SCIENCE CHINA
Mathematics

• ARTICLES •

August 2016 Vol. 59 No. 8: 1635–1662
doi: 10.1007/s11425-016-0274-0

Accelerating large partial EVD/SVD calculations by filtered block Davidson methods

ZHOU Yunkai[1,*], WANG Zheng[1] & ZHOU Aihui[2]

[1]*Department of Mathematics, Southern Methodist University, Dallas, TX* 75275*, USA;*
[2]*LSEC, Institute of Computational Mathematics and Scientific/Engineering Computing, Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing* 100190*, China*
Email: yzhou@smu.edu, zwang@smu.edu, azhou@lsec.cc.ac.cn

Received December 13, 2015; accepted April 26, 2016; published online June 12, 2016

Abstract Partial eigenvalue decomposition (PEVD) and partial singular value decomposition (PSVD) of large sparse matrices are of fundamental importance in a wide range of applications, including latent semantic indexing, spectral clustering, and kernel methods for machine learning. The more challenging problems are when a large number of eigenpairs or singular triplets need to be computed. We develop practical and efficient algorithms for these challenging problems. Our algorithms are based on a filter-accelerated block Davidson method. Two types of filters are utilized, one is Chebyshev polynomial filtering, the other is rational-function filtering by solving linear equations. The former utilizes the fastest growth of the Chebyshev polynomial among same degree polynomials; the latter employs the traditional idea of shift-invert, for which we address the important issue of automatic choice of shifts and propose a practical method for solving the shifted linear equations inside the block Davidson method. Our two filters can efficiently generate high-quality basis vectors to augment the projection subspace at each Davidson iteration step, which allows a restart scheme using an active projection subspace of small dimension. This makes our algorithms memory-economical, thus practical for large PEVD/PSVD calculations. We compare our algorithms with representative methods, including ARPACK, PROPACK, the randomized SVD method, and the limited memory SVD method. Extensive numerical tests on representative datasets demonstrate that, in general, our methods have similar or faster convergence speed in terms of CPU time, while requiring much lower memory comparing with other methods. The much lower memory requirement makes our methods more practical for large-scale PEVD/PSVD computations.

Keywords partial EVD/SVD, polynomial filter, rational filter, kernel, graph

MSC(2010) 15A18, 15A23, 15A90, 65F15, 65F25, 65F50

Citation: Zhou Y, Wang Z, Zhou A. Accelerating large partial EVD/SVD calculations by filtered block Davidson methods. Sci China Math, 2016, 59: 1635–1662, doi: 10.1007/s11425-016-0274-0

1 Introduction

The need to compute partial eigenvalue decomposition (PEVD) or partial singular value decomposition (PSVD) arises naturally in many scientific and engineering disciplines, including the electronic structure calculations in materials science (see [24, 36]), and the many spectral methods in statistical and machine learning (see [5, 7, 25, 39, 75]). For the PEVD in this paper, we focus on symmetric matrices. The PSVD can be considered as a special case of symmetric PEVD.

Large eigenvalue or singular value problems are ubiquitous in real-world applications. The matrices formed in realistic applications are usually of large scale. One example is querying a database containing

*Corresponding author

millions of documents by latent semantic indexing (LSI) [13], which requires computing the PSVD of a matrix with millions of columns. Other applications of PSVD include [8,35,37,45,64]. For an example of PEVD, we mention the link prediction problem, which requires computing PEVD of symmetric matrices with dimension of over several millions, derived from graphs of social networks such as Facebook or LinkedIn.

Many algorithms have been proposed for large PEVD and PSVD computations, as represented by the Lanczos-type methods [26, 27, 60], the Davidson-type methods [12, 15, 38, 59, 69, 70, 74], and the variants of subspace iteration [21, 34, 48]. Discussions on many representative algorithms and available solver packages may be found in [4, 54] and the references therein.

While there exists vast literature on numerical methods for eigen-problems, solving very large PEVD and PSVD efficiently remains challenging. One of the particular challenges arises from the large number of eigenpairs or singular triplets required in modern applications, the number may be 5%–20% of the large matrix dimension. Solving such large problems can be very demanding in terms of CPU time and memory usage. The other challenges include nonlinearity, i.e., the matrices are not fixed but instead dynamically updated, such as those encountered in DFT calculations (see [24,36]) or in matrix completion (see [8,37]). This paper does not address this type of nonlinear eigen-problems, but our methods discussed here can be used to either provide good initial vectors for the dynamically updated matrices, or solve the sequence of linearized eigen-problems employed for solving a nonlinear eigen-problem.

We focus on practical algorithms for solving large eigen-problems arising from the latent semantic indexing (LSI) and the application of a class of graph-kernel-based learning methods. These applications typically require computing a large number of the principal eigenpairs (or singular triplets) of sparse matrices with millions of rows and/or columns. Although there are efficient algorithms for finding principal eigenpairs or singular triplets of large matrices, most of them are designed to efficiently converge only a small number of eigenpairs—they can become inefficient or impractical when applied to computing a large number of eigenpairs. The main reason is that such algorithms often need to form a large projection subspace in order to compute many eigenpairs. This leads to high computational cost from orthogonalization and subspace refinement, which often results in overwhelming memory demand. The economical usage of memory is one key factor to make our methods practical for such problems.

Our methods are based on a block Davidson algorithm, which uses a restart scheme to restrict the dimension of the projection subspace [70]. We utilize spectrum filtering techniques for acceleration. Two types of filters are integrated into the Davidson framework. One filter utilizes the property of "fastest growth outside $[-1, 1]$" of the Chebyshev polynomial [74]. This filter is implemented through three-term recurrences associated with the polynomial. The other filter employs the traditional idea of inexact shift-invert (see [6, 16]), realized by applying the conjugate residual method with a fixed total iteration step [53] to solve the shifted linear systems.

The two problems we study are (i) the symmetric PEVD,

$$\boldsymbol{A}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i, \quad i = 1, \ldots, k, \tag{1.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is symmetric, $\lambda_1 \geqslant \cdots \geqslant \lambda_k$ are the largest k eigenvalues of $\boldsymbol{A}$, and $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ are the corresponding eigenvectors; and (ii) the PSVD,

$$\boldsymbol{M}\boldsymbol{v}_i = \sigma_i \boldsymbol{u}_i, \quad i = 1, \ldots, k, \tag{1.2}$$

where $\boldsymbol{M} \in \mathbb{R}^{m\times n}$, $\sigma_1 \geqslant \sigma_2 \geqslant \cdots \geqslant \sigma_k > 0$ are the principal singular values of M, $k < \min(m, n)$. The $\boldsymbol{u}_i$ and $\boldsymbol{v}_i$ are, respectively, the left and right singular vectors of σ_i, and $(\boldsymbol{u}_i, \sigma_i, \boldsymbol{v}_i)$ is called a singular triplet of $\boldsymbol{M}$.

The PSVD is a special case of the symmetric PEVD due to the well-known fact that the SVD of $\boldsymbol{M}$ can be obtained from the EVD of any one of the following matrices:

$$\boldsymbol{M}^{\mathrm{T}}\boldsymbol{M} \in \mathbb{R}^{n\times n}, \quad \boldsymbol{M}\boldsymbol{M}^{\mathrm{T}} \in \mathbb{R}^{m\times m}, \quad \begin{bmatrix} \boldsymbol{0} & \boldsymbol{M} \\ \boldsymbol{M}^{\mathrm{T}} & \boldsymbol{0} \end{bmatrix} \in \mathbb{R}^{(m+n)\times(m+n)}. \tag{1.3}$$

Most of the algorithms for computing the PSVD of $\boldsymbol{M}$ employ the symmetric PEVD of one of the matrices in (1.3). These include the standard [18] for dense matrices and the more recent [21, 34, 48] for sparse matrices.

2 The block Davidson method for symmetric PEVD calculations

The main theme of this paper is on using filters to accelerate the block Davidson method.

We first list in Algorithm 1 the framework of the block Davidson method for the PEVD problem (1.1). Here we denote the number of wanted eigenpairs as k_{want}, the number of converged eigenpairs as k_c, the block size as k_b, the dimension of the projection subspace as k_{sub}, and the dimension of the active subspace as k_{act}. This algorithmic structure is adapted from [70] to compute the largest k_{want} eigenvalues.

Algorithm 1: Framework of a block Davidson method for symmetric PEVD calculations

```
   Input: Block size $k_b$, number of wanted eigenpairs $k_{\text{want}}$.
   Output: The converged $k_c$ eigenvectors in $V(:, 1:k_c)$ and their associated eigenvalues.
 1 Initialize parameters $k_{\text{sub}}$ and $k_{\text{act}}$; set $k_c = 0$;
 2 while $k_c \leqslant k_{\text{want}}$ do
 3     Call a specific method to construct $k_b$ augmentation vectors $V_{\text{aug}}$;
 4     Orthonormalize $V_{\text{aug}}$ against $V(:, 1:k_{\text{sub}})$ and store the resulting vectors in $V(:, k_{\text{sub}}+1 :$
       $k_{\text{sub}} + k_b')$. (Here $k_b'$ may be smaller than $k_b$);
 5     Update $k_{\text{sub}}$ and $k_{\text{act}}$: $k_{\text{sub}} = k_{\text{sub}} + k_b'$, $k_{\text{act}} = k_{\text{act}} + k_b'$;
 6     Compute $W = A * V(:, k_{\text{sub}} - k_b' + 1 : k_{\text{sub}})$ ;
 7     Compute $H(1 : k_{\text{act}}, k_{\text{act}} - k_b' + 1 : k_{\text{act}}) = W^{\mathrm{T}} * V(:, k_c + 1 : k_{\text{sub}})$, and symmetrize
       $H(1 : k_{\text{act}}, 1 : k_{\text{act}})$ ;
 8     Compute the eigendecomposition of $H$: $H(1 : k_{\text{act}}, 1 : k_{\text{act}}) = Q*D*Q^{\mathrm{T}}$, where diag($D$)
       contains non-increasing eigenvalues of $H$, and $Q$ contains their associated eigenvectors;
 9     Rotate the active subspace: $V(:, k_c + 1 : k_{\text{sub}}) = V(:, k_c + 1 : k_{\text{sub}}) * Q$;
10     Test for convergence of the Ritz vectors in $V(:, k_c + 1 : k_{\text{sub}})$. Denote the number of
       newly converged eigenpairs as $e_c$: if $e_c > 0$, update $k_c$ as $k_c = k_c + e_c$, reorder converged
       Ritz vectors if necessary;
11     Set $k_{\text{old}} = k_{\text{act}}$. Update $k_{\text{act}}$ as $k_{\text{act}} = k_{\text{act}} - e_c$;
12     if $k_{\text{act}} + k_b > k_{\text{amax}}$ then
13         do inner restart simply by setting $k_{\text{act}} = k_{\text{keep}}$, $k_{\text{sub}} = k_c + k_{\text{keep}}$;
14     Update $H$ as $H(1 : k_{\text{act}}, 1 : k_{\text{act}}) = D(e_c + 1 : e_c + k_{\text{act}}, e_c + 1 : e_c + k_{\text{act}})$;
```

The newly generated vectors in V_{aug} (see Step 3 in Algorithm 1) need to be orthonormalized against the existing projection subspace V. The resulting k_b' orthonormal vectors are then added to the projection basis to augment the active subspace V_{act}, in which a Rayleigh-Ritz refinement procedure is performed (see Steps 7–9). Note that k_b' could be smaller than k_b if some vectors in V_{aug} are numerically linearly dependent to vectors in the previous projection subspace $V_{\text{sub}} = [V_c, V_{\text{act}}]$. (One can easily add random vectors if one wishes to keep the block size to be a constant.)

