-art solvers.

2 Background

2.1 Notation

Consider a matrix $X \in \mathbb{C}^{m\times n}$. We denote by $X_{(i:j,\,k:l)}$ the submatrix of X consisting of rows i to j and columns k to l. The symbol $\kappa(X)$ denotes the condition number of X, and X^H denotes the conjugate transpose of X. Let $\lambda_{\min}$ and $\lambda_{\max}$ denote the minimum and maximum eigenvalues, respectively. For a set $\mathcal{I}$, $|\mathcal{I}|$ denotes its cardinality. For a subset $\mathcal{J} \subset \mathcal{I}$, $\mathcal{I} \setminus \mathcal{J}$ denotes the complement of $\mathcal{J}$ with respect to $\mathcal{I}$.

2.2 Filtered Subspace Iteration

Given a symmetric (or complex Hermitian) matrix $A \in \mathbb{C}^{n\times n}$ and an interval of interest $[a, b] \subset [\lambda_{\min}, \lambda_{\max}]$, let $\{\lambda_j\}_{j=1}^{n}$ denote the eigenvalues of A with corresponding eigenvectors v_j satisfying $\langle v_i, v_j\rangle = \delta_{ij}$. We consider

$$Av_j = \lambda_j v_j, \tag{1}$$

where $\lambda_j \in [a, b]$.

The modern Subspace Iteration (SI) framework has gone through a long period of development. A limitation of subspace iteration methods is their inability to target eigenpairs in arbitrary intervals of interest, i.e. solving the interior eigenvalue problem (1). Filtering techniques [33] provide an effective solution. The technique constructs a function $\rho(x)$ that maps target eigenvalues of A to dominant eigenvalues of $\rho(A)$, while suppressing non-target eigenvalues to near zero. By replacing A with $\rho(A)$ in subspace iterations, we can approximate the invariant subspace corresponding to the

[1] YHAMG is an industry-grade, high-performance algebraic multigrid (AMG) package that provides a wide range of efficient sparse-matrix operations, delivering performance comparable to or exceeding that of HYPRE [42, 44].

[2] pEVSL, the MPI version of EVSL, is still under development and does not yet implement polynomial-filtered subspace methods.

Algorithm 1: Filtered Subspace iteration framework for Hermitian eigenvalue problem.

Input: The Hermitian matrix A, the orthogonal matrix $\hat{V}^{(0)}$, the filtering function $\rho(x)$ and the interval of interest $[a, b]$.
Output: Computed eigenpairs $(\hat{\lambda}_j, \hat{v}_j)$, $\hat{\lambda}_j \in [a, b]$.

1 $i = 1$
2 **while** *not converged* **do**
3 $\quad \hat{X}^{(i)} = \rho(A)\hat{V}^{(i-1)}$
4 $\quad Q^{(i)} = \text{orth}(\hat{X}^{(i)})$
5 $\quad [\hat{V}^{(i)}, \hat{\Lambda}^{(i)}]$= Rayleigh-Ritz($A$, $Q^{(i)}$)
6 $\quad \hat{\lambda}_j = \hat{\Lambda}^{(i)}_{j,j}$, $\hat{v}_j = \hat{V}^{(i)}_{:,j}$
7 $\quad i = i + 1$
8 **end**
9 **function** `Rayleigh-Ritz`(A, Q)
10 $\quad B = Q^H AQ$
11 $\quad$ Compute eigen-decomposition $B = \hat{U}\hat{\Lambda}\hat{U}^H$
12 $\quad \hat{V} = Q\hat{U}$
13 $\quad$ **return** $\hat{V}$, $\hat{\Lambda}$
14 **end**

desired eigenvalues. As far as we know, existing filters fall into two primary types – rational filters and polynomial filters. Our algorithm employs the latter approach, detailed in Section 2.3.

The filtered subspace iteration (FSI) framework for Hermitian eigenvalue problem pseudo-code is shown in Algorithm 1, where it is assumed that a suitable filtering function $\rho(x)$ has already been established. The columns of the $n \times p$ orthogonal matrix $\hat{V}^{(0)}$ form an initial set of orthonormal basis vectors for the subspace. The FSI algorithm may fail to converge when $p < e$, where e denotes the number of eigenvalues in the interval of interest [26]. Therefore, it is necessary to ensure $p \geq e$.

The main computational overhead of FSI comes from Line 3 of Algorithm 1. The subspace basis is orthogonalized using QR decomposition in Line 4. We denote by orth(X) the orthonormal matrix from the QR factorization. In Line 5, Rayleigh-Ritz process projects A onto a small-dimension subspace resulting in a smaller matrix B whose eigenvalues approximate some eigenvalues of A. This method is widely applied in various solution frameworks [16, 19, 24]. We compute all eigenpairs of B using the QR or divide-and-conquer algorithm [6] in LAPACK [1].

2.3 Polynomial Filtering

The idea of the filter is to construct a function $\rho(x)$, such that values outside the region of interest S are close to 0. For the interior eigenvalue problem (1), a suitable choice is the step function

$$s(x) = \begin{cases} 1, & x \in S, \\ 0, & x \notin S. \end{cases}$$

Let the spectral decomposition of Hermitian matrix A be

$$A = \sum_{j=1}^{n} \lambda_j v_j v_j^H,$$

where (λ_i, v_i) are the eigenpairs of A. Suppose $\lambda_1, \cdots, \lambda_e \in [a, b]$ and $\lambda_{e+1}, \cdots, \lambda_n \notin [a, b]$, then

$$s(A) = \sum_{j=1}^{n} s(\lambda_j) v_j v_j^H = \sum_{j=1}^{e} v_j v_j^H.$$

In other words, $s(A)$ is equal to the orthogonal projector onto $\text{span}\{v_1, \cdots, v_e\}$. However, it is impractical to compute $s(A)$ in implementation. Some works have shown how to approximately construct suitable filters [15, 24, 30]. In the remainder of this section, we will show how to approximate $s(x)$ using polynomial series.

Let $\lambda_{\min}$ and $\lambda_{\max}$ denote the minimum and maximum eigenvalues of A, respectively. Without loss of generality, let $S = [a, b] \subset [-1, 1]$. For the general case, we introduce the linear transformation

$$l(x) = \frac{2x - \lambda_{\max} - \lambda_{\min}}{\lambda_{\max} - \lambda_{\min}} = l_1 x + l_2 \tag{2}$$

and replace A with $l(A)$. Following the approach in CJ-FEAST [15], we redefine the step function as

$$s(x) = \begin{cases} 1, & x \in (a, b), \\ \frac{1}{2}, & x = a \text{ or } b, \\ 0, & x \in [-1, 1] \setminus [a, b]. \end{cases} \tag{3}$$

This ensures that the Chebyshev expansion

$$s(x) = \lim_{k\to\infty} \rho_k(x) = \lim_{k\to\infty} \sum_{j=0}^{k} c_j T_j(x),$$

where the Fourier coefficients are

$$c_j = \begin{cases} \frac{1}{\pi}(\arccos(a) - \arccos(b)), & j = 0, \\ \frac{2}{\pi}\left(\frac{\sin(j\arccos(a)) - \sin(j\arccos(b))}{j}\right), & j > 0, \end{cases} \tag{4}$$

and $T_j(x)$ denotes the j-th Chebyshev polynomial of the first kind. As k increases, $\rho_k(x)$ converges to $s(x)$ point by point, but there are Gibbs oscillations [7] at the discontinuities. One approach to mitigate this effect is to introduce a damping factor. The damping factors, denoted by $d_{j,k}$, depend only on the current index j and the maximum polynomial degree k. By incorporating these coefficients into $\rho_k(x)$, we obtain the damped Chebyshev series of degree k, expressed as

$$\rho_k^D(x) = \sum_{j=0}^{k} c_j d_{j,k} T_j(x). \tag{5}$$

For $j > 0$, we consider the following damping factors:

- **Jackson damping factors** [15, 32]:

$$\begin{aligned} d_{j,k}^J = &\frac{(k+2-j)\sin\left(\frac{\pi}{k+2}\right)\cos\left(\frac{j\pi}{k+2}\right)}{(k+2)\sin\left(\frac{\pi}{k+2}\right)} \\ &+ \frac{\cos\left(\frac{\pi}{k+2}\right)\sin\left(\frac{j\pi}{k+2}\right)}{(k+2)\sin\left(\frac{\pi}{k+2}\right)}. \end{aligned} \tag{6}$$

- **Lanczos damping factors (or σ-factors)** [21]:

$$d_{j,k}^L = \frac{\sin\left(\frac{j\pi}{k+1}\right)}{\frac{j\pi}{k+1}}. \tag{7}$$

Algorithm 2: Matrix-matrix multiplication of the polynomial filter.