This orthogonalization step is necessary for two reasons: (i) The Rayleigh-Ritz procedure needs to start with an orthonormal basis of V_{act}, therefore V_{aug} should be orthogonal to the old basis vectors in V_{act}. (ii) New eigen-directions are extracted from the active subspace V_{act} via the Rayleigh-Ritz refinement of the basis vectors in V_{act}, and the new eigen-directions should be orthogonal to the converged eigenvectors in V_c. Therefore, V_{act} needs to be orthogonal to V_c to prevent repeatedly searching for the already converged eigen-directions.

The Rayleigh-Ritz refinement generates k_{act} Ritz pairs $(\tau_i, \boldsymbol{u}_i)$, where $\boldsymbol{u}_i = V_{\text{act}}\boldsymbol{q}_i$, $i = 1, \ldots, k_{\text{act}}$. The τ_i's are the eigenvalues of H, stored in the diagonal matrix D, and the $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_{k_{\text{act}}}$ are the corresponding eigenvectors stored in Q, from Step 8. After testing for convergence (see Step 10) of these Ritz pairs, we deflate the converged Ritz vectors into V_c, and reorder the converged vectors if necessary such that the eigenvectors in $V(:,1:k_c)$ correspond to eigenvalues in non-increasing order. The non-converged Ritz vectors are left in V_{act} for the next Davidson iteration.

Step 10 of Algorithm 1 requires a convergence test on the Ritz pairs $(\tau_1, \boldsymbol{u}_1), \ldots, (\tau_{k_{\mathrm{act}}}, \boldsymbol{u}_{k_{\mathrm{act}}})$. We use the following convergence criteria for each j,

$$\|\boldsymbol{A}\boldsymbol{u}_j - \tau_j \boldsymbol{u}_j\|_2 \leqslant \mathrm{tol} * \tau_{\max}(\boldsymbol{A}). \tag{2.1}$$

Here the $\tau_{\max}(\boldsymbol{A})$ refers to the largest absolute value of the converged Ritz value, which is guaranteed to be no larger than the largest absolute value of the eigenvalues of $\boldsymbol{A}$, and tol is a user specified tolerance. A Ritz pair $(\tau_j, \boldsymbol{u}_j)$ is considered converged if (2.1) is satisfied.

Since we sort the Ritz values in a non-increasing order, $\tau_1 \geqslant \cdots \geqslant \tau_{k_{\mathrm{act}}}$, we start the convergence test from the first Ritz pair $(\tau_1, \boldsymbol{u}_1)$, and stop when the first non-converged Ritz pair is detected.

Various Davidson-type methods differ in the ways they construct the augmentation vectors V_{aug} (Step 3 in Algorithm 1). The quality, as measured by the directions of the vectors in V_{aug}, crucially determines the convergence speed of the underlying Davidson method.

To accelerate the convergence, we need to exploit filtering techniques for constructing V_{aug}.

Before we proceed to the filters, which are essential for the acceleration of the PEVD/PSVD calculations, we discuss the inner-restart technique, which is necessary to reduce the memory requirement when k_{want} is large.

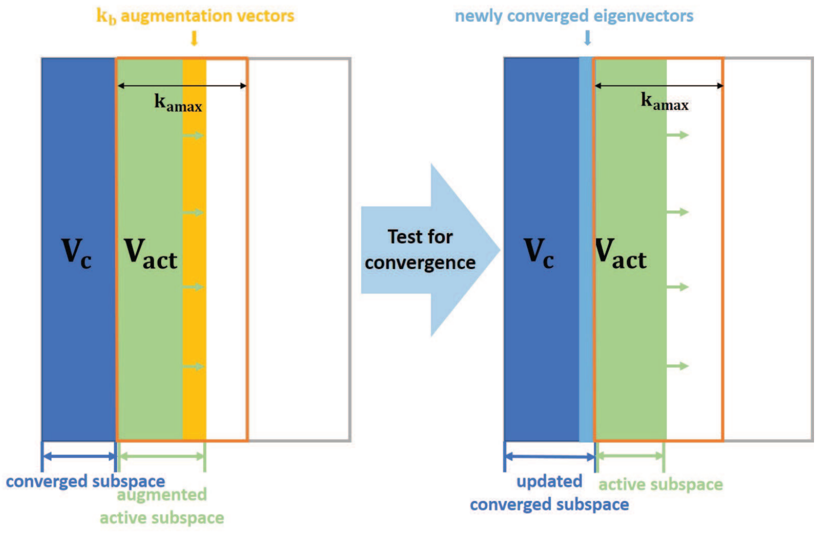
2.1 Inner-restart on the active subspace

Our inner-restart is to be distinguished from the inner-outer iteration techniques as in [16,17,20,52]. The inner iteration in these inner-outer methods refers to using an iterative method to solve a linear equation within an outer iteration, while our inner-restart refers to the repeated restart inside a smaller active projection subspace before this subspace reaches a preassigned maximum subspace dimension. With the filters we construct, this repeated restart can progressively refine the basis vectors in the projection subspace, allowing the total subspace dimension to be kept relatively small.

The inner restart resets the dimension of the active subspace k_{act} to a smaller value k_{keep} when k_{act} exceeds an upper limit k_{amax} assigned as the maximum dimension of the active subspace. The inner-restart is done by removing the least significant basis vectors in V_{act}: Assume that at a certain Davidson iteration, the dimension of V_{act} after deflation is k_{act}. If $k_{\mathrm{act}} + k_b > k_{\mathrm{amax}}$, then the inner restart is carried out by truncating the last $k_{\mathrm{act}} - k_{\mathrm{keep}}$ columns in V_{act}. When the largest k_{want} eigenvalues and eigenvectors are to be computed, during an inner restart, we truncate columns that are the Ritz vectors associated with the smallest few Ritz values in the active subspace. This is easy to do because eigenvalues of the Rayleigh quotient matrix H are ordered non-increasingly, from which we can tell which basis vectors are least significant. In other words, the inner restart keeps the best k_{keep} basis vectors that are currently available in the active subspace.

We do not need to enforce a restart on the entire projection subspace V explicitly, since the inner restart automatically sets a maximum dimension for V: Note that the number of the converged eigenvectors in V_c is k_c, and the dimension of V is k_{sub}, therefore we have $k_{\mathrm{sub}} = k_c + k_{\mathrm{act}} \leqslant k_{\mathrm{want}} + k_{\mathrm{amax}}$.

Our inner-restart is also different from the standard restart commonly used in Krylov subspace methods, which is performed, less frequently, on the entire subspace V that is always of a dimension greater than k_{want}. The standard restart corresponds to the outer restart in the inner-outer restart method [70]. Our inner-restart is performed inside an active subspace whose dimension k_{amax} is around 5–10 times the block size k_b, which can be much smaller than k_{want} when k_{want} is large. The dimension of the total iterative subspace V only needs to be $k_{\mathrm{want}} + k_{\mathrm{amax}}$, in contrast to other methods that need a projection subspace with dimension $2 * k_{\mathrm{want}}$ or even larger. This inner restart technique, enabled by filtering that generates effective V_{aug} at each iteration, makes our method highly memory-economical comparing with other established methods.



k_b augmentation vectors
k_{amax}
V_c
V_{act}
converged subspace
augmented active subspace
Test for convergence
newly converged eigenvectors
k_{amax}
V_c
V_{act}
updated converged subspace
active subspace

Figure 1 Illustration of a block Davidson iteration with inner-restart. If new eigenvectors are converged, they are deflated into the converged subspace V_c, and the active subspace $V_{\rm act}$ slides to the right. $V_{\rm act}$ is augmented by $V_{\rm aug}$ for the next iteration. This $V_{\rm act}$ is always constrained inside the $k_{\rm amax}$-wide frame—the last few columns of $V_{\rm act}$ will be truncated if the dimension of $V_{\rm act}$ exceeds $k_{\rm amax}$

Figure 1 provides a graphical view of the inner-restart in the block Davidson method.

The inner restart technique also makes our method time-efficient. Note that the complexity of major computations performed in the active subspace are as follows:

- Orthogonalization involving the non-converged basis vectors: $\mathcal{O}(nk_{\rm act}^2)$.
- Eigenvalue decomposition in the Rayleigh-Ritz refinement: $\mathcal{O}(k_{\rm act}^3)$.
- Basis rotation of the projection subspace: $\mathcal{O}(nk_{\rm act}^2)$.

Restricting the dimension of the active subspace $k_{\rm act}$ below a relatively small value can reduce the computational cost of the above three steps per iteration. As a trade-off, more Davidson outer iterations are often required to converge all the $k_{\rm want}$ eigenpairs. However, the total CPU time cost can still be greatly reduced comparing with the methods without restart or methods with only standard restart on a large projection subspace V.

2.2 The main idea of spectrum filtering for symmetric PEVD

As mentioned earlier, the convergence rate of the Davidson-type method depends on the quality of the augmentation vectors. To accelerate convergence for the PEVD problem (1.1), the augmentation vectors should be made closer to the eigenspace of $\boldsymbol{A}$ corresponding to the wanted eigenvalues. One effective way to achieve this goal is to apply a spectrum filter $f(\cdot)$ at each Davidson iteration, with the goal to introduce favorable gap-ratios among the eigenvalues of the filtered matrix $f(\boldsymbol{A})$.

The main idea of spectrum filtering can be summarized as follows: By a well-known result, the eigenvalues of $f(\boldsymbol{A})$ are $f(\lambda_i), i = 1, 2, \ldots, n$, while the eigenvectors of $f(\boldsymbol{A})$ remain the same as those of $\boldsymbol{A}$. For any initial vector $\boldsymbol{x}_0$, one can expand it as $\boldsymbol{x}_0 = \sum_{i=1}^n \alpha_i \boldsymbol{v}_i$, where the orthonormal basis $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ are the eigenvectors of $\boldsymbol{A}$. Then, $f(\boldsymbol{A})\boldsymbol{x}_0 = \sum_{i=1}^n \alpha_i f(\lambda_i)\boldsymbol{v}_i$. To obtain fast convergence, we need a filter $f(\cdot)$ such that $f(\boldsymbol{A})\boldsymbol{x}_0$ is closer to the invariant subspace spanned by the wanted eigenvectors. Since the goal of (1.1) is to extract the largest k eigenpairs $\{(\lambda_i, \boldsymbol{v}_i)\}_{i=1,\ldots,k}$, we need an $f(\cdot)$ with the property that $f(\lambda_1), \ldots, f(\lambda_k)$ are much larger than $f(\lambda_i)$, $i > k$. In other words, the filter should be designed to (i) have significantly larger values at wanted eigenvalues than at unwanted ones, and/or (ii) dampen the unwanted eigenvalues so that each iteration can progressively remove the unwanted eigen-directions from the iterative subspace.

Spectrum filtering techniques have long been used in eigen-algorithms. In fact, the power method, the Lanczos method, and all their variants (see [11, 27, 48]), are methods that apply polynomial filters.

More sophisticated methods include ARPACK [30, 60] and its variations (see [3, 9]), which apply implicit polynomials to filter out basis vectors with unwanted directions; and the Chebyshev-Davidson method [70, 74], where explicit polynomial filters are applied.

To improve performance, the filter should not be kept fixed at each iteration. Instead, it should be adaptively updated, utilizing the newer Ritz values that likely provide improved estimates of the eigenvalues of $\boldsymbol{A}$. We note that both ARPACK and the Chebyshev-Davidson method use adaptive filters based on newer Ritz values at each iteration, while ARPACK and its variants have the option of using other values such as the Leja points instead of Ritz values to adapt the filters.

In the next section, we discuss Chebyshev polynomial filters, and show why it has advantage over other polynomials of same degree. In Section 4, we focus on rational filters implemented by the conjugate residual method for solving linear equations obtained from multiple shifts.

3 The Chebyshev polynomial filtered Davidson method

The well-known Chebyshev polynomial of degree m is defined as

$$C_m(x) = \begin{cases} \cos(m\cos^{-1}(x)), & |x| \leqslant 1, \\ \cosh(m\cosh^{-1}(x)), & x > 1, \\ (-1)^m \cosh(m\cosh^{-1}(-x)), & x < -1. \end{cases} \tag{3.1}$$

There are many usages of this orthogonal polynomial, most of them focus on the best uniform approximation on the $[-1, 1]$ interval (see [2, 56]). While in eigenvalue calculations, it is the following extremum property outside the $[-1, 1]$ interval that can be used as a guiding principle for selecting the Chebyshev polynomial over other same degree polynomials.

Theorem 3.1. *For any polynomial $p(x)$ of degree $\leqslant m$ that satisfies $|p(x)| \leqslant 1$ on $[-1, 1]$, it holds true that $|p(x)| \leqslant |C_m(x)|$ for all $|x| \geqslant 1$.*

This result is proved by Chebyshev himself [10] (see also [47]). An immediate implication of Theorem 3.1 is that, for all degree m polynomials that are bounded by 1 on $[-1, 1]$, the $C_m(x)$ is the polynomial that can introduce largest gaps among polynomial values for x's that are outside $[-1, 1]$. This makes (3.1) the optimal degree m polynomial filter for symmetric eigenvalue calculations.

In fact, the Chebyshev polynomial can simultaneously achieve two goals for the acceleration of symmetric PEVD calculations: (i) Dampen any unwanted eigenvalues that are mapped inside $[-1, 1]$, and (ii) magnify the polynomial values at wanted eigenvalues, which should have been mapped outside $[-1, 1]$ before applying the polynomial filter.

Thus an important step before applying the filter (3.1) is to map unwanted eigenvalues into $[-1, 1]$ and the wanted ones out of $[-1, 1]$. But this step is far from obvious, because the location of eigenvalues may not be known apriori. Some computations may be necessary to probe the location of the eigenvalues. Usually it is not hard to estimate the exterior (i.e., smallest or largest) eigenvalues. However, the interior eigenvalue separating the wanted and the unwanted parts of the spectrum is less obvious to find. We overcome this difficulty by not insisting on using a value that correctly separate the wanted and unwanted parts of the spectrum at the beginning, instead we use a value that is much easier to obtain and then we gradually refine it throughout the iteration. We discuss the choice of the filter bounds in Subsection 3.1.

Assume that we already have an estimated interval $[b_{\text{low}}, b_{\text{up}}]$ that encloses the eigenvalues we want to dampen, then we only need to map this interval into $[-1, 1]$. This is easily achieved by a shift-and-scale mapping,

$$\mathcal{L}(x) = \frac{x - c}{e}, \quad \text{where} \quad c = \frac{b_{\text{low}} + b_{\text{up}}}{2}, \quad e = \frac{b_{\text{up}} - b_{\text{low}}}{2}. \tag{3.2}$$

The eigenvalues greater than b_{up} that we want to compute are automatically mapped outside $[-1, 1]$ by $\mathcal{L}(\cdot)$, so that they'll be magnified by the Chebyshev filter.

The Chebyshev filtering is to apply the filtered matrix $C_m(\mathcal{L}(\boldsymbol{A}))$ to a block of vectors. Algorithm 2 implements this filtering, the output vectors in Y can be used as the $V_{\rm aug}$ in Algorithm 1 (see Step 3).

Algorithm 2 computes $C_m(\mathcal{L}(\boldsymbol{A}))X$, it utilizes the well-known three-term-recurrences associated with the Chebyshev polynomial, namely $C_m(t) = 2tC_{m-1}(t) - C_{m-2}(t)$, starting from $C_0(t) = 1$, $C_1(t) = t$.

Algorithm 2: Chebyshev filter: $Y = \texttt{Cheb\_filter}(\boldsymbol{A}, X, m, b_{\rm up}, b_{\rm low})$

```
     Input: The matrix $\boldsymbol{A}$, block vectors $X$, degree $m$, bounds $b_{\rm low}$ and $b_{\rm up}$;
     Output: The filtered vectors stored in $Y$.
  1  Compute $c = (b_{\rm up} + b_{\rm low})/2$; $e = (b_{\rm up} - b_{\rm low})/2$;
  2  Compute $Y = (AX - cX)/e$;
  3  for $i = 2 : m$ do
  4      Compute $Y_{\rm tmp} = 2(AY - cY)/e - X$;
  5      $X \leftarrow Y$;
  6      $Y \leftarrow Y_{\rm tmp}$;
```

3.1 Choice of the Chebyshev filtering bounds

For the goal of computing the largest $k_{\rm want}$ eigenvalues, the $b_{\rm low}$ in (3.2) should bound all the eigenvalues from below, so that no eigenvalues can be mapped to the left of -1.

For some matrices, a theoretical lower bound for the spectrum is available. For example, eigenvalues of any symmetric positive semi-definite matrix cannot fall below 0; and for any normalized adjacency matrix, its eigenvalues cannot fall below -1.

In the more common cases where no apriori lower bounds are available, we can numerically estimate $b_{\rm low}$ with minimum computational cost, by a method adapted from [73]. The formulas in [73] used for estimating the upper bound of the total spectrum of Hermitian matrices utilize a few steps of symmetric Lanczos iteration, and then add a final safe-guard step to guarantee that a total upper bound is obtained. Here we adapt this approach to compute a total lower bound of a symmetric matrix, as listed in Algorithm 3.

Algorithm 3: k-step Lanczos iteration with a safe-guard

```
     Input: The matrix $\boldsymbol{A}$, the number of Lanczos steps $k$.
  1  Generate a random vector $v$, set $v \leftarrow v/\|v\|_2$;
  2  Compute $f = \boldsymbol{A} * v$; $\alpha = f^{\rm T}v$; $f \leftarrow f - \alpha v$; $T(1,1) = \alpha$;
  3  for $j = 2$ to $k$ do
  4      $\beta = \|f\|_2$;
  5      if $\beta$ is below machine epsilon, break;
  6      $v_0 \leftarrow v$; $v \leftarrow f/\beta$;
  7      $f = \boldsymbol{A} * v$; $f \leftarrow f - \beta v_0$;
  8      $\alpha = f^{\rm T}v$; $f \leftarrow f - \alpha v$;
  9      $T(j, j-1) = \beta$; $T(j-1, j) = \beta$; $T(j, j) = \alpha$;
 10  Return $b_{\rm low} = \lambda_{\min}(T) - \|f\|_2$, return also the eigenvalues of $T$.
```

Besides the lower bound $b_{\rm low}$, the choice of the filter upper bound $b_{\rm up}$ for the unwanted eigenvalues also plays a critical role in the performance of the filter. At the beginning of the Davidson iteration, the location of the largest unwanted eigenvalue is usually unknown. However, we do not need to fix $b_{\rm up}$ at each iteration. In fact, initially the $b_{\rm up}$ only needs to satisfy $b_{\rm low} < b_{\rm up} < \lambda_{\max}(\boldsymbol{A})$. This will map the larger eigenvalues located in $(b_{\rm up}, \lambda_{\max}(\boldsymbol{A})]$ to the right of $[-1, 1]$ so that they will be converged earlier than eigenvalues located in $[b_{\rm low}, b_{\rm up})$. The Courant-Fisher Theorem [42] guarantees that the Ritz values are all inside $[\lambda_{\min}(\boldsymbol{A}), \lambda_{\max}(\boldsymbol{A})]$, therefore we can utilize Ritz values to gradually refine $b_{\rm up}$.

In [70, 74], $b_{\rm up}$ is chosen as the median of all the non-converged Ritz values, this approach is effective in practice. Here we update $b_{\rm up}$ based on a convex combination,

$$b_{\rm up} = \alpha\tau_{\max} + (1-\alpha)\tau_{\min}, \tag{3.3}$$

where $\tau_{\min}$ and $\tau_{\max}$ are respectively the smallest and the largest non-converged Ritz values computed from the previous Davidson iteration. At the first Davidson iteration, we simply use the two extreme Ritz values stored in the eigenvalues of T from Algorithm 3 as the $\tau_{\min}$ and $\tau_{\max}$.

Numerical studies indicate that choosing $b_{\rm up}$ according to (3.3) with a proper $\alpha \in (0,1)$ can lead to even faster convergence compared with using the median. The advantage of such a convex combination were also observed in [71, 72], although in [72] the goal is to compute the smallest many eigenvalues.

4 The rational function filtered Davidson method

The Chebyshev filters are particularly effective for accelerating convergence of exterior eigenvalues. However, its effectiveness may significantly deteriorate when converging interior eigenvalues, especially so when the interior eigenvalues are clustered. This presents difficulty when many eigenvalues are needed, since in this case part of the wanted eigenvalues may be interior eigenvalues.

Instead of the Chebyshev filters, we apply the traditional rational function $\varphi(x) = \frac{1}{x-\mu}$ as the filter, where the shift μ should be placed close to some wanted interior eigenvalues. In matrix form this filter leads to the familiar shift-invert operation

$$\boldsymbol{z} = \varphi(\boldsymbol{A})\boldsymbol{x} = (\boldsymbol{A} - \mu\boldsymbol{I})^{-1}\boldsymbol{x}. \tag{4.1}$$

Applying the filter $\varphi(\cdot)$ to $\boldsymbol{A}$ can greatly magnify the eigenvalues close to μ, while dampening the others further away from μ. This shift-invert technique [19, 42] has long been used in conjunction with subspace projection methods to accelerate interior eigenvalue computations.

By the Cayley-Hamilton theorem, the inverse of a size-n nonsingular matrix is a degree $n-1$ polynomial of the matrix. Thus the rational function filter (4.1) is essentially a polynomial filter. However, when n is large, it is impractical to apply a degree $n-1$ polynomial, moreover, the coefficients of this polynomial are non-trivial to get.

A more practical way, also well-known in the literature, is to solve the linear equation

$$(\boldsymbol{A} - \mu\boldsymbol{I})\boldsymbol{z} = \boldsymbol{x} \tag{4.2}$$

approximately for the filtered vector $\boldsymbol{z}$. This is known as the *inexact* shift-invert method (see [6, 29, 58]). Iterative linear solvers are preferred over direct methods when $\boldsymbol{A}$ is large and sparse.

Integrating inexact shifted-inverse within a subspace projection method is the *inner-outer* technique we mentioned earlier. Iterative eigenvalue algorithms that apply the *inner-outer* technique include, e.g., [16, 17, 58]. The inner iteration refers to inexactly solving (4.2) by a linear solver to get a continuation vector that is required by the the outer subspace projection iteration.

Solving (4.2) by an iterative linear solver is to apply an implicit polynomial to get an approximate solution $\hat{\boldsymbol{z}}$ such that the relative residual norm is reduced to below some tolerance δ:

$$\|\boldsymbol{x} - (\boldsymbol{A} - \mu\boldsymbol{I})\hat{\boldsymbol{z}}\| / \|\boldsymbol{x}\| < \delta. \tag{4.3}$$

For a solver to be practical, the degree of the implicit polynomial should be much lower than $n-1$.

The shifted linear equations in eigenvalue algorithms are often ill-conditioned, since the shift μ should be chosen to be close to wanted eigenvalues, this will increase the gap between wanted and unwanted eigenvalues of the filtered matrix for faster convergence. The ill-conditioning of the linear equation can make iterative methods converge slowly to the true solution of the linear equation. A large number of iterations may be needed before the residual norm satisfies (4.3).