Input: The Hermitian matrix A, $\{c_j\}_{j=0}^{k}$, $\{d_{j,k}\}_{j=0}^{k}$, $\{l_1, l_2\}$ and the matrix X.
Output: The matrix Y

$Y_1 = X$
$Y_2 = l_1 AX - l_2 X$
$Y = c_0 d_{0,k} Y_1 + c_1 d_{1,k} Y_2$
for $j = 2, 3, \cdots, k$ **do**
  $Y_{\text{temp}} = 2(l_1 A Y_2 - l_2 Y_2) - Y_1$
  $Y_1 = Y_2$
  $Y_2 = Y_{\text{temp}}$
  $Y += c_j d_{j,k} Y_2$
end

- **Exponent-μ Lanczos damping factors** [21]:

$$d_{j,k}^{\mu} = \left(\frac{\sin\left(\frac{j\pi}{k+1}\right)}{\frac{j\pi}{k+1}} \right)^{\mu}, \tag{8}$$

where the parameter $\mu \geq 0$ controls the strength of damping.

In practice, we first compute the Fourier coefficients $\{c_j\}_{j=0}^{k}$, the damping factors $\{d_{j,k}\}_{j=0}^{k}$ and linear transformation factors $\{l_1, l_2\}$ by equations (4), (6)-(8) and (2), respectively. Then implicitly compute $\rho_k^D(l(A)) \cdot X$ by Algorithm 2, where X is a vector or matrix. This process utilizes the three-term recurrence of the Chebyshev polynomial of the first kind, i.e.

$$\begin{cases} T_0(x) = 1,\ T_1(x) = x, \\ T_{j+1}(x) = 2xT_j(x) - T_{j-1}(x),\ j > 1. \end{cases}$$

3 Our approach

In this section, our approach is presented within the framework of filtered subspace iteration. Notably, the approach generalizes to other projection-based frameworks, such as filtered Lanczos methods [24] and Davidson-type methods [46].

3.1 Filter Construction

The construction of the filter depends on the spectral interval of interest. In theory, the faster the filter function varies near the endpoints of the interval, the faster the convergence of the filtered subspace iteration. When polynomial filters are employed, this typically requires a sufficiently high polynomial degree. However, an excessively high degree increases the computational cost without delivering proportional benefits.

For the choice of polynomial degree, the polynomial degree is either fixed throughout the computation or adaptively adjusted at each iteration. When using a fixed polynomial degree, one must carefully balance computational efficiency against approximation accuracy for the corresponding problems and algorithms. Significant research has addressed this trade-off, as demonstrated in various works [15, 24]. In contrast, adaptively adjusted filters offer a more flexible alternative. These methods dynamically modify the polynomial degree based on information from iterations, potentially providing a better balance between accuracy and efficiency. Recently, some works have shown that adaptive strategy can effectively improve the performance of some algorithms [39, 45].

Our approach adopts the latter option. The construction of the filter consists of two steps. In this subsection, we develop a polynomial filter $\rho_k(x)$ that approximates the step function $s(x)$. As the polynomial degree k increases, $\rho_k(x)$ converges to $s(x)$. The next subsection introduces the adaptive strategy.

We select the polynomial filter (5) with the exponent-μ Lanczos damping (8), denoted $\rho_k^{\mu}(x)$. This damping can be flexibly adjusted by controlling μ, and it is simple to compute. Moreover, $\rho_k^{\mu}(x)$ is the k-th degree approximation of $s(x)$ in $\|\cdot\|_w$ induced by the Chebyshev inner product

$$\langle \psi_p, \psi_q \rangle_w = \int_{-1}^{1} \frac{\psi_p(s)\psi_q(s)}{\sqrt{1-s^2}} ds.$$

The following proposition provides an error bound.

Proposition 3.1. *Let $\rho_k^{\mu}(x)$ be the Chebyshev series of degree k with the exponent-μ Lanczos damping to approximate the step function $s(x)$. The damping factors are defined as*

$$d_{j,k}^{\mu} = \begin{cases} 1, & j = 0, \\ \left(\frac{\sin(j\alpha_k)}{j\alpha_k}\right)^{\mu}, & j > 0, \end{cases}$$

where $\alpha_k = \frac{\pi}{k+1}$ and $\mu \geq 0$. Then, the squared weighted L_2-error between $\rho_k^{\mu}(x)$ and $s(x)$ satisfies

$$\|\rho_k^{\mu}(x) - s(x)\|_w^2 \leq \frac{\pi^3\mu^2 + 4\pi}{3(k+1)}. \tag{9}$$

Proof. See Appendix 8.1. □

When $\mu = 0$, the function $\rho_k^{\mu}(x)$ reduces to the classical Chebyshev polynomial approximation $\rho_k(x)$ without damping, obtained through L_2-norm minimization. Consequently, the bound in equation (9) simplifies to that of the standard Chebyshev approximation, as presented in Proposition 3.1 of [7]. However, the Gibbs phenomenon reduces the approximation accuracy of Chebyshev expansions near discontinuity points. Conversely, larger values of μ require higher k to approximate $s(x)$, resulting in increased rounding errors in practice. With an appropriate choice of μ, the exponent-μ Lanczos damping can achieve effects similar to those of other damping schemes. Specifically, when $\mu = 1$, it reduces to the Lanczos damping (7), and when μ lies between 2 and 3 with a sufficiently large polynomial degree k, it closely approximates the Jackson damping (6), as shown in Figure 1.

In our approach, controlling $\mu \in (0, 1]$ allows us to suppress the Gibbs phenomenon while selecting the **smaller** degree k than other damping factors such as Jackson damping (6). With the parameter fixed at $\mu = 0.5$, Figure 2 compares the Chebyshev polynomial approximations to the step function using the exponent-μ Lanczos damping and the Jackson damping for polynomial degrees $k = 50$ and $k = 100$. For the same polynomial degree, exponent-μ Lanczos damping provides a more accurate filter than Jackson damping.

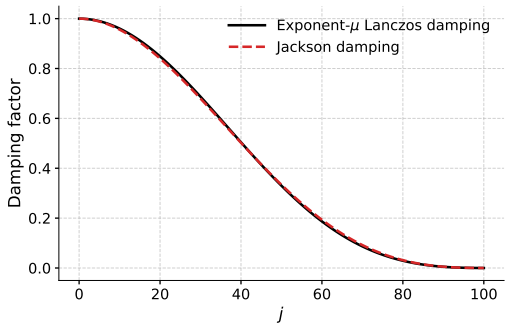


Figure 1: Comparison between the exponent-μ Lanczos damping and the Jackson damping with $\mu = 2.51$ and $k = 100$.

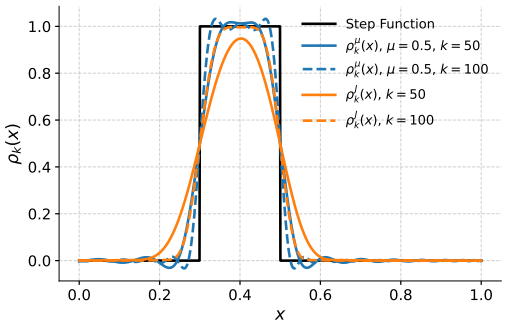


Figure 2: Comparison of Chebyshev approximations with exponent-μ Lanczos ($\mu = 0.5$) and Jackson damping for degrees $k = 50$ and $k = 100$; the black curve denotes the step function $s(x)$ with $[a, b] = [0.3, 0.5]$.

3.2 Adaptive Strategy

A typical representative of adaptive filtering is the Chebyshev polynomial–based adaptive filter, exemplified by the ChASE algorithm [39, 40]. In ChASE, the polynomial degree is adjusted at each iteration according to the residual norm of each eigenpair. Unlike ChASE, we design an adaptive strategy that dynamically adjusts according to the *observed convergence rate* of the filtered subspace iteration.

Let the polynomial degree at the $(i+1)$-th iteration be denoted by $k^{(i+1)}$ (where $k^{(i+1)} \le k$). The first objective is to construct a polynomial $\hat{q}_{k^{(i+1)}}(x)$ of degree $k^{(i+1)}$ such that

$$\hat{q}_{k^{(i+1)}}(x) = \arg \min_{q_{k^{(i+1)}} \in \mathbb{P}_{k^{(i+1)}}} \|q_{k^{(i+1)}} - \rho_k^\mu\|_w^2, \tag{10}$$

where $\mathbb{P}_{k^{(i+1)}}$ denotes the space of polynomials of degree at most $k^{(i+1)}$. Let $q_{k^{(i+1)}}(x) = \sum_{j=0}^{k^{(i+1)}} s_j T_j(x)$. Then we have

$$\|q_{k^{(i+1)}} - \rho_k^\mu\|_w^2 = \sum_{j=0}^{k^{(i+1)}} (s_j - c_j d_{j,k}^\mu)^2 + \sum_{j=k^{(i+1)}+1}^{k} (c_j d_{j,k}^\mu)^2.$$

Therefore, the optimization problem (10) is minimized when $s_j = c_j d_{j,k}^\mu$. In other words, the polynomial $\rho_k^\mu(x)$ can be decomposed as

$$\begin{aligned}\rho_k^\mu(x) &= \sum_{j=0}^{k} c_j d_{j,k}^\mu T_j(x) \\ &= \sum_{j=0}^{k^{(i+1)}} c_j d_{j,k}^\mu T_j(x) + \sum_{j=k^{(i+1)}+1}^{k} c_j d_{j,k}^\mu T_j(x) \\ &= \rho_{k^{(i+1)},k}^\mu(x) + r^{(i+1)}(x),\end{aligned}$$

We obtain the optimal solution $\hat{q}_{k^{(i+1)}} = \rho_{k^{(i+1)},k}^\mu$.