The ill-conditioning issue has attracted much research. Various preconditioners are proposed to improve conditioning in order to solve the shifted equation in fewer iterations (see [22, 58, 62]). The more common practice is that, at the beginning of the outer-iteration, the inner-iteration need not solve the shifted equation to high accuracy (see [6, 16, 20, 68]), but the focus in the existing literature is mainly on solving the shifted equation to some specified accuracy.

However, we emphasize that in eigenvector computations, obtaining the wanted direction of the solution is more essential than obtaining small residual of the linear equation (4.2). This is mainly because an eigenvector can be scaled but remains to be an eigenvector. We notice that with an unpreconditioned iterative linear solver, the approximated solution $\hat{\boldsymbol{z}}$ converges to the desired eigenvector (or invariant subspace in a block method) much faster than converging to the true solution of (4.2). The potential ill-conditioning of the shifted linear equation is actually beneficial for obtaining the correct eigen-direction [44]. Therefore, we simply apply a suitable iterative linear solver with a fixed total iteration number, without any concern of the ill-conditioning of the shifted equation. In fact, we intentionally choose shifts that make the shifted equations of form (4.2) at each outer-iteration to be ill-conditioned.

The linear solver we choose for the shift-invert filter in the block Davidson method is the conjugate residual (CR) method. In the literature, the conjugate gradient (CG) method is more commonly utilized to solve the shifted equations (see [20, 40, 41, 51]). We prefer CR over CG, mostly because the shifted equations are all indefinite when the shifts are close to interior eigenvalues, while CG is better suited for definite linear equations. Our choice of CR is inspired by [58], where CR is applied inside a Rayleigh-quotient iteration.

4.1 Fixed k-step CR method for block multi-shifted linear equations

The CR method [53] is a Krylov subspace method for solving symmetric linear equations, its extension can be used to solve non-symmetric equations. We list the pseudo code of the CR method in Algorithm 4. The name of this method comes from the fact that the residual vectors are conjugate (i.e., $\boldsymbol{H}$ orthogonal) to each other. The CR solution $\boldsymbol{x}_{j+1}$ minimizes the residual norm $\|\boldsymbol{b}-\boldsymbol{H}\tilde{\boldsymbol{x}}\|_2$ for all $\tilde{\boldsymbol{x}}$ in the Krylov space $\boldsymbol{x}_0+\mathcal{K}_j(\boldsymbol{H},\boldsymbol{b}-\boldsymbol{H}\boldsymbol{x}_0)$, where $\boldsymbol{x}_0$ is the initial vector.

Algorithm 4: Conjugate residual (CR) method for solving $\boldsymbol{Hx}=\boldsymbol{b}$

```
Input: Matrix $\boldsymbol{H}$, right-hand side $\boldsymbol{b}$, initial vector $\boldsymbol{x}_0$, and tolerance $\delta$.
Output: The approximate solution $\boldsymbol{x}$ stored in the last $x_{j+1}$.
1 $r_0 = \boldsymbol{b} - \boldsymbol{H}\boldsymbol{x}_0$; $p_0 = r_0$;
2 for $j = 0, 1, \dots$ do
3     $\alpha_j = r_j^{\mathrm{T}}\boldsymbol{H}r_j / \|\boldsymbol{H}p_j\|_2^2$;
4     $x_{j+1} = x_j + \alpha_j p_j$; $r_{j+1} = r_j - \alpha_j \boldsymbol{H}p_j$;
5     If $\|r_{j+1}\|_2 < \delta\|\boldsymbol{b}\|_2$ then break;
6     $\beta_j = r_{j+1}^{\mathrm{T}}\boldsymbol{H}r_{j+1} / r_j^{\mathrm{T}}\boldsymbol{H}r_j$; $p_{j+1} = r_{j+1} + \beta_j p_j$;
7     $\boldsymbol{H}p_{j+1} = \boldsymbol{H}r_{j+1} + \beta_j \boldsymbol{H}p_j$;
```

As standard in the Krylov methods, the matrix $\boldsymbol{H}$ only need to be accessed via matrix-vector products. To apply the CR method to the shifted linear equation (4.2), we only need to replace $\boldsymbol{Hr}_j$ in Algorithm 4 with $\boldsymbol{Ar}_j - \mu\boldsymbol{r}_j$.

The main differences of our approach with those existing in the literature are: (i) We do not concern about the ill-conditioning of the linear equation, thus we do not construct or apply any preconditioner for the shifted equations; (ii) instead of using the residual norm to determine when to stop the CR iteration, we fix the iteration number to k for each of the inner iterations; (iii) we use multiple shifts that are tailored to the outer block Davidson iteration.

Considering that constructing and applying a preconditioner is problem dependent and may be quite expensive, our preference for (i) reduces the cost associated with using preconditioners. Moreover, ill-conditioning is actually beneficial, as our goal is not to get a solution that makes the residual norm small, but to find the correct eigen-direction from each shifted equation.

Our fixed k-step CR shift-invert filter is presented in Algorithm 5. In the standard CR method as presented in Algorithm 4, the matrix vector products can be reduced by using extra arrays to store the products and reuse them when possible. This issue is addressed in Algorithm 5.

Algorithm 5: Fixed k-step CR filter: $Y = \texttt{CR\_filter}(X, k, \boldsymbol{\mu}, Y^{(0)})$

```
for $j = 1 : k_b$ do
    $y_j = y_j^{(0)}$;
    $r = x_j - Ay_j + \mu_j y_j$; $p = r$; $r_A = Ar - \mu_j r$; $p_A = r_A$;
    $\rho_s = r^{\mathrm{T}} r_A$; $\alpha = \rho_s / p_A^{\mathrm{T}} p_A$;
    $y_j = y_j + \alpha p$;
    for $i = 2 : k$ do
        $r = r - \alpha p_A$; $r_A = Ar - \mu_j r$; $\rho = r^{\mathrm{T}} r_A$; $\beta = \rho/\rho_s$; $\rho_s = \rho$;
        $p = r + \beta p$; $p_A = r_A + \beta p_A$; $\alpha = \rho_s / p_A^{\mathrm{T}} p_A$;
        $y_j = y_j + \alpha p$;
Return $Y = [y_1, \ldots, y_{k_b}]$.
```

The j in Algorithm 4 that counts the number of iterations is not necessary in implementation, because one can implement Algorithm 4 memory-economically by overwriting/updating vectors without storing a sequence of them. While in our Algorithm 5, the j is necessary, because it points to different columns in the iterative block of size k_b, as well as to the specific shift associated with each column.

The `CR_filter` in Algorithm 5 applies a fixed k-step CR iteration to approximately solve each of the k_b linear equations

$$(\boldsymbol{A} - \mu_j \boldsymbol{I})\boldsymbol{y}_j = \boldsymbol{x}_j, \quad j = 1, \ldots, k_b. \tag{4.4}$$

The input X contains the right-hand side vectors $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{k_b}]$, which are the vectors chosen to be filtered. The $\boldsymbol{\mu}$ contains the corresponding shifts $[\mu_1, \ldots, \mu_{k_b}]$ for each column in X. The $Y^{(0)}$ provides initial guesses $[\boldsymbol{y}_1^{(0)}, \ldots, \boldsymbol{y}_{k_b}^{(0)}]$ to start the CR iteration. The output Y contains the approximated solutions $[\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{k_b}]$ for each of the equations in (4.4).

When `CR_filter` is used to compute V_{aug} at Step 3 of Algorithm 1, we choose X as the k_b Ritz vectors corresponding to the largest k_b non-converged Ritz values. Note that this choice of X is the same as when the `Cheb_filter` is used.

We also use the same Ritz vectors in X as the initial guesses $Y^{(0)}$. This is because the largest k_b Ritz vectors are the best approximations in the current projection subspace to the remaining largest k_b eigenvectors to be computed. Using a better block of initial vectors can help to make the filtered solution Y evolve closer to the wanted eigenvectors.

4.2 Choice of the shifts for the shift-invert filter

At a certain Davidson iteration, we denote the largest k_b non-converged Ritz values as $\tau_1 \geqslant \cdots \geqslant \tau_{k_b}$, and the corresponding Ritz vectors as $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{k_b}$. Then as just discussed, the vectors to be filtered by `CR_filter` should be chosen as these largest $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{k_b}$. The remaining problem is to pick a shift μ_j for each $\boldsymbol{u}_j$, $j = 1, \ldots, k_b$.

The choice of the shifts $\{\mu_1, \ldots, \mu_{k_b}\}$ is crucial for a method that utilizes shift-invert. This important issue appears to have received less attention in the literature than it deserves. One reason may be that choosing shifts for a shift-invert method is challenging, similar challenging issues are well-known to exist in the low-rank Smith method [43] and the CFADI method [32] for solving large Lyapunov equations, and in the rational Krylov method [50] for model reduction of large dynamical systems.

We emphasize that ill-conditioning in the shifted equation (4.4) should be embraced instead of avoided when choosing the shifts. The residual norm of the solution of an equation in (4.4) may be very large owing to ill-conditioning, but what we need is the wanted eigen-direction out of the solution of each shifted equation. In addition, ill-conditioning of the shifted equation can help to find the wanted eigen-direction faster (see [44]).

To introduce ill-conditioning, we should choose shifts to be as close to the wanted eigenvalues as possible. Therefore we exploit the largest Ritz values $\tau_1, \ldots, \tau_{k_b}$, which are the currently available best approximations of the eigenvalues that have not been converged. We consider the following four approaches in choosing the shifts based on Ritz values,

(I) Single shift: $\mu_j = \text{median}(\tau_1, \ldots, \tau_{k_b})$ for all $j = 1, \ldots, k_b$.

(II) Single shift: $\mu_j = \beta\tau_1 + (1-\beta)\tau_{k_b}$, for all $j = 1, \ldots, k_b$. In particular, we consider using the middle point, with $\beta = 0.5$.

(III) Multiple shifts: $\mu_j = \tau_j$ for all $j = 1, \ldots, k_b$.

(IV) Mixed shifts: Let $p = \lceil \frac{1}{2} k_b \rceil$, $\mu_j = \tau_j$ for $j = 1, \ldots, p$; and $\mu_j = \tau_{p+1}$ for $j = p+1, \ldots, k_b$.

The two single shift strategies (I) and (II) are mainly used for comparisons, because a single shift method has often been preferred, particularly when a decomposition method is used to solve the shifted equation. This is because decomposition with multiple shifts can be quite expensive for large matrices.

The mixed shifts (IV) tries to compromise the possibly different needs of an interval that encloses $[\tau_{k_b}, \tau_1]$: For the more interior lower 'half' part, only the Ritz values τ_{p+1} is used as a shift; while for the more exterior higher 'half' part of this interval, for which the Ritz values are likely better approximations of the non-converged eigenvalues, each of these Ritz values τ_j $(j = 1, \ldots, p)$ is used as a shift.

The multiple shifts (III) uses the most number of shifts, this should not be a concern since we do not apply a decomposition method to solve the shifted equations.

For all the four shift strategies, the right-hand side vectors are the Ritz vectors corresponding to the current largest k_b non-converged Ritz values, and the same Ritz vectors are used as the initial vectors for the CR method.

We reason that the multiple shifts (III) have the best possibility to provide a balanced combination of good shifts and good initial vectors. This is because the Ritz pair $(\tau_j, \boldsymbol{x}_j)$ is currently available optimal approximation to the j-th non-converged eigen-pair $(\lambda_j, \boldsymbol{v}_j)$ from the current projection subspace. The good shift together with the good initial vector is expected to provide acceleration in finding the wanted eigen-direction using the CR filter in Algorithm 5.

Indeed, our extensive numerical tests show that the shift strategy (III) readily performs better than the single shift strategies (I) and (II). The mixed strategy (IV) has its merit, but for simplicity of comparison, numerical results in Section 7 are all done with the multiple shifts strategy (III).