Next, a strategy is required to determine the degree of the adaptive polynomial. Let the eigenvalues of $\rho_{k^{(i+1)},k}^\mu(l(A))$ be denoted by $\gamma_j^{(i+1)} = \rho_{k^{(i+1)},k}^\mu(l(\lambda_j))$, arranged in nonincreasing order as

$$\left|\gamma_1^{(i+1)}\right| \ge \cdots \ge \left|\gamma_e^{(i+1)}\right| \ge \cdots \ge \left|\gamma_p^{(i+1)}\right| \ge \cdots \ge \left|\gamma_n^{(i+1)}\right|,$$

where p denotes the dimension of the subspace. According to the convergence theory of subspace iteration[15, 33], the convergence behavior at the $(i+1)$-th iteration is primarily governed by the ratio $\gamma_{p+1}^{(i+1)}/\gamma_e^{(i+1)}$. Since λ_e and λ_{p+1} are unavailable a priori, we approximate them using *Ritz* values produced at the i-th iteration. Let $\{\hat{\lambda}_j^{(i)}\}_{j=1}^p$ be the Ritz values computed at iteration i, and let $e^{(i)}$ denote the number of those Ritz values that lie in the interval $[a, b]$. We set an adaptive threshold τ_a. The adaptive update then uses the approximate criterion

$$\left|\frac{\rho_{k^{(i+1)},k}^\mu(\hat{\lambda}_p^{(i)})}{\rho_{k^{(i+1)},k}^\mu(\hat{\lambda}_{e^{(i)}}^{(i)})}\right| < \tau_a.$$

In practice, the polynomial degree $k^{(i+1)}$ is usually chosen to be smaller than the initial degree k. The adaptive selection strategy for $k^{(i+1)}$ is described in Algorithm 3. By dynamically adjusting $k^{(i+1)}$ across iterations, the algorithm achieves a favorable balance between computational efficiency and filtering effectiveness in suppressing non-target eigenvalues.

Algorithm 3: Compute the polynomial degree of the filter in the $(i+1)$-th iteration.

Input: The Ritz values $\hat{\lambda}^{(i)} = [\hat{\lambda}_1^{(i)}, \cdots, \hat{\lambda}_p^{(i)}]$, estimated eigenvalue count $e^{(i)}$ in $[a, b]$, $\{c_j\}_{j=0}^k$, $\{d_{j,k}\}_{j=0}^k$, $\{l_1, l_2\}$ and the adaptive threshold τ_a

Output: Polynomial degree $k^{(i+1)}$ for the next iteration filter

$h_0 = 1$, $l(\hat{\lambda}^{(i)}) = l_1\lambda^{(i)} + l_2$, $h_1 = l(\hat{\lambda}^{(i)})$.
$\hat{\gamma}^{(i)} = c_0 d_{0,k} h_0$.
for $j = 1, 2, \cdots, k$ **do**
  $\hat{\gamma}^{(i)} += c_j d_{j,k} h_1$.
  Sort $\hat{\gamma}^{(i)}$ descending such that $\left|\hat{\gamma}_1^{(i)}\right| \ge \cdots \ge \left|\hat{\gamma}_p^{(i)}\right|$.
  if $\left|\hat{\gamma}_p^{(i)}/\hat{\gamma}_{e^{(i)}}^{(i)}\right| < \tau_a$ **then**
    $k^{(i+1)} = j$.
    return $k^{(i+1)}$.
  end
  $h_{\text{next}} = 2l(\hat{\lambda}^{(i)}) \cdot h_1 - h_0$.
  $h_0 = h_1$; $h_1 = h_{\text{next}}$.
end

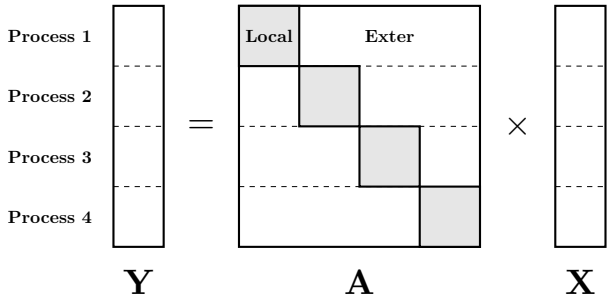


Figure 3: $Y = AX$ **computed by 4 processes.**

3.3 Implementation Details

The complete pseudo-code of the proposed algorithm is presented in Algorithm 4. It is built upon the basic framework of Algorithm 1, with several key enhancements introduced to improve its efficiency and robustness. This subsection describes the implementation details and highlights the innovations incorporated into the algorithmic framework.

3.3.1 Initialization. Before the algorithm starts, several parameters need to be initialized. To obtain the coefficients of the polynomial filter, we first estimate the spectral bounds of matrix A, which can be achieved by running a few steps of the Lanczos algorithm [6]. In addition, the subspace dimension p must be estimated prior to the iteration and chosen larger than the number of eigenvalues in the interval of interest. Here, we estimate p via a trace estimation [7] of the filtered matrix $\rho_k^\mu(A)$. After determining p, the initial approximate basis $\hat{V}^{(0)}$ is randomly generated from a Gaussian distribution with zero mean and unit variance, i.e., $\mathcal{N}(0,1)$. We use Cholesky–QR [41] for orthogonalization, which is more efficient for distributed parallel computing than Householder–QR [33].

3.3.2 ParMETIS plus SpMM. A distinctive feature of FSI with polynomial filtering is that its computational complexity is dominated by matrix–vector multiplications with the large-scale sparse matrix. Moreover, since the subspace dimension p is typically larger than one, most computations fall into the BLAS-3 level. Algorithm 4 involves multiple sparse matrix–matrix multiplications with the fixed matrix A, making the efficiency of these SpMM operations critical to overall performance and naturally motivating the preprocessing of A.

In the distributed-memory parallel implementation of Algorithm 4, we apply the ParMETIS_V3_PartKway routine [17] to reorder the matrix A so that its nonzeros are clustered near the diagonal. This preprocessing improves data locality and reduces inter-process communication, thereby enhancing SpMM scalability, as illustrated in Figure 3.

3.3.3 Exclude Spurious Eigenvalues. In line 13 of Algorithm 4, as the iteration proceeds, some Ritz values with large residuals may remain within the interval of interest. These are referred to as *spurious eigenvalues*. If such Ritz pairs are included in the convergence check, they may interfere with convergence detection and lead to unnecessary iterations. To address this, we introduce a relatively large threshold τ_s such that any eigenpairs whose residuals remain above this value are excluded from the convergence criterion. These details are shown in Algorithm 4, lines 16–21.

Algorithm 4: The AdaPolySI algorithm

Input: The n-by-n symmetric (Hermitian) matrix A, the initial degree of Chebyshev filter $k^{(0)}$, the interval of interest $[a,b]$ and the thresholds τ_a, τ_s, τ_c.
Output: The eigenpairs (Λ, V) of matrix A in $[a,b]$.

1 Reordering A by ParMetis [17].
2 Compute $\{c_j\}_{j=0}^{k}$, $\{d_{j,k}\}_{j=0}^{k}$, $\{l_1, l_2\}$ and p.
3 Select an n-by-p order orthogonal matrix $\hat{V}^{(0)}$. Set $\Lambda = [\ ]$, $V = [\ ]$, $n_{\text{lock}} = 0$, $n_{\text{check}} = 0$.
4 **for** $i = 1, 2, \cdots$ **do**
5 $\quad \hat{X}^{(i)} = \rho_{k^{(i)},k}^{\mu}(A) \cdot \hat{V}^{(i-1)}$. (see Algorithm 2)
6 $\quad \hat{Q}^{(i)} = \text{QR}([V, \hat{X}^{(i)}])$. (see [41])
7 $\quad Q^{(i)} = \hat{Q}^{(i)}_{(:,(n_{\text{lock}}+1):p)}$.
8 $\quad [\hat{V}^{(i)}, \hat{\Lambda}^{(i)}] = \text{Rayleigh-Ritz}(A, Q^{(i)})$.
9 $\quad$ // Find all Ritz pairs in the interval.
10 $\quad \mathcal{I}^{(i)} = \left\{ j : \hat{\Lambda}^{(i)}_{j,j} \in [a,b] \right\}$.
11 $\quad$ **if** $\mathcal{I}^{(i)}$ *is empty* **then**
12 $\quad\quad$ **return** (Λ, V).
13 $\quad$ **end**
14 $\quad$ Compute the filter degree $k^{(i+1)}$ by Algorithm 3.
15 $\quad$ Compute $\|r_j\|_2$ $(j \in \mathcal{I}^{(i)})$ by (11).
16 $\quad$ // Exclude spurious eigenpairs.
17 $\quad C^{(i)} = \left\{ j \in \mathcal{I}^{(i)} : \|r_j\|_2 < \tau_s \right\}$.
18 $\quad n_{\text{check}}^{(i)} = |C^{(i)}|$, $n_{\text{check}}^{(i)} \mathrel{+}= n_{\text{lock}}$.
19 $\quad$ **if** $n_{\text{check}}^{(i)} > 0$ *and equal to* $n_{\text{check}}^{(i-1)}$ **then**
20 $\quad\quad \mathcal{I}^{(i)} = C^{(i)}$.
21 $\quad$ **end**
22 $\quad$ // Locking the converged eigenpairs.
23 $\quad \mathcal{L}^{(i)} = \left\{ j \in \mathcal{I}^{(i)} : \|r_j\|_2 < \tau_c \right\}$, $n_{\text{lock}}^{(i)} = |\mathcal{L}^{(i)}|$.
24 $\quad \Lambda := \text{diag}(\Lambda, \hat{\Lambda}^{(i)}_{(\mathcal{L}^{(i)},\mathcal{L}^{(i)})})$, $V := [V, V^{(i)}_{(:,\mathcal{L}^{(i)})}]$.
25 $\quad \hat{V}^{(i)} := \hat{V}^{(i)}_{(:,\,(\mathcal{L}^{(i)})^c)}$, $n_{\text{lock}} \mathrel{+}= n_{\text{lock}}^{(i)}$.
26 $\quad$ **if** $\mathcal{I}^{(i)} \setminus \mathcal{L}^{(i)}$ *is empty* **then**
27 $\quad\quad$ **return** (Λ, V).
28 $\quad$ **end**
29 **end**

3.3.4 Deflation. Deflation is a common strategy in eigenvalue computations that reduces the problem dimension by removing the influence of already computed eigenvalues. The locking technique [33, 39] adopted in this work represents an implicit form of deflation. In Algorithm 4 (lines 6–7 and 22–25), once some eigenvalues have converged, their corresponding eigenvectors are locked to prevent further updates. This effectively reduces the dimension of the search subspace for subsequent iterations and eliminates redundant computation in the filtering step (line 5).

Figure 4 shows partial implementation details of Algorithm 4, where the numbers represent the indices of eigenpairs. Line 23 moves the converged eigenpairs to the front of the ordering, whereas lines 13 and 17 only record the corresponding indices. Eigenpairs marked with red indices are excluded from the convergence check but are carried over to the next iteration.

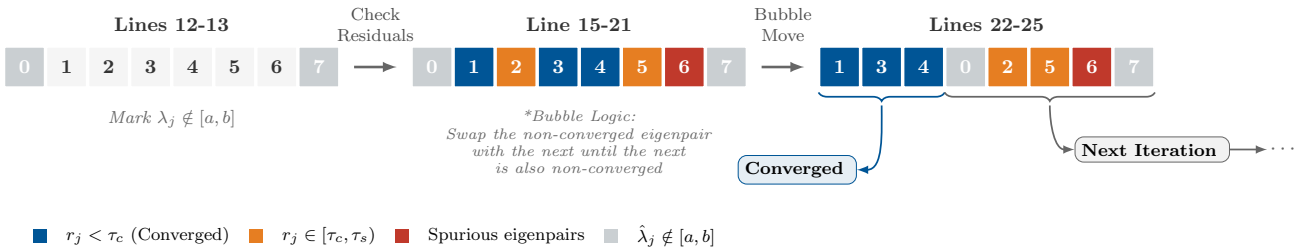


Figure 4: Partial execution details of Algorithm 4. The numbers indicate the indices of eigenpairs. The figure illustrates how the ordering of eigenpairs is updated during some stages of the algorithm. Different types of eigenpairs are marked with distinct colors, as explained in the legend below, where τ_c and τ_s denote the convergence threshold and the spurious eigenpair detection threshold in Algorithm 4, respectively.

4 Experiment Setup

4.1 Evaluation Platforms

The numerical experiments are divided into two parts: (i) verification of AdaPolySI convergence, and (ii) performance evaluation in a high-performance parallel environment. The convergence tests are conducted in MATLAB, while the parallel performance tests use a high-performance implementation integrated into the YHAMG library [43].

The MATLAB experiments are conducted on a workstation equipped with a 13th Gen Intel(R) Core(TM) i5-13400F processor and 32 GB of memory. The C++ experiments are performed on a high-performance computing (HPC) cluster comprising 64 nodes, each with one Intel Xeon Gold 6348 CPUs (28 cores per node, 2.60 GHz) and 128 GB of DDR4-3200 memory, totaling 1,792 CPU cores.

The software environment consists of the Kylin Linux operating system, MPICH 4.1.2, and the Intel 2023 compiler with the `-Ofast` optimization flag enabled. Intel MKL 2023.0 is employed for mathematical routines, while ParMETIS 4.0.3 is used for domain decomposition and load balancing in multi-process MPI settings only. Each test is executed five times, and the average values are reported to ensure reproducibility.

4.2 Workloads

The first seven columns of Table 1 list information about the matrices and the corresponding intervals of interest used in our evaluation. The sixth and seventh columns denote the number of eigenvalues in the interval of interest and their ratio to the total number of eigenvalues, respectively. These matrices originate from diverse application domains, including electronic structure calculations, quantum chemistry, structural and 2D/3D problems. They are collected from the **SuiteSparse** [5] and **ELSES** [13] matrix collections, with sizes ranging from several hundred thousand to over one million. Therefore, they are suitable for assessing the generalization capability of our algorithm.

4.3 Baseline Solvers

As baselines, we evaluate three representative polynomial filtering subspace iteration algorithms: CJ-FEAST [15, 16], EVSL-PSI [23] and SLEPc [11]. CJ-FEAST was originally developed for SVD computations and we naturally extend it to symmetric eigenvalue problems. To ensure a fair comparison, CJ-FEAST is implemented within our enhanced subspace iteration framework (Algorithm 4) **without** adapting the polynomial degree. This allows the evaluation to isolate the effectiveness of the filtering scheme itself, rather than differences arising from the algorithmic framework. A performance-optimized implementation of CJ-FEAST and AdaPolySI is developed using YHAMG [43]. EVSL-PSI is taken from the EVSL software package, where the polynomial filter is constructed via a Chebyshev expansion of the Dirac delta function [24]. In this work, we use SLEPc Version 3.24.1.

To ensure a fair comparison with baseline methods, the reported runtimes exclude setup costs, including matrix reordering (via ParMETIS in the multi-process setting) and other common steps such as spectrum estimation, spectral slicing, and data allocation. Only the runtime of the polynomial filtered subspace iteration is reported. Subsection 5.4 further reports the proportion of time associated with ParMETIS in the MPI parallel setting.

4.4 Convergence Criterion

Unless otherwise specified, the convergence tolerance τ_c for the residual norm

$$r_j := \max_j \|Av_j - \lambda_j v_j\|_2 \tag{11}$$

is set to 10^{-8}. Both the adaptive threshold τ_a and the spurious eigenpair detection threshold τ_s are fixed at 10^{-3}. These parameter values were selected based on preliminary testing to balance convergence speed and numerical stability.

5 Numerical experiments

In this section, we present a comprehensive experimental evaluation of AdaPolySI algorithm. All experiments are designed to demonstrate the efficiency and robustness of our method (Algorithm 4) compared to state-of-the-art methods.

5.1 Accuracy and Convergence

We test the accuracy and convergence of AdaPolySI algorithm through some examples with exact solutions implemented in MATLAB. We discretize the one-dimensional Laplace operator $-\Delta$ in $\Omega = [0, 1]$ with Dirichlet boundary conditions using finite difference method, setting the function values at the boundary to zero.

Table 1: Sparse matrices from the SuiteSparse and ELSES collections, along with the spectral intervals of interest and the iteration counts, with the value in parentheses giving the corresponding average degree of polynomials.

ID	Matrix	Rows	#nnz	Interval	#evals	Ratio	EVSL-PSI	CJ-FEAST	AdaPolySI
1	Ge87H76	112,985	7,892,195	$[-0.64, 0.80]$	1039	0.92%	60(17)	11(69)	12(25.0)
2	Ge99H100	112,985	8,451,395	$[-0.65, 0.80]$	1125	1.00%	60(17)	11(69)	13(25.4)
3	Si41Ge41H72	185,639	15,011,265	$[-0.64, 0.60]$	738	0.40%	70(22)	10(107)	13(30.4)
4	Si87H76	240,369	10,661,631	$[-0.66, 0.40]$	677	0.28%	90(22)	11(112)	12(44.8)
5	Ga41As41H72	268,096	18,488,476	$[-0.64, -0.30]$	100	0.04%	35(959)	9(1000)	11(335.8)
6	VCNT40000std	40,000	1,600,000	$[-0.5, -0.35]$	2695	6.74%	45(17)	9(73)	13(20.9)
7	VCNT100000std	100,000	3,999,424	$[-0.64, -0.61]$	1020	1.02%	55(68)	5(461)	12(87.9)
8	VCNT400000std	400,000	15,997,248	$[-0.85, -0.83]$	531	0.13%	105(14)	12(72)	9(47.7)
9	NCCS430080std	430,080	19,165,216	$[-0.8, -0.78]$	191	0.04%	60(44)	6(274)	11(50.1)
10	VCNT1000000std	1,000,000	39,991,680	$[-0.85, -0.84]$	85	0.01%	145(22)	12(119)	7(104.8)

This yields a tridiagonal matrix

$$T_n = \begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}. \tag{12}$$

Since the eigenvalues and eigenvectors of T_n admit explicit expressions, we can readily compute its exact eigenvalues $\{\lambda_i\}$ and eigenvectors $\{v_i\}$. We set the interval of interest to $[0.6, 1.2]$. For each random seed from 1 to 100, a distinct initial subspace $\hat{V}^{(0)}$ is generated. We allow the algorithm to iterate sufficiently until the residual ceases to decrease, and record the corresponding residual, eigenvalue error, and eigenvector error at convergence for each set of experiments. The eigenvalue and eigenvector errors are computed respectively as

$$|\lambda_i - \hat{\lambda}_i| \quad \text{and} \quad 1 - |v_i^H \hat{v}_i|^2, \tag{13}$$

where $\hat{\lambda}_i$ and $\hat{v}_i$ are the computed eigenvalues and eigenvectors respectively. Rows 2-4 of Table 2 show the `Residual` of computed eigenpairs, rows 5-7 shows the accuracy of computed eigenvalues, and rows 8-10 show the orthognality of computed eigenvectors. From Table 2, it can be observed that as the matrix size increases, the maximum, minimum, and average values of these three metrics remain only one to three orders of magnitude above machine precision.