5 Obtaining better initial vectors by Lanczos iteration

Applying the `Cheb_filter` (see Algorithm 2) or the `CR_filter` (see Algorithm 5) to compute the V_{aug} at Step 3 of Algorithm 1 leads to two filtered block Davidson algorithms for symmetric PEVD calculations.

Either algorithm can filter random k_b vectors at the first iteration to generate the first V_{aug} at Step 3 of Algorithm 1. However, we can do better than starting with random vectors.

We apply a k_{li}-step simple Lanczos iteration with full re-orthogonalization to $\boldsymbol{A}$. The iteration is essentially Algorithm 3, except that memory should be allocated to save all the k_{li} Lanczos vectors generated; and that a Rayleigh-Ritz refinement procedure should be added at the end of the Lanczos iteration to extract the k_{li} Ritz vectors. In general, the k_{li} may range from k_{amax} to $k_{\text{want}} + k_{\text{amax}}$.

Although Algorithm 3 is used to estimate the filter lower bound, while our goal is to compute the principal eigenpairs, we can still reuse the k_b largest Ritz pairs computed by Algorithm 3 (with the above mentioned additions) as initial to start the filtered block Davidson iteration.

Compared with using random initial vectors, the reuse of Lanczos vectors from the filter bound estimate as the initial vectors has the following advantages.

• Some of the largest eigenpairs may already be converged in the k_{li}-step simple Lanczos iteration for the filter bound estimate, especially when the largest eigenvalues are well-separated.

• We perform a convergence test after the Rayleigh-Ritz refinement in Algorithm 3, and deflate the already converged eigenpairs into V_c in Algorithm 1. The number of converged eigenvectors k_c is initialized accordingly.

The k_b Ritz vectors corresponding to the largest k_b non-converged Ritz values are then used as the initials to be filtered at Step 3 of Algorithm 1. These largest non-converged Ritz vectors are generally better initial vectors than random vectors, because the Ritz vectors from the Lanczos iteration are closer to the wanted invariant subspace compared with random vectors. Therefore, the main iteration (see Algorithm 1) should start with filtering the first k_b non-converged Ritz vectors.

The above choice of initial vectors also works when no eigenpair is converged in the Lanczos initialization process (i.e., $k_c = 0$).

• Based on the non-converged Ritz values, we can obtain more favorable parameters for the two filters. For example, if the `Cheb_filter` (see Algorithm 2) is used, then using a larger k_{li} than the small Lanczos step as used in [73,74] can lead to better filter bounds b_{low} and b_{up} for the first filtering step in the filtered Davidson method (see Algorithm 1).

For the k-step CR filter, obtaining better approximation of Ritz values to eigenvalues is even more important. This is because better shifts that locate close to the wanted eigenvalues, as well as better right-hand side vectors close to the wanted eigen-direction, can significantly speed up the convergence of the filtered block Davidson method.

Our extensive numerical tests show that using the Lanczos initialization within our block Davidson method is in general more efficient than initializing with random vectors. Therefore we use Lanczos iteration as the default initialization method, this default is used for all the numerical results reported in Section 7.

6 Filtered Davidson method for PSVD calculations

PSVD is a special case of symmetric PEVD. Most PSVD solvers utilize the connection between SVD and a symmetric EVD, as summarized in the following well-known lemma.

Lemma 6.1. *Let the SVD of* $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ $(m \geqslant n)$ *be*

$$\boldsymbol{M} = \boldsymbol{U} \begin{bmatrix} \operatorname{diag}(\sigma_1, \ldots, \sigma_n) \\ \mathbf{0}_{m-n} \end{bmatrix} \boldsymbol{V}^{\mathrm{T}}, \tag{6.1}$$

where $\boldsymbol{U} \in \mathbb{R}^{m\times m}$ *and* $\boldsymbol{V} \in \mathbb{R}^{n\times n}$ *are orthogonal matrices. Then the following three eigenvalue decompositions hold true,*

$$\boldsymbol{M}^{\mathrm{T}}\boldsymbol{M} = \boldsymbol{V} \operatorname{diag}(\sigma_1^2, \ldots, \sigma_n^2)\boldsymbol{V}^{\mathrm{T}}, \tag{6.2}$$

$$\boldsymbol{M}\boldsymbol{M}^{\mathrm{T}} = \boldsymbol{U} \operatorname{diag}(\sigma_1^2, \ldots, \sigma_n^2, \underbrace{0, \ldots, 0}_{m-n})\boldsymbol{U}^{\mathrm{T}}, \tag{6.3}$$

$$\begin{bmatrix} \mathbf{0} & \boldsymbol{M} \\ \boldsymbol{M}^{\mathrm{T}} & \mathbf{0} \end{bmatrix} = \boldsymbol{Q} \operatorname{diag}(\sigma_1, \ldots, \sigma_n, -\sigma_1, \ldots, -\sigma_n, \underbrace{0, \ldots, 0}_{m-n})\boldsymbol{Q}^{\mathrm{T}}, \tag{6.4}$$

where the $\boldsymbol{Q}$ *matrix is linked to* $\boldsymbol{U}$ *and* $\boldsymbol{V}$ *as*

$$\boldsymbol{Q} = \frac{1}{\sqrt{2}} \begin{bmatrix} \boldsymbol{U}_1 & \boldsymbol{U}_1 & \sqrt{2}\boldsymbol{U}_2 \\ \boldsymbol{V} & -\boldsymbol{V} & \mathbf{0} \end{bmatrix},$$

in which the $\boldsymbol{U}$ *matrix from* (6.1) *is denoted as* $\boldsymbol{U} = [\boldsymbol{U}_1 \;\; \boldsymbol{U}_2]$ *and* $\boldsymbol{U}_1 \in \mathbb{R}^{m\times n}$.

For newer applications in data science where a PSVD is applied, usually the singular triplets associated with small singular values below a certain threshold are associated with noise in the data and they have no physical meaning. Thus triplets associated with larger singular values above a threshold are of interest.

Fortunately, the squaring of $\boldsymbol{M}$, as represented in either (6.2) or (6.3), helps to magnify the larger singular values and make smaller singular values less significant. Thus, although squaring of $\boldsymbol{M}$ is known to make the smallest few singular triplets less accurate, it is actually preferred for PSVD calculations when the goal is to compute the largest part of the singular values, as for the applications from data mining. Some problems from other fields may need accurate solutions of the smallest few singular triplets, for such problems squaring should be avoided, and we refer to [33, 67] for suitable algorithms.

We compute the PSVD of a matrix $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ by computing the PEVD of the smaller-sized squaring matrix, $\boldsymbol{M}^{\mathrm{T}}\boldsymbol{M}$ (if $m > n$), or $\boldsymbol{M}\boldsymbol{M}^{\mathrm{T}}$ (if $m < n$). This is easily done by setting the matrix-vector product subroutine as $\texttt{Hx}(\boldsymbol{M}, X)$, as in Algorithm 6. The input X to Algorithm 6 refers to a block of vectors of appropriate dimension, to be multiplied by either $\boldsymbol{M}^{\mathrm{T}}\boldsymbol{M}$ or $\boldsymbol{M}\boldsymbol{M}^{\mathrm{T}}$.

Algorithm 6: Matrix-vector Product in PSVD solvers: $Y = \texttt{Hx}(\boldsymbol{M}, X)$

```
if $m \geqslant n$ then
    $Y_o = \boldsymbol{M}X$; $Y = \boldsymbol{M}^{\mathrm{T}}Y_o$;
else
    $Y_o = \boldsymbol{M}^{\mathrm{T}}X$; $Y = \boldsymbol{M}Y_o$;
```

When replacing the matrix-vector product subroutine used in a symmetric PEVD solver by the $\texttt{Hx}(\boldsymbol{M}, X)$, we turn the PEVD solver into a PSVD solver. Using the `Cheb_filter` (see Algorithm 2) as an example, it can be written as Algorithm 7. Besides replacing the matrix-vector product subroutine, we note that Algorithm 7 employs a unique property of PSVD: Since $\boldsymbol{M}^{\mathrm{T}}\boldsymbol{M}$ and $\boldsymbol{M}\boldsymbol{M}^{\mathrm{T}}$ are SPSD matrices, the lower bound b_{low} in the Chebyshev filter can be directly set to zero.

Algorithm 7: Chebyshev Filter for PSVD: $Y = \texttt{Cheb\_filter\_PSVD}(X, m, b_{\mathrm{up}})$

```
$e = b_{\mathrm{up}}/2$;
$Y = \texttt{Hx}(\boldsymbol{M}, X)/e - X$;
for $i = 2 : m$ do
    $Y_{\mathrm{tmp}} = (2/e) * \texttt{Hx}(\boldsymbol{M}, Y) - 2Y - X$;
    $X = Y$;
    $Y = Y_{\mathrm{tmp}}$;
```

Similarly, replacing the matrix-vector product subroutine by $\texttt{Hx}(\boldsymbol{M}, X)$ in the shift-invert CR filter (see Algorithm 5) also turns our PEVD solver into a PSVD solver, we omit the details here.

Once the eigenvectors $\boldsymbol{U}$ or $\boldsymbol{V}$ is computed by a PEVD solver, to compute the PSVD, we only need to take the square roots of the eigenvalues as the singular values, and then compute either $\boldsymbol{M}\boldsymbol{V}\boldsymbol{\Sigma}^{-1}$ for $\boldsymbol{U}$ if $\boldsymbol{V}$ is computed, or $\boldsymbol{M}^{\mathbf{T}}\boldsymbol{U}\boldsymbol{\Sigma}^{-1}$ for $\boldsymbol{V}$ if $\boldsymbol{U}$ is computed. Here the $\boldsymbol{\Sigma}^{-1}$ operation is nothing but the column scaling by the reciprocal of the principal singular values that are already computed.

In many applications that require computing a low-rank approximation of a given matrix $\boldsymbol{M}$, the number of needed singular triplets is unknown in advance. What is needed is that the relative approximation error being smaller than a tolerance η. To address this more common scenario, we integrate an automatic stopping criterion: Every time a new singular value σ_c is converged, we check the ratio $\frac{\sigma_c}{\sigma_1}$. If $\frac{\sigma_c}{\sigma_1} \leqslant \eta$, we terminate the iteration process. Because of the repeated refinement in the previous iterations, as well as the test of convergence criteria we use (we stop checking for convergence whenever the first non-convergence is detected in the iterative block), the larger singular values normally converge earlier than the smaller ones. Thus when $\frac{\sigma_c}{\sigma_1} \leqslant \eta$ is found, we know that the computed PSVD normally has a relative error no greater than η. In practice, to guarantee that the error is below η, one can simply use a threshold tolerance somewhat smaller than η.

Our algorithms have this desirable feature of automatically stopping the PSVD calculation whenever the approximation error is below a threshold tolerance. To our knowledge, this often desired feature has not appeared in other PSVD codes, including those that we compare with in Section 7.

7 Datasets and numerical results

We report the performance of our block filtered Davidson methods applied to realistic test problems from the LSI and the graph-based learning applications.

To gauge the relative strength of our methods, we compare with a number of major and representative methods for PEVD or PSVD calculations. The comparative study examines the cost in CPU time, the memory usage (measured mainly by the size of the major projection subspace), and the accuracy of the computed eigenpairs or singular triplets.

We first describe the matrices used in the numerical experiments, and then briefly mention the algorithms to be compared with. The main numerical results are presented in Subsection 7.3.

7.1 Datasets and test matrices

Our numerical experiments consist of solving two types of test problems: (i) computing k_{want} top singular triplets of a term-document matrix; and (ii) computing k_{want} top eigenpairs of a normalized adjacency matrix.