Table 2: Convergence results for the one-dimensional Laplace operator discretization.

	n	500	1000	2000	4000
Residual	Max	1.8×10^{-13}	1.9×10^{-12}	1.9×10^{-12}	8.4×10^{-13}
	Min	5.1×10^{-15}	8.0×10^{-15}	1.3×10^{-14}	2.3×10^{-14}
	AVG	1.4×10^{-14}	4.7×10^{-14}	8.7×10^{-14}	1.3×10^{-13}
Values	Max	1.2×10^{-14}	1.5×10^{-14}	1.9×10^{-14}	2.8×10^{-14}
	Min	3.8×10^{-15}	6.0×10^{-15}	9.2×10^{-15}	1.2×10^{-14}
	AVG	6.2×10^{-15}	8.9×10^{-15}	1.2×10^{-14}	1.7×10^{-14}
Vectors	Max	1.5×10^{-15}	9.4×10^{-16}	1.3×10^{-15}	1.1×10^{-15}
	Min	3.6×10^{-16}	5.0×10^{-16}	6.4×10^{-16}	6.4×10^{-16}
	AVG	6.8×10^{-16}	7.0×10^{-16}	9.3×10^{-16}	8.0×10^{-16}

5.2 Performance analysis

We evaluate ten matrices listed in Table 1. We compare three methods for computing the target eigenpairs: AdaPolySI, CJ-FEAST, and the polynomial subspace iteration method in EVSL (EVSL-PSI). All experiments in this subsection are conducted using a single MPI process and 28 OpenMP threads on a single CPU socket, since EVSL does not support MPI parallelism.

The performance results are presented in Figure 5, which shows the relative runtimes of CJ-FEAST and AdaPolySI with respect to EVSL-PSI across all test matrices. Each bar is divided into two segments representing the runtime of the filtering stage and the remaining operations. The last three columns of Table 1 list, for each algorithm, the number of iterations and the corresponding average polynomial degree (in parentheses).

It can be observed that, except for matrix 6, the filtering stage dominates the total runtime for all other cases, especially for CJ-FEAST and AdaPolySI. This is because matrix 6 has the largest ratio of eigenvalues in the interval of interest to the total matrix dimension, as shown in column 7 of Table 1. For the remaining matrices, the "Others" portion in CJ-FEAST and AdaPolySI is relatively small and comparable, whereas it is significantly larger in EVSL-PSI. The reason is that EVSL-PSI requires more iterations than CJ-FEAST and AdaPolySI, leading to a higher proportion of orthogonalization and Rayleigh–Ritz procedures in the total runtime.

The results in Figure 5 demonstrate that AdaPolySI achieves on average, **5.04**× speedup over the EVSL-PSI eigensolver and **2.29**× over CJ-FEAST, with peak speedups of up to 14.1× and 2.63×, respectively. We note that EVSL also includes other algorithms, such as restarted Lanczos and rational filtering methods; however, in this subsection, we focus specifically on its polynomial filtered subspace iteration method.

There are two primary reasons why our approach (AdaPolySI) outperforms EVSL-PSI in terms of computational efficiency. First, AdaPolySI employs a different polynomial filtering technique that requires significantly fewer sparse matrix-vector multiplications (SpMVs) and iterations than EVSL-PSI. Second, our implementations of both AdaPolySI and CJ-FEAST are based on SpMM, whereas EVSL relies solely on SpMV. SpMM generally offers better data locality and memory access efficiency, making it faster than SpMV in practice. Comparing with CJ-FEAST [16], AdaPolySI adaptively adjusts the polynomial degree and thus requires much fewer SpMVs,

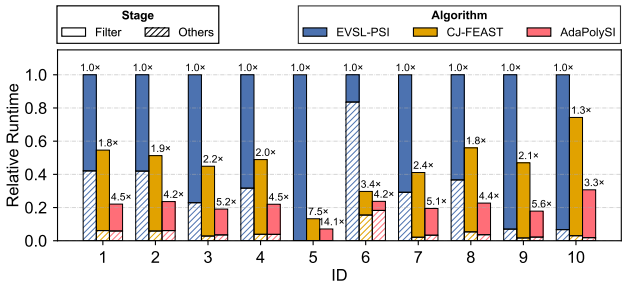


Figure 5: The relative runtime of AdaPolySI and CJ-FEAST over EVSL.

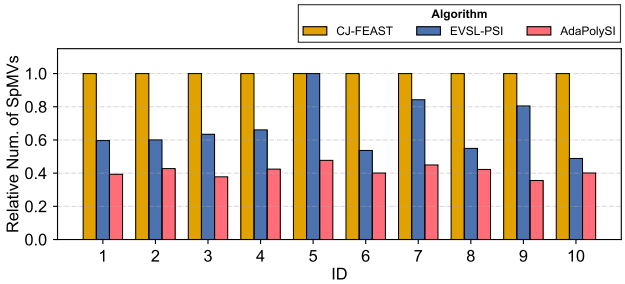


Figure 6: Relative number of SpMVs performed by EVSL-PSI and AdaPolySI with respect to CJ-FEAST. (For matrix 5, EVSL-PSI requires approximately 2.67× more SpMVs than CJ-FEAST; this detail is omitted from the plot for clarity.)

as shown in the last three columns of Table 1 (values in parentheses denote average degree of polynomials).

Figure 6 shows the relative number of SpMVs performed by the three methods, with CJ-FEAST as the baseline. As observed, AdaPolySI consistently requires the fewest SpMVs among all three methods. For most matrices, CJ-FEAST requires more SpMVs than EVSL. However, CJ-FEAST still achieves better performance because it requires far fewer iterations than EVSL-PSI and utilizes SpMM instead of SpMV, which offers improved data locality and computational efficiency.

To further assess the impact of varying the interval of interest, we conducted additional experiments on the matrices `Si41Ge41H72`, `VCNT100000std` and `Ge87H76`. For each matrix, we examined five spectral regions—the two ends of the spectrum and three interior segments. Within each region, we tested two intervals containing approximately 500 and 1000 eigenvalues.

The results in Figure 7 demonstrate that AdaPolySI outperforms CJ-FEAST and EVSL-PSI in almost all cases, achieving an average speedup of 2.32× and 4.87×, respectively, with peak speedups of 3.02× and 11.31×. In general, AdaPolySI and CJ-FEAST tend to exhibit a more pronounced advantage over EVSL-PSI for larger intervals. This is because smaller intervals have closer spectral endpoints, requiring a higher polynomial degree in the Chebyshev expansion to approximate the step function accurately.

5.3 Parameter Sensitivity Analysis

The sensitivity of AdaPolySI with respect to the parameters μ and τ_a is analyzed. We use the test cases with matrices `Ga41As41H72`

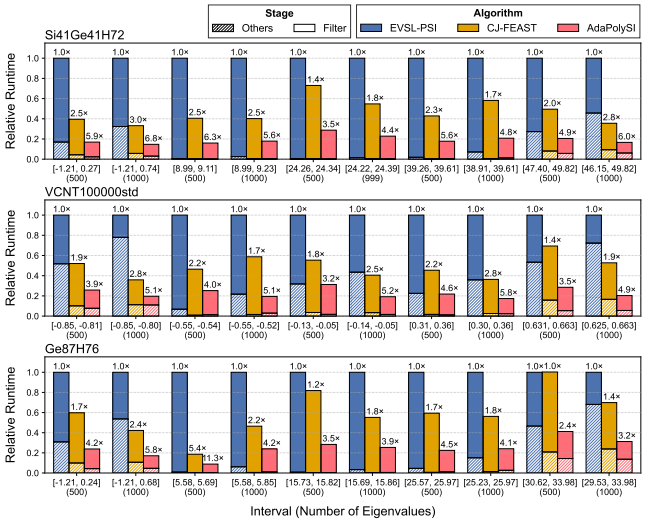


Figure 7: Performance comparison of AdaPolySI, CJ-FEAST, and EVSL-PSI over multiple matrices, spectral regions, and interval sizes.

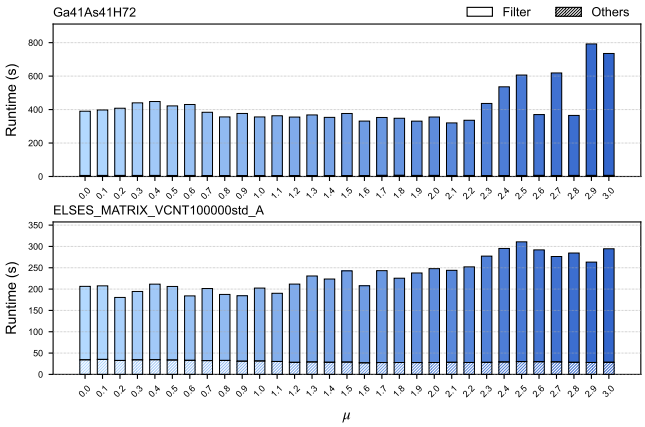


Figure 8: Runtime of AdaPolySI on Ga41As41H72 and VCNT100000std for $\mu = 0 : 0.1 : 3$.

and `VCNT100000std` from Table 1 for this evaluation. All experiments are performed using a single MPI process with 28 OpenMP threads. To examine the effect of μ, we test AdaPolySI over the range $\mu \in [0, 3]$ with a step size of 0.1. The results are shown in Figure 8. The runtime remains nearly unchanged when μ is small, but gradually increases as μ becomes larger. This indicates that choosing a small value of μ leads to similar-and generally better—performance compared with using larger values. Therefore, setting $\mu = 0.5$ is an appropriate choice.

Figure 2 indicates that for a fixed k, an appropriately chosen μ makes the exponent-μ Lanczos damping be close to the Jackson damping. This further suggests that using a Lanczos damping with a small exponent μ offers a performance advantage over the Jackson damping within the AdaPolySI framework.

For the adaptive parameter that determines the polynomial degree, we evaluate τ_a over the range 10^{-4}–10^{-1}, and the results are

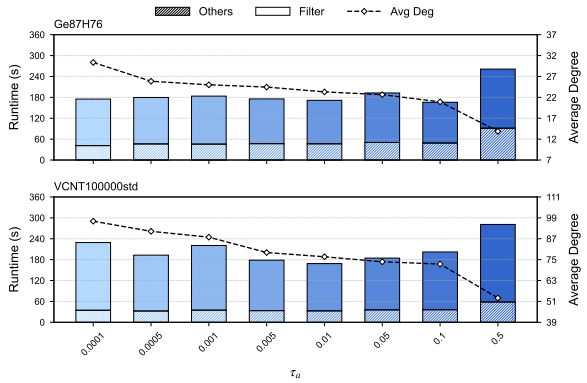


Figure 9: Runtime and average degree of AdaPolySI for different values of τ_a.

Table 3: Timing breakdown form 2 to 64 nodes for $T_{64\times64\times64}$, including ParMETIS time and its share of total runtime.

#Processes	Setup (s)	Solve (s)	ParMETIS (s)	ParMETIS/Total
2	0.7388	8.6305	0.0826	0.88%
4	0.5410	4.0029	0.0596	1.31%
8	0.4274	2.2807	0.0437	1.61%
16	0.3716	0.9811	0.0413	3.05%
32	0.3571	0.6275	0.0514	5.22%
64	0.5625	0.4808	0.0739	7.08%

shown in Figure 9. We observe that the performance is largely insensitive to τ_a, and the overall runtime remains nearly unchanged as long as τ_a is sufficiently small, e.g., less than 0.01. As τ_a increases, the average polynomial degree decreases, which in turn leads to a larger number of iterations. This results in additional computational costs in the orthogonalization and Rayleigh–Ritz stages. In particular, when $\tau_a = 0.5$, the time spent in the "Others" stage increases significantly, rendering the adaptive strategy ineffective. Based on these observations, we select a relatively small default value, $\tau_a = 10^{-3}$.

5.4 Scalability Analysis

In this subsection, we evaluate the strong and weak scalability of AdaPolySI. Each MPI process is bound to a single CPU socket and utilizes 28 OpenMP threads. The primary test matrices correspond to the 3D Laplacian operator discretized by the 7-point finite-difference stencil, $T_{n_1\times n_2\times n_3} = T_{n_1} \otimes T_{n_2} \otimes T_{n_3}$, with matrix size $n = n_1 n_2 n_3$, where n_i $(i = 1, 2, 3)$ denotes the number of grid points in each spatial direction and T_n is defined in equation (12). Prior to the scalability study, we employ the distributed-memory version of AdaPolySI to solve $T_{64\times64\times64}$ over the target interval $[0.007, 0.1]$. We measure the proportion of time spent in ParMETIS relative to both the setup phase and the overall runtime. Detailed timing results are reported in Table 3. The results show that the ParMETIS overhead constitutes only a small fraction of the setup time and accounts for no more than 7.08% of the total runtime.

5.4.1 Strong Scaling. For the matrices $T_{64\times64\times64}$ and $T_{96\times96\times96}$, we consider the interval $[0.007, 0.1]$, which contains 102 and 383 eigenvalues, respectively. For the Laplacian matrix, we perform ten independent runs with different random initializations to mitigate

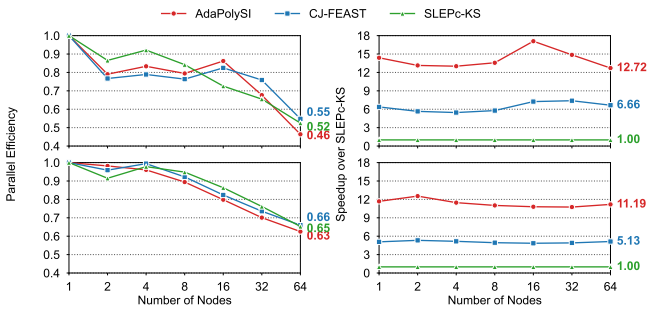


Figure 10: Strong scalability results for the matrix $T_{64\times64\times64}$ (top) and $T_{96\times96\times96}$ (bottom) using AdaPolySI, CJ-FEAST, and SLEPc-KS. The left panel reports the parallel efficiency as the number of nodes increases from 1 to 64, while the right panel shows the speedup of AdaPolySI and CJ-FEAST relative to SLEPc-KS across different node counts.

stochastic variability. We measure the runtime of AdaPolySI, CJ-FEAST, and SLEPc-KS, where SLEPc-KS denotes the Krylov–Schur eigensolver [38] in the SLEPc[3] library [11], equipped with a Lanczos-based polynomial filter [9]. To ensure convergence, the polynomial degree in SLEPc-KS is set to 500 and 77 for $T_{64\times64\times64}$ and $T_{96\times96\times96}$, respectively. The experiments are conducted using between 1 and 64 processors. ParMETIS is used to partition the matrix into multiple subdomains.

The results are presented in Figure 10, which reports both the parallel efficiency and the speedup of each method relative to SLEPc-KS across different processor counts. We observe that the multi-threaded implementation in PETSc provides only limited performance gains. With one thread per MPI process, AdaPolySI achieves an average speedup of 1.51× over SLEPc-KS (Due to page limitations, the results are not shown). The strong-scaling results show that AdaPolySI and CJ-FEAST exhibit nearly identical parallel efficiency curves, as both algorithms are built upon the same computational framework and differ only in their polynomial filtering strategies. As the number of nodes increases, the parallel efficiency of AdaPolySI and CJ-FEAST becomes closer to that of SLEPc-KS, which is particularly evident for $T_{96\times96\times96}$. Moreover, across all tested processor configurations, AdaPolySI consistently attains greater speedups than both CJ-FEAST and S LEPc-KS, demonstrating that its adaptive filtering strategy and overall algorithmic design provide a clear performance advantage.

Additionally, we select two test problems from Table 1 to evaluate the scalability of our approach combining with spectrum slicing strategy. The interval of interest for each matrix is reselected for this experiment. We partition the selected intervals into *two* slices using the DOS algorithm [25], as summarized in Table 4. The DOS algorithm ensures that the numbers of eigenvalues in the two slices are nearly identical, which helps achieve balanced workload during parallel execution. When only a single MPI process is used, the two slices are solved sequentially. When multiple MPI processes are employed, they are evenly assigned to the two slices for concurrent execution.

[3]Since the filtered subspace iteration method in SLEPc is currently less optimized and exhibits noticeably lower performance than Krylov–Schur, we do not include its results in this study.

Table 4: Test matrix and slicing for scalability analysis.