The test matrices are constructed from real-world datasets as needed by the LSI applications and the graph-kernel-based learning techniques

Table 1 summarizes the basic statistics of the matrices to be used in our experiments. In the matrix names, the prefix 'td_' refers to a term-document matrix whose top singular triplets are needed; and the prefix 'na_' refers to a normalized adjacency matrix whose top eigenpairs are to be computed. For a matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, its sparsity is defined as its total number of nonzero elements divided by $m * n$. In fact, this commonly used term "sparsity" refers to "density".

The datasets *Enron Emails* [1], 20 *Newsgroups* [46], *NYTimes News Articles* [1] and *PubMed Abstracts* [1] are bag-of-words files recording the number of occurrences of every word in every text. These datasets are generated from four different collections of texts, Table 2 summarizes their respective sizes and their text-content types.

We construct the term-document matrices from these datasets using the TF-IDF (term frequency-inverse document frequency) weighting, which is a widely-used weighting scheme in the LSI applications (see [23, 55]).

For the normalized adjacency matrices, we construct them from five datasets: *SIAM_competition* [14], *MNIST* [28], *epsilon* [14], *Youtube_network* [31], and *LiveJournal_network* [31]. Their statistics are summarized below:

Table 1 Statistics of the matrices used in the numerical experiments

Matrix name	# of rows	# of columns	Sparsity
`td_News20`	53,975	11,269	2.41×10^{-3}
`td_Enron`	28,102	39,861	3.31×10^{-3}
`td_NYTimes`	102,660	300,000	2.26×10^{-3}
`td_PubMed`	141,043	8,200,000	4.18×10^{-4}
`na_Siam`	21,519	21,519	8.29×10^{-4}
`na_MNIST`	70,000	70,000	2.13×10^{-4}
`na_Epsilon`	400,000	400,000	4.99×10^{-5}
`na_Youtube`	1,134,890	1,134,890	4.64×10^{-6}
`na_LiveJournal`	3,997,962	3,997,962	4.34×10^{-6}

Table 2 Properties (size and content type) of the text collection datasets used to construct term-document matrices

Dataset name	#words	#documents	Content type	Matrix name
20 Newsgroups	53,975	11,269	news article	`td_News20`
Enron Emails	28,102	39,861	email	`td_Enron`
NYTimes News Articles	102,660	300,000	news article	`td_NYTimes`
PubMed Abstracts	141,043	8,200,000	journal abstract	`td_PubMed`

• Dataset *SIAM_competition* is constructed from a collection of texts for a competition on text classification. It contains 21,519 data points, each having 30,438 features. Each data point represents a text, and its features are binary term frequencies of the words in the text. All data points are then normalized to unit length.

• Dataset *MNIST* is constructed from a collection of images of handwritten digits (integers from 0–9). It contains 70,000 data points, each having 784 features. Each data point represents an image of 28×28 pixels. This dataset is often used for testing pattern recognition methods such as learning algorithms for classification and clustering.

• Dataset *epsilon* is constructed from an artificial dataset used in the Pascal large scale learning challenge on classification. It contains 400,000 data points, each having 2,000 features.

• Datasets *Youtube_network* and *LiveJournal_network* are undirected graphs that models online social networks. The vertices represent social network users. Two vertices are linked by an undirected edge of weight 1 if the corresponding users are connected, otherwise the two vertices are not linked. *Youtube_network* has 1,134,890 vertices with 2,987,624 edges, and *LiveJournal_network* has 3,997,962 vertices with 3,4681,189 edges.

We first convert the datasets *SIAM_competition*, *MNIST* and *epsilon* into unweighted 10-nearest-neighbor (10NN) graphs according to Euclidean distances [65, 75], i.e., two distinct vertexes are connected by an edge of weight 1 if either one of them is among the 10 nearest neighbors of the other, measured in the Euclidean distance; otherwise they are not linked by an edge. Once each dataset is represented as a graph, we can construct an adjacency matrix and then apply the standard normalization to get the normalized adjacency matrix for this graph.

We add a prefix 'na_' to the matrix name to indicate that it is a normalized adjacency matrix. Graph-kernel-based learning methods require the top (largest) eigenpairs to build graph kernels.

Since the eigenvalues of a normalized adjacency matrix are all located in the $[-1,1]$ interval, when the matrix dimension is large, the eigenvalues become highly clustered. This can present severe challenges to existing eigensolvers.

7.2 Compared algorithms

We implemented the block Davidson algorithm with two filters—the Chebyshev filter and the fixed k-step conjugate residual filter—in Matlab®. The resulting solvers are denoted as `bdav`$_+$`Cheb` and `bdav`$_+$`CR`, respectively. Both can be applied for symmetric PEVD and PSVD computations.

Our `bdav`$_+$`Cheb` and `bdav`$_+$`CR` are compared with the following methods/packages, in the Matlab® environment:

• `ARPACK`, which is a Fortran package designed to solve large-scale eigenvalue problems [30]. `ARPACK` contains a set of general purpose eigensolvers that are robust, accurate and efficient, and therefore is widely accepted as a standard tool in eigenvalue computations. In Matlab®, the built-in function `eigs` essentially calls ARPACK, which applies the implicit restarted Lanczos algorithm [60, 61] for symmetric PEVD. Matlab® also provides a built-in function `svds` for PSVD computations. The difference is that `svds` computes the PSVD of $\boldsymbol{M}$ by applying `eigs` on the augmented matrix $\left[\begin{smallmatrix} \mathbf{0} & \boldsymbol{M} \\ \boldsymbol{M}^{\mathrm{T}} & \mathbf{0} \end{smallmatrix}\right]$. We believe this is not the optimal way applying `ARPACK` for PSVD computations, especially for large matrices. But for the sake of comparison with the directly available PSVD solver provided by MathWorks, we use `svds` without modification to its internal implementations.

A parameter p in `eigs` determines the number of Lanczos basis vectors spanning the projection subspace. We use the `ARPACK` default $p = 2 * k_{\text{want}}$ throughout all experiments. (Although p may be reduced from $2 * k_{\text{want}}$, using a p that is only slightly larger than k_{want} may significantly increase the CPU time or lead to only partial convergence.)

• `PROPACK`[1], which is designed to efficiently compute the PSVD of large and sparse or structured matrices. It is available in both Fortran and Matlab®. Our experiments use the Matlab® code as two other compared methods are available only in Matlab®. The PSVD solver, denoted as `LanBPRO`, is based on the Lanczos bidiagonalization algorithm with partial reorthogonalization [27]. PROPACK also provides a symmetric eigensolver, denoted as `LanPRO`, which implements the symmetric Lanczos algorithm with partial reorthogonalization [57, 66].

`LanBPRO` and `LanPRO` do not perform restart, therefore the dimension of the projection subspace is proportional to the number of iterations required for convergence. A parameter l determines the number of new basis vectors added to the subspace in every iteration. By default PROPACK sets the l to

$$1 + \left\lceil \min\left(100, \max\left(2, \frac{k_s(k_{\text{want}} - k_c)}{2(k_c + 1)}\right)\right)\right\rceil,$$

where k_c is the number of converged eigenpairs or singular triplets, and k_s is the dimension of the existing subspace. This k_s can be significantly larger than $2 * k_{\text{want}}$, especially when eigenvalues are clustered and many iterations are required, making PROPACK, which does not use restart, very memory demanding when many iterations are needed to reach convergence. Nevertheless, PROPACK is adopted in several highly cited recent publications (see [8, 35, 64]), mainly because PSVDs constitute the bottleneck of computations for the proposed algorithms there, but no significantly more robust (particularly in terms of speed and accuracy) PSVD code is available.

• `RandSVD`, which is a PSVD solver written in Matlab® based on the randomized SVD algorithm [21, 48]. It employs the main structure of a random PCA code[2], and provides options for specifying the types of initial random matrix and projection subspace. In our experiments, the initial random matrix is drawn from the standard Gaussian distribution as suggested in [21], and a block Krylov subspace iteration is used to enhance the accuracy [48].

The dimension of the random SVD projection subspace is $(1 + q)(k_{\text{want}} + p)$, determined by two parameters—the oversampling parameter p and the number of subspace iterations q. In `RandSVD` no convergence tests based on residual norms are used, instead it adopts a simpler approach: It applies the subspace iteration to $\boldsymbol{M}\boldsymbol{M}^{\mathrm{T}}$ starting from an initial $\boldsymbol{M}\boldsymbol{\Omega}$, where $\boldsymbol{\Omega}$ contains $k_{\text{want}} + p$ random vectors, and *stops after q iterations*. This approach simplifies the coding complexity but comes with a cost. As shown by the numerical results in Subsection 7.3, `RandSVD` has the least accuracy among all methods compared.

We set $p = 10$ and $q = 3$ for all experiments, as these parameters provide the overall best performance for `RandSVD` applying to the test problems listed in Subsection 7.1. Using a larger q would increase the accuracy of `RandSVD`, but with increased computational cost. As each iteration costs about the same in `RandSVD`, e.g., increasing q from 3 to 6 would roughly double the CPU time.

• `LMSVD`[3], which implements in Matlab® the limited memory PSVD algorithm [34]. The projection subspace in LMSVD has the form $[X^{(i)}, X^{(i-1)}, \ldots, X^{(i-s)}]$, where s is an integer and each $X^{(j)}$ contains k_{want} vectors. Thus the subspace dimension is $(1 + s)k_{\text{want}}$. By default, s is chosen from $\{3, 4, 5\}$ based on the relation between k_{want} and the size of the matrix. The projection subspace does not dominate the memory usage, however, `LMSVD` needs to save a large amount of intermediate results, including $\boldsymbol{A} * [X^{(i)}, \ldots, X^{(i-s)}]$, and other projections, for the purpose of reducing computational cost.

[1] Code available at http://sun.stanford.edu/∼rmunk/PROPACK/
[2] Code available at https://cims.nyu.edu/∼tygert/software.html
[3] Code available at http://www.caam.rice.edu/∼zhang/LMSVD/lmsvd.html

Table 3 Comparison on the major memory requirement of different algorithms. In the numerical tests, $q = 3$, $p = 10$, $\beta \in [0.2, 0.4]$, $s \in \{3, 4, 5\}$. The k_s is usually much larger than $2 * k_{\text{want}}$

Method	PEVD of $\boldsymbol{A} \in \mathbb{R}^{n\times n}$	PSVD of $\boldsymbol{M} \in \mathbb{R}^{m\times n}$
`bdav`$_+$`Cheb` `bdav`$_+$`CR`	$[(1+\beta)k_{\text{want}}]n$	$[(1+\beta)k_{\text{want}}]\min(m,n)$
`ARPACK(eigs/svds)`	$(2k_{\text{want}})n$	$(2k_{\text{want}})(m+n)$
`PROPACK(LanPRO/LanBPRO)`	$k_s n$	$k_s(m+n)$
`RandSVD`	(not for PEVD)	$[(1+q)(k_{\text{want}}+p)]\max(m,n)$
`LMSVD`	(not for PEVD)	$[(1+2s)k_{\text{want}}](m+n)$

We summarize the theoretical cost of the major memory usage of the compared algorithms in Table 3. For all methods except `LMSVD`, we present the size of the major projection subspace, which dominates the memory cost. As for `LMSVD`, it uses three intermediate matrices, one to save $\boldsymbol{A} * [X^{(i)}, \ldots, X^{(i-s)}]$, the other two (denoted as P_X and P_Y in [34]) are of a combined size $(m+n)s \times k_{\text{want}}$. They consume a significant amount of memory. Therefore we also count them in addition to the main projection subspace $[X^{(i)}, \ldots, X^{(i-s)}]$ of `LMSVD` when comparing the major memory usage.