ID	Slice 1	#evals	Slice 2	#evals
5	[-0.5, -0.27]	36	[-0.27, -0.1]	45
9	[-0.8, -0.77]	447	[-0.77, -0.76]	310

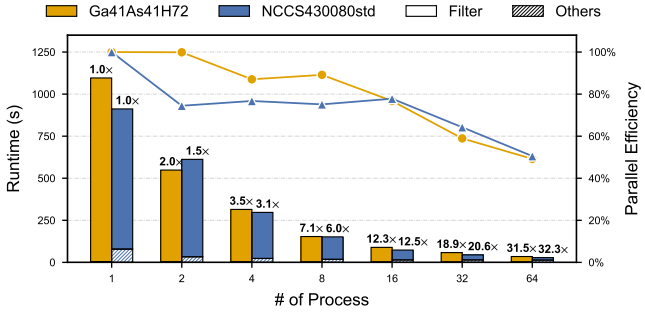


Figure 11: Strong scaling performance of `Ga41As41H72` and `NCCS430080std` with increasing numbers of processes. The numbers above the bars denote the speedup relative to the single-process baseline, while the overlaid lines correspond to the parallel efficiency shown on the right axis.

Figure 11 presents the runtime (bars), speedup (numbers above the bars), and parallel efficiencies (lines) obtained with 1-64 MPI processes. Our approach AdaPolySI demonstrates good scalability. In the case of `NCCS430080std`, for instance, the speedup increases from 1.5× with 2 processes to 32.3× with 64 processes, while the parallel efficiency remains above 50% across all tests.

5.4.2 Weak Scaling. This set of weak-scaling experiments focuses on the matrix $T_{n_1 \times n_2 \times n_3}$ with a fixed interval of interest $[0.6, 1.2]$. For interior eigenvalue problems, as the matrix size increases, the number of eigenvalues contained in the target interval also grows. As a result, the fraction of time spent in the "Others" stage (e.g., orthogonalization and Rayleigh–Ritz procedures) increases accordingly, which makes it difficult to accurately assess the weak scalability of eigensolvers. To mitigate this effect and enable a more meaningful weak-scaling study, we apply the density-of-states (DOS) algorithm [25] to partition the interval of interest into ten slices, and the eigenpairs in these slices are computed sequentially, interval by interval, using multiple processes.

Figure 12 reports the runtime, the parallel efficiency relative to a single node, and the corresponding matrix size for different numbers of nodes. Note that whenever the number of processors is doubled, the number of nonzero entries in the matrix increases by approximately a factor of $\sqrt{2}$. At the same time, the chosen spectral interval ensures that the number of eigenvalues within the interval also grows by about $\sqrt{2}$. As a result, the workload per processor per iteration remains approximately constant. AdaPolySI demonstrates good scalability from 1 to 64 processes.

6 Related Work

Filter-based methods constitute an attractive class of algorithms that reduce large-scale eigenvalue problems to smaller ones by rapidly localizing the target spectral region through spectral filtering. The Sakurai–Sugiura (SS) method [34], which is based on

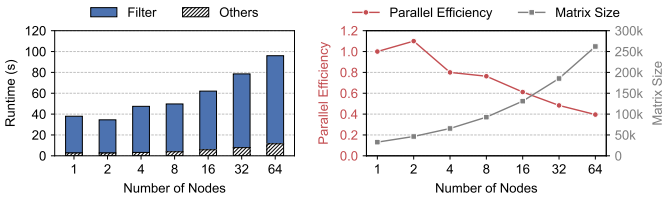


Figure 12: Weak scaling results for the matrix $T_{n_1 \times n_2 \times n_3}$. The left panel shows the runtime breakdown for different numbers of nodes. The right panel reports the parallel efficiency relative to one node (left axis) together with the increasing matrix size $T_{n_1 \times n_2 \times n_3}$ (right axis) under weak scaling.

contour-integral filtering, was proposed in 2003. This method and its variants [14, 35] form the foundation of the z-Pares software package. A closely related rational filtering approach was introduced by Polizzi in 2009 [30], and its subsequent extensions [10, 26] led to the development of the FEAST software package [31].

In contrast to rational filtering methods, polynomial filtering avoids the need to solve linear systems. As a result, the accuracy of polynomial filtering algorithms is independent of the condition number $\kappa(A - \sigma I)$, where σ denotes a spectral shift. This property enables polynomial filtering methods to achieve higher-precision eigenpairs in practice. Li et al. [24] proposed a Chebyshev series expansion of the Dirac delta function as a polynomial filter and incorporated it into both the thick-restart Lanczos method and subspace iteration. This approach is particularly effective for narrow spectral intervals and therefore includes a spectrum-slicing strategy. These high-performance implementations collectively form the EVSL software package [23]. Note that the MPI-based parallel version of EVSL (pEVSL) is still under development and currently does not support the subspace iteration framework.

Similarly, Jia et al. employed the Chebyshev–Jackson series expansion of a step function as a filter within a subspace iteration framework, resulting in the CJ-FEAST algorithm [15] for singular value decomposition (SVD) problems. Their work establishes a pointwise convergence bound for the CJ filter. In most cases, CJ-FEAST demonstrates efficient and stable performance, as well as significantly higher convergence accuracy compared with FEAST [30]. A backward-stable variant of this algorithm was later developed in [16]. In addition, Chebyshev polynomial filtered subspace iteration (CheFSI) has been successfully applied to compute electron densities from discretized Hamiltonians in [4].

7 Conclusions

We propose an adaptive polynomial filter based on the Chebyshev expansion of the step function (3) and integrate it into a filtered subspace iteration framework, referred to as `AdaPolySI`, for solving large-scale Hermitian interior eigenvalue problems. We establish an approximation error bound (9) for the proposed filter under the weighted L^2 norm $\|\cdot\|_w$, yielding an asymptotic convergence rate of $O(1/k)$. In addition, we introduce an implicit deflation strategy within the FSI framework, which effectively reduces the cost of later iterations. For distributed-memory parallelism, we employ ParMETIS to reorder the coefficient matrix at startup, clustering nonzeros near the diagonal and thereby reducing inter-process communication.

Extensive numerical experiments evaluate both the convergence behavior and parallel performance of AdaPolySI. The results demonstrate that AdaPolySI reliably reduces residuals to near machine precision and achieves substantial performance gains—on average, 5.04× speedup over the EVSL-PSI eigensolver and 2.29× over CJ-FEAST, with peak speedups of up to 14.1× and 3.02×, respectively. Moreover, ParMETIS reordering significantly improves distributed-memory efficiency, and AdaPolySI exhibits excellent strong scalability across multiple compute nodes.

In future work, we plan to integrate the adaptive filtering strategy into other eigenvalue solvers, such as filtered Lanczos methods. Further optimization of sparse–dense matrix multiplication (SpMM), particularly on GPU and multi-core CPU architectures, is also of interest, following the approaches in [20]. More broadly, the adaptive filtering concept can be extended to other problems in numerical linear algebra.

Acknowledgments

This work was supported by the National Key R&D Program of China (2023YFB3001702), National Natural Science Foundation of China (No. 62032023, 42104078, 62502048, and 61902411), and the Postgraduate Scientific Research Innovation Project of Hunan Province CX20250935.

References

[1] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. 1999. *LAPACK users' guide*. SIAM.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and editors H. Van der Vost. 2000. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia.

[3] Chris G Baker, Ulrich L Hetmaniuk, Richard B Lehoucq, and Heidi K Thornquist. 2009. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)* 36, 3 (2009), 1–23.

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

[5] T. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25.

[6] James W Demmel. 1997. *Applied numerical linear algebra*. SIAM.

[7] Edoardo Di Napoli, Eric Polizzi, and Yousef Saad. 2016. Efficient estimation of eigenvalue counts in an interval. *Numerical Linear Algebra with Applications* 23, 4 (2016), 674–692.

[8] Jed A Duersch, Meiyue Shao, Chao Yang, and Ming Gu. 2018. A robust and efficient implementation of LOBPCG. *SIAM Journal on Scientific Computing* 40, 5 (2018), C655–C676.

[9] Haw-Ren Fang and Yousef Saad. 2012. A filtered Lanczos procedure for extreme and interior eigenvalue problems. *SIAM Journal on Scientific Computing* 34, 4 (2012), A2220–A2246.

[10] Brendan Gavin, Agnieszka Międlar, and Eric Polizzi. 2018. FEAST eigensolver for nonlinear eigenvalue problems. *Journal of computational science* 27 (2018), 107–117.

[11] Vicente Hernandez, Jose E Roman, and Vicente Vidal. 2005. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)* 31, 3 (2005), 351–362.

[12] Ulrich Hetmaniuk and Rich Lehoucq. 2006. Basis selection in LOBPCG. *J. Comput. Phys.* 218, 1 (2006), 324–332.

[13] Takeo Hoshi. [n. d.]. ELSES matrix library. http://www.elses.jp/matrix/

[14] Tsutomu Ikegami, Tetsuya Sakurai, and Umpei Nagashima. 2010. A filter diagonalization for generalized eigenvalue problems based on the Sakurai–Sugiura projection method. *J. Comput. Appl. Math.* 233, 8 (2010), 1927–1936.

[15] Zhongxiao Jia and Kailiang Zhang. 2023. A FEAST SVDsolver based on Chebyshev–Jackson series for computing partial singular triplets of large matrices. *Journal of Scientific Computing* 97, 1 (2023), 21.

[16] Zhongxiao Jia and Kailiang Zhang. 2024. An augmented matrix-based CJ-FEAST SVDsolver for computing a partial singular value decomposition with the singular values in a given interval. *SIAM J. Matrix Anal. Appl.* 45, 1 (2024), 24–58.

[17] George Karypis, Kirk Schloegel, and Vipin Kumar. 2003. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version* 2 (2003).

[18] Eric Kleinherbers, Thomas Stegmann, and Nikodem Szpak. 2023. Electronic transport in bent carbon nanotubes. *Physical Review B* 107, 19 (2023), 195424.

[19] Andrew V Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing* 23, 2 (2001), 517–541.

[20] Moritz Kreutzer, Dominik Ernst, Alan R Bishop, Holger Fehske, Georg Hager, Kengo Nakajima, and Gerhard Wellein. 2018. Chebyshev filter diagonalization on modern manycore processors and GPGPUs. In *International Conference on High Performance Computing*. Springer, 329–349.

[21] Cornelius Lanczos. 1988. *Applied analysis*. Courier Corporation.

[22] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. 1998. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM.

[23] Ruipeng Li, Yuanzhe Xi, Lucas Erlandson, and Yousef Saad. 2019. The eigenvalues slicing library (EVSL): Algorithms, implementation, and software. *SIAM Journal on Scientific Computing* 41, 4 (2019), C393–C415.

[24] Ruipeng Li, Yuanzhe Xi, Eugene Vecharynski, Chao Yang, and Yousef Saad. 2016. A thick-restart Lanczos algorithm with polynomial filtering for Hermitian eigenvalue problems. *SIAM Journal on Scientific Computing* 38, 4 (2016), A2512–A2534.

[25] Lin Lin, Yousef Saad, and Chao Yang. 2016. Approximating spectral densities of large matrices. *SIAM review* 58, 1 (2016), 34–65.

[26] Ping Tak Peter Tang and Eric Polizzi. 2014. FEAST as a subspace iteration eigensolver accelerated by approximate spectral projection. *SIAM J. Matrix Anal. Appl.* 35, 2 (2014), 354–390.

[27] Andreas Pieper, Moritz Kreutzer, Andreas Alvermann, Martin Galgon, Holger Fehske, Georg Hager, Bruno Lang, and Gerhard Wellein. 2016. High-performance implementation of Chebyshev filter diagonalization for interior eigenvalue computations. *J. Comput. Phys.* 325 (2016), 226–243.

[28] Manuel Pino and Jose E Roman. 2024. Correlated volumes for extended wave functions on a random-regular graph. *Physical Review B* 109, 18 (2024), 184204.

[29] Wolfgang Plank, Rudolf Pfeiffer, Christoph Schaman, Hans Kuzmany, Matteo Calvaresi, Francesco Zerbetto, and Jannik Meyer. 2010. Electronic structure of carbon nanotubes with ultrahigh curvature. *ACS nano* 4, 8 (2010), 4515–4522.

[30] Eric Polizzi. 2009. Density-matrix-based algorithm for solving eigenvalue problems. *Physical Review B—Condensed Matter and Materials Physics* 79, 11 (2009), 115112.

[31] Eric Polizzi. 2020. FEAST eigenvalue solver v4. 0 user guide. *arXiv preprint arXiv:2002.04807* (2020).

[32] Theodore J Rivlin. 1981. *An introduction to the approximation of functions*. Courier Corporation.

[33] Yousef Saad. 2011. *Numerical methods for large eigenvalue problems: revised edition*. SIAM.

[34] Tetsuya Sakurai and Hiroshi Sugiura. 2003. A projection method for generalized eigenvalue problems using numerical integration. *Journal of computational and applied mathematics* 159, 1 (2003), 119–128.

[35] Tetsuya Sakurai and Hiroto Tadano. 2007. CIRR: a Rayleigh-Ritz type method with contour integral for generalized eigenvalue problems. *Hokkaido mathematical journal* 36, 4 (2007), 745–757.

[36] Ahmed Sameh and Zhanye Tong. 2000. The trace minimization method for the symmetric generalized eigenvalue problem. *Journal of computational and applied mathematics* 123, 1-2 (2000), 155–175.

[37] Gerard LG Sleijpen and Henk A Van der Vorst. 2000. A Jacobi–Davidson iteration method for linear eigenvalue problems. *SIAM review* 42, 2 (2000), 267–293.

[38] Gilbert W Stewart. 2002. A Krylov–Schur algorithm for large eigenproblems. *SIAM J. Matrix Anal. Appl.* 23, 3 (2002), 601–614.

[39] Jan Winkelmann, Paul Springer, and Edoardo Di Napoli. 2019. ChASE: Chebyshev Accelerated Subspace iteration Eigensolver for sequences of Hermitian eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)* 45, 2 (2019), 1–34.

[40] Xinzhe Wu, Davor Davidović, Sebastian Achilles, and Edoardo Di Napoli. 2022. ChASE: a distributed hybrid CPU-GPU eigensolver for large-scale hermitian eigenvalue problems. In *Proceedings of the Platform for Advanced Scientific Computing Conference*. 1–12.

[41] Ichitaro Yamazaki, Stanimire Tomov, and Jack Dongarra. 2015. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM Journal on Scientific Computing* 37, 3 (2015), C307–C330.

[42] Xiaojian Yang, Shengguo Li, Fan Yuan, Dezun Dong, Chun Huang, and Zheng Wang. 2023. Optimizing Multi-Grid Computation and Parallelization on Multi-Cores. In *Proceedings of the 37th International Conference on Supercomputing*. ACM, New York, NY, USA, 227–239.

[43] Fan Yuan and Shengguo Li. 2022. YHAMG: High-Performance Hybrid Algebraic Multigrid Solver. https://gitee.com/e-level-parallel-algorithm/yhamg.

[44] Fan Yuan, Xiaojian Yang, Yunqing Huang, Dezun Dong, Chuanfu Xu, Jie Liu, Xiaoqiang Yue, Shengguo Li, and Hongxia Wang. 2025. CRAMG: A Communication-Reduced Algebraic Multigrid Method. *ICS* (2025).

[45] Tianqi Zhang and Fei Xue. 2024. A Chebyshev Locally Optimal Block Preconditioned Conjugate Gradient Method for Product and Standard Symmetric Eigenvalue Problems. *SIAM J. Matrix Anal. Appl.* 45, 4 (2024), 2211–2242.
[46] Yunkai Zhou and Yousef Saad. 2007. A Chebyshev–Davidson algorithm for large symmetric eigenproblems. *SIAM J. Matrix Anal. Appl.* 29, 3 (2007), 954–971.

8 Appendix

8.1 Proof of Proposition 3.1

Compute

$$\begin{aligned}&\|\rho_k^{\mu}(x) - s(x)\|_w^2 \\ &= \|\sum_{j=1}^{k}(d_{j,k}^{\mu} - 1)c_j T_j(x)\|_w^2 + \|\sum_{j=k+1}^{\infty} c_j T_j(x)\|_w^2. \end{aligned} \tag{14}$$

When $0 \le \mu \le 1$, the Maclaurin series ensures that $\cos^{\mu} x \ge 1 - \frac{\mu x^2}{2}$. By Bernoulli inequality, we have

$$(d_{j,k}^{\mu} - 1)^2 = \left(\left(\frac{\sin(j\alpha_k)}{j\alpha_k}\right)^{\mu} - 1\right)^2 \le \left(\cos^{\mu}(j\alpha_k) - 1\right)^2 \le \frac{\mu^2 j^4 \alpha_k^4}{4}.$$

When $\mu > 1$, Bernoulli inequality directly yields

$$\begin{aligned}(d_{j,k}^{\mu} - 1)^2 &= \left(1 - \left(\frac{\sin j\alpha_k}{j\alpha_k}\right)^{\mu}\right)^2 \le \mu^2 \left(1 - \frac{\sin j\alpha_k}{j\alpha_k}\right)^2 \\ &\le \mu^2 \left(1 - \cos(j\alpha_k)\right)^2 = 4\mu^2 \sin^4\left(\frac{j\alpha_k}{2}\right) \le \frac{\mu^2 j^4 \alpha_k^4}{4}.\end{aligned}$$

Thus, the inequality $(d_{j,k}^{\mu} - 1)^2 \le \frac{\mu^2 j^4 \alpha_k^4}{4}$ holds for all $\mu \ge 0$. Consider the first term of (14), we have

$$\begin{aligned}\|\sum_{j=1}^{k}(d_{j,k}^{\mu} - 1)c_j T_j(x)\|_w^2 &= \frac{\pi}{2}\sum_{j=1}^{k}(d_{j,k}^{\mu} - 1)^2 c_j^2 \\ \le \frac{\mu^2 \alpha_k^4}{\pi}\sum_{j=1}^{k} j^2 &\le \frac{\pi^3 \mu^2}{3(k+1)}.\end{aligned} \tag{15}$$

For the second term of (14), we obtain that

$$\|\sum_{j=k+1}^{\infty} c_j T_j(x)\|_w^2 \le \frac{4\pi}{3(k+1)} \tag{16}$$

in [7, Proposition 3.1]. Equation (9) follows directly from the combination of equations (15) and (16).