There are newer codes than ARPACK and PROPACK, such as PRIMME [63] and SLEPc [49]. But there is reason why PROPACK was the dominant choice for PSVD calculations in the highly cited papers [8, 35, 64]. Therefore we believe comparisons with PROPACK as well as the well-established ARPACK provide adequate measure of the efficiency of our two methods.

In the numerical tests reported in Subsection 7.3, the real memory usage is measured by adding up the length of all vectors in the major projection subspace. That is, we add up the total floating point numbers (denoted as $\mathcal{N}$) stored in the matrix representing the major projection subspace, then convert it to the memory usage $\mathcal{M}$ (in GBs) using $\mathcal{M} = 8\mathcal{N}/1024^3$, since each floating point number takes 8 bytes (64 bits) in our computation environment.

7.3 Numerical results

Our filtered block Davidson algorithms (`bdav`$_+$`Cheb` and `bdav`$_+$`CR`) have a few parameters that can affect their efficiency. Both algorithms need the block size k_b, the maximum active subspace dimension k_{amax}, and the Lanczos initialization steps k_{li}; the algorithm using the Chebyshev filter needs the polynomial degree m; while the shift-invert CR filter needs the number of CR iterations it_{cr}.

In practice, these parameters are quite straightforward to select, and the default values in our code in general work well for many cases, testifying the robustness of our filtered algorithms.

Here we list some general guidance for the selection of parameters: (i) In general, a k_b between 5 and 30 should work well; if k_{want} is relatively large, then a larger block size k_b may be preferred. (ii) For `bdav`$_+$`Cheb`, a polynomial degree m between 4–30 should work well for most cases; when the eigenvalues are well separated (as for the term-document matrices), a small m between 4 and 8 is preferred, while for highly clustered eigenvalues (as for the normalized adjacency matrices), a larger m between 30 and 40 should be used. But a larger m, say $m > 50$, may be counter-productive, since some noise in the vectors to be filtered may be magnified into the already converged principal eigen-directions, slowing down convergence. (iii) For `bdav`$_+$`CR`, the number of CR iterations is also determined by how well the eigenvalues are separated, for well-separated cases such as the term-document matrices, it_{cr} can be set to a small integer such as 4, while for highly clustered eigenvalues, a larger it_{cr} between 30 and 40 is preferred over a smaller one. (iv) The k_{amax} is important and it depends on k_b and k_{want}, as well as on m or it_{cr}. In general, k_{amax} can simply be set as $c * k_b$, where c is an integer. When k_b is relatively small, and m or it_{cr} is small, then a larger k_{amax} is preferred, which means c should be slightly larger, say $c = 20$; otherwise, the c can be between 5–10. When k_{want} is relatively large, we can also simply choose $k_{\text{amax}} \leqslant 0.2k_{\text{want}}$.

Table 4 Default parameters in `bdav+Cheb` and `bdav+CR`

Parameter	Term-document matrix		Normalized adjacency matrix	
	`bdav+Cheb`	`bdav+CR`	`bdav+Cheb`	`bdav+CR`
k_b	15	15	15	15
k_{amax}	$0.2k_{\text{want}}$	$0.2k_{\text{want}}$	$0.2k_{\text{want}}$	$0.2k_{\text{want}}$
k_{li}	$k_{\text{want}} + k_{\text{amax}}$	$k_{\text{want}} + k_{\text{amax}}$	k_{want}	k_{want}
m	6	–	40	–
it_{cr}	–	4	–	30

The default values for our two filters used in our code are listed in Table 4.

We implement the filtered Davidson methods (`bdav+Cheb` and `bdav+CR`) in Matlab® and compare them with representative methods listed in Subsection 7.2. The two quantities compared are the CPU time and the memory usage needed by the projection subspace (see Table 3).

We separate the test matrices listed in Table 1 into two groups, based on their sizes. Group (I) consists of the `td_News20`, `td_Enron`, `na_Siam`, and `na_MNIST` matrices. Group (II) consists of the larger matrices `td_NYTimes`, `td_PubMed`, `na_Epsilon`, `na_Youtube`, and `na_LiveJournal`.

We perform more comparisons with the matrices in Group (I), since these moderately sized matrices are approachable by all methods compared within the test setting. We vary k_{want} as $\{400, 800, 1200, 1600, 2000\}$, and compare how each method performs. The point is that a method with better scaling should incur less CPU and memory cost increments as k_{want} increases—an important requirement of a practical method for large data.

For the larger matrices in Group (II), we need to fix k_{want} at a moderate value. This is because it is very time consuming for all the methods to compute a relatively large k_{want}, and the methods requiring larger memory than others would exit with an 'out-of-memory' error without finishing the computations.

All the computations were carried out in Matlab® on the SMU HPC cluster. Experiments involving moderately-sized matrices were conducted on the batch worker nodes with 8-core Intel Xeon 2.53 GHz CPU and 48 GB RAM. Tests on Group (II) matrices were performed on the high-memory nodes with 2 quad-core Intel Xeon 2.66 GHz CPU and 144 GB RAM. The Matlab® version on the batch worker nodes is R2013a (8.1.0.604) while that on the high-memory node is R2009a (7.8.0.347).

To make the measurements on CPU time more reliable, we ensure that no other programs were running on the computing node where our experiments were carried out. We also ran the same set of experiment independently for five times and report the average of the recorded CPU time.

7.3.1 Comparisons on the moderately sized matrices in Group (I)

We compare `bdav+Cheb` and `bdav+CR` with four PSVD solvers (`svds`, `LanBPRO`, `RandSVD`, `LMSVD`) and two eigensolvers (`eigs`, `LanPRO`).

The comparisons on CPU time as well as memory usage are presented in Figures 2 and 3, in which the k_{want} is increased from 400 to 2000 with a stride of 400.

Figure 2 shows the comparisons for PSVD calculations. The `svds` uses the most CPU time among all methods compared, especially when k_{want} increases. This is mainly due to the augmentation of M into $\left[\begin{smallmatrix} \mathbf{0} & \boldsymbol{M} \\ \boldsymbol{M}^{\mathrm{T}} & \mathbf{0} \end{smallmatrix}\right]$ enforced in `svds`. In the older days, this augmentation was necessary in order to compute the smallest few singular values to high accuracy. For the newer applications in data science, however, this is no longer necessary, since the smallest portion of singular values correspond to noise in the data and are safe to be neglected. A more elaborate implementation should avoid this augmentation to bring the cost of `svds` down to that of `eigs`, as shown in Figure 3. `LanBPRO` is comparable to our methods in terms of CPU time cost, especially when k_{want} is comparatively small. However, the memory usage of `LanBPRO` far exceeds our methods, due to the fact that `LanBPRO` does not use restart.

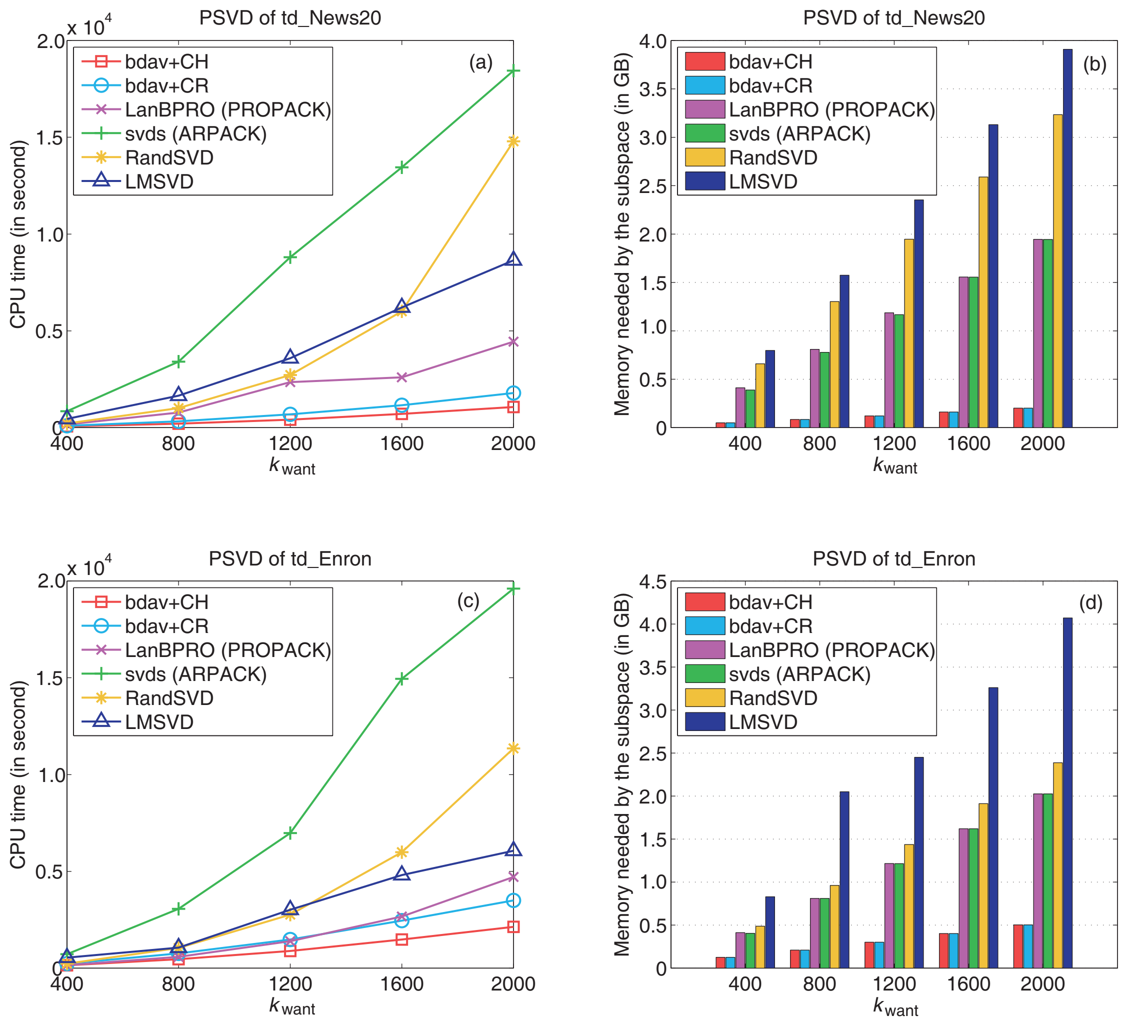


Figure 2 Comparison of six PSVD solvers on computing k_{want} singular triplets of `td_News20` (see (a) and (b)) and `td_Enron` (see (c) and (d)). Plots on the left show the CPU time cost. Plots on the right show the major memory usage of each solver

`RandSVD` is efficient when k_{want} is small, however its CPU time grows rapidly as k_{want} increases. `LMSVD` scales better than `RandSVD`, although slightly less efficient when k_{want} is small. The main concern is that `LMSVD` stores many intermediate results in exchange for speed, making the method very memory demanding.

Figure 3 shows the comparisons for PEVD calculations. The `LanPRO` uses significantly more CPU time than the other three methods, especially when k_{want} becomes larger. This is again mainly due to the fact that no restart is used in `LanPRO`, which can be very expensive in terms of both CPU time and memory consumption when many eigenpairs need to be computed. Figure 3 also reveals that, without the burden caused by the augmentation of matrices artificially enforced in `svds`, the `eigs` which calls the Fortran ARPACK is indeed efficient.

As seen from both Figures 2 and 3, the proposed `bdav`$_{+}$`Cheb` and `bdav`$_{+}$`CR` methods are faster than the other methods compared. The better performance of our methods becomes more evident when k_{want} becomes larger, implying that our methods are more scalable than the compared ones. We emphasize that the gain in speed is under the situation that our two methods use the least memory among all methods compared.

For the numerical tests reported in this subsection, the convergence tolerance `tol` is set to 10^{-6}

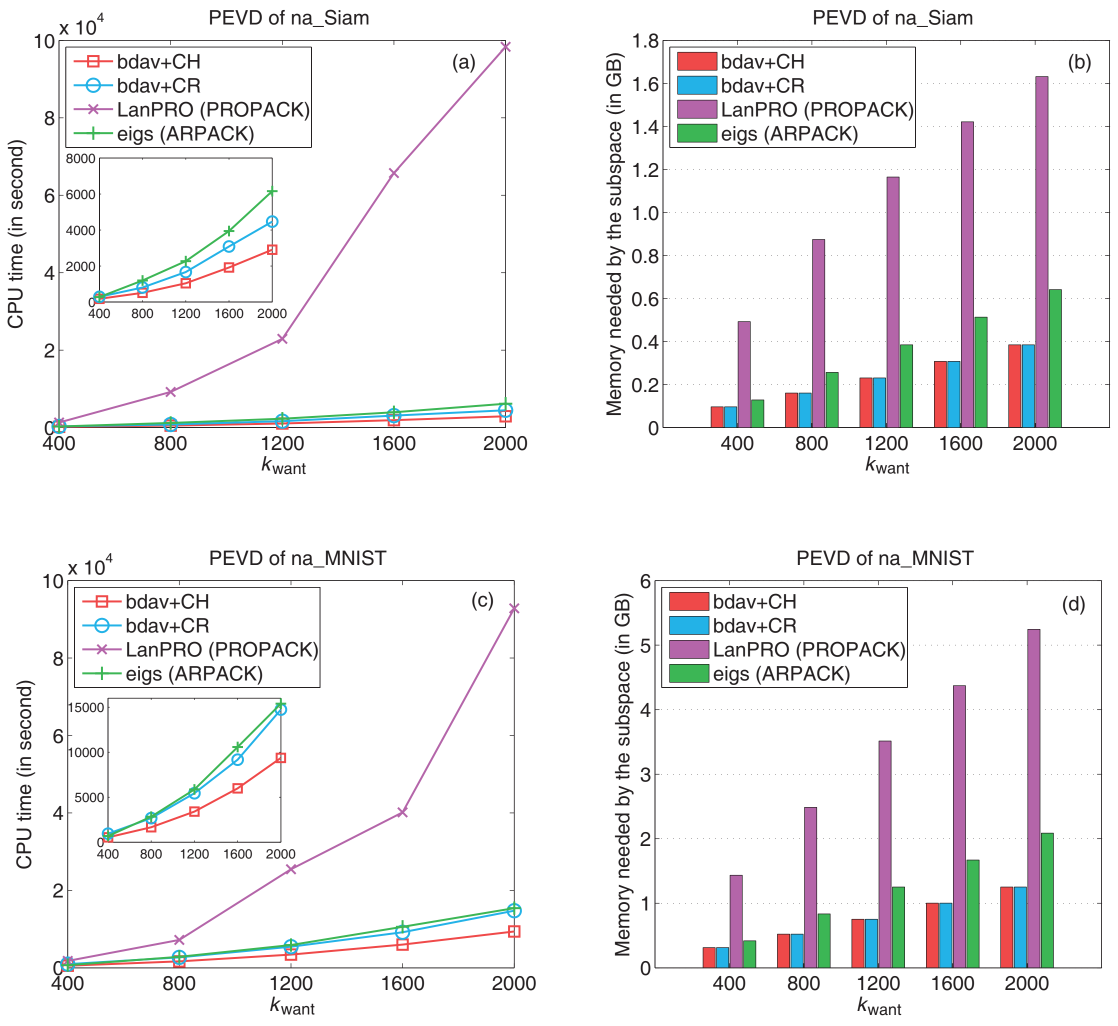


Figure 3 Comparison of four PEVD solvers on computing $k_{\rm want}$ eigenpairs of `na_Siam` (see (a) and (b)) and `na_MNIST` (see (c) and (d)). Plots on the left show the CPU time cost. The smaller windows zoom in and show only the CPU time of `bdav`$_+$`Cheb`, `bdav`$_+$`CR`, and `eigs`. The bar graphs on the right show the major memory usage of each solver

for all methods except for `RandSVD`. This is because the original `RandSVD` code [21, 48] does not test for convergence thus a `tol` is not needed. The maximum active subspace dimension $k_{\rm amax}$ for `bdav`$_+$`Cheb` and `bdav`$_+$`CR` varies with $k_{\rm want}$. We set $k_{\rm amax} = 200$ if $k_{\rm want} = 400$ or 800, and we simply set $k_{\rm amax} = 0.2k_{\rm want}$ for $k_{\rm want} > 1000$. Other parameters in `bdav`$_+$`Cheb` and `bdav`$_+$`CR` are set to the default values in Table 4. For the Chebyshev filter, the filter upper-bound $b_{\rm up}$ is chosen via the formula (3.3), with $\alpha = 0.5$ for the term-document matrices, and $\alpha = 0.9$ for the normalized adjacency matrices. Parameters for the compared methods are as specified in Subsection 7.2.

7.3.2 Comparisons on the larger matrices in Group (II)

We list in Table 5 the test problems from Group (II), as well as the parameter values used in `bdav`$_+$`Cheb` and `bdav`$_+$`CR` for each problem. Our extensive numerical tests not reported here show that different parameter values within a suitable range for our algorithms lead to mostly minor performance difference when applied to a same problem.

The CPU time profiles of the compared methods are listed in Table 6.

To avoid the 'out of memory' issue, especially for the `LanPRO` solver, we preset the maximum memory for the major projection subspace to 60 GB, and stop any method that reaches this memory limit.

Table 5 Group (II) test problems and parameter values used in `bdav+Cheb` and `bdav+CR`

Test problem	Parameters					
Matrix	k_{want}	k_b	k_{amax}	m	α in b_{up}	it$_{\text{cr}}$
`td_NYTimes`	1000	15	300	6	0.50	4
`td_PubMed`	200	6	100	6	0.50	4
`na_Epsilon`	500	15	200	40	0.95	30
`na_Youtube`	200	5	50	40	0.95	40
`na_LiveJournal`	200	5	50	50	0.95	50

Table 6 Comparison of the CPU seconds of different methods on Group (II) matrices. The '(+)' marker indicates that the `LanPRO` method in `PROPACK` converges only part of the k_{want} number of eigenpairs when the preset 60 GB memory limit is reached. The '–' marker shows the `RandSVD` solver is not applicable for PEVD calculations

	PSVD		PEVD		
Method	`td_NYTimes`	`td_PubMed`	`na_Epsilon`	`na_Youtube`	`na_LiveJournal`
`bdav+Cheb`	8675.47	7680.19	8414.17	10465.16	65996.39
`bdav+CR`	14224.77	13441.97	14212.41	9938.91	69965.49
`ARPACK`	49212.38	47666.23	22666.64	21828.75	112377.70
`PROPACK`	13867.99	14250.30	44864.12	119781.40 (+)	27962.40 (+)
`RandSVD`	12368.81	13318.31	–	–	–

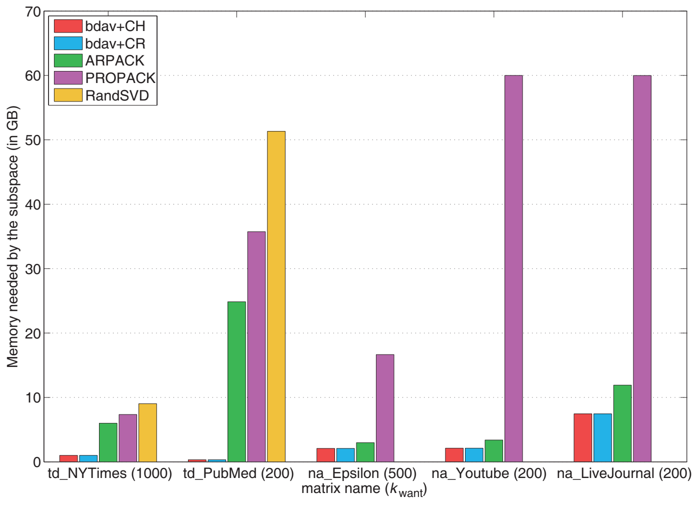


Figure 4 Comparison of the memory usage of different methods on Group (II) matrices. An upper limit of 60 GB is set for the memory used by the projection subspace. When tested on `na_Youtube` and `na_LiveJournal`, the `LanPRO` solver from `PROPACK` uses up all 60 GB memory but converges only part of the wanted eigenpairs

When this limit is reached by `LanPRO`, it converges 197 eigenpairs for the `na_Youtube` matrix, and only 9 eigenpairs for the `na_LiveJournal` matrix, out of a $k_{\text{want}} = 200$. The latter indicates that the highly clustered eigenvalues present significant difficulty for the `LanPRO` solver from `PROPACK`.

In Figure 4, we list the comparison of the memory usage.

As seen in Figure 4, `bdav+Cheb` and `bdav+CR` require the least memory among all the compared

Table 7 A detailed profile of `bdav+Cheb` and `bdav+CR` corresponding to the experimental results in Table 6. The `#iter` refers to the total number of iterations required for convergence, and the `#MVprod` refers to the number of matrix-vector products performed by the filters. The last row reports the CPU seconds used by the filters

	Method	`td_NYTimes`	`td_PubMed`	`na_Epsilon`	`na_Youtube`	`na_LiveJournal`
`#iter`	`bdav+Cheb`	70	41	67	260	293
	`bdav+CR`	111	69	110	200	221
`#MVprod` of filtering	`bdav+Cheb`	12600	2952	40200	52000	73250
	`bdav+CR`	16650	4140	51150	41000	56355
CPU seconds of filtering	`bdav+Cheb`	2601.10	4678.60	2038.20	2800.70	36229.00
	`bdav+CR`	5849.34	9717.61	4048.58	4072.70	47856.65

Table 8 Cross-validation on the accuracy of the eigenvalues or singular values: The maximum relative errors between the sets of eigenvalues or singular values computed by different methods are calculated according to (7.1). The '–' marker means the related solvers are not for PEVD calculations. (The $k_{\text{want}} = 400$ case)

Matrix	Method	`bdav+Cheb`	`bdav+CR`	`ARPACK`	`PROPACK`	`RandSVD`	`LMSVD`
`td_News20` (singular value)	`bdav+Cheb`	0	1.62E−07	1.46E−07	2.20E−07	4.14E−03	6.73E−07
	`bdav+CR`	1.62E−07	0	1.27E−07	1.84E−07	4.14E−03	6.17E−07
	`ARPACK`	1.46E−07	1.27E−07	0	2.10E−07	4.14E−03	6.51E−07
	`PROPACK`	2.20E−07	1.84E−07	2.10E−07	0	4.14E−03	6.50E−07
	`RandSVD`	4.12E−03	4.12E−03	4.12E−03	4.12E−03	0	4.12E−03
	`LMSVD`	6.73E−07	6.17E−07	6.51E−07	6.50E−07	4.14E−03	0
`td_Enron` (singular value)	`bdav+Cheb`	0	2.09E−07	2.17E−07	2.17E−07	2.12E−03	2.17E−07
	`bdav+CR`	2.09E−07	0	1.89E−07	1.89E−07	2.12E−03	1.89E−07
	`ARPACK`	2.17E−07	1.89E−07	0	3.56E−09	2.12E−03	1.37E−08
	`PROPACK`	2.17E−07	1.89E−07	3.56E−09	0	2.12E−03	1.23E−08
	`RandSVD`	2.11E−03	2.11E−03	2.11E−03	2.11E−03	0	2.11E−03
	`LMSVD`	2.17E−07	1.89E−07	1.37E−08	1.23E−08	2.12E−03	0
`na_Siam` (eigenvalue)	`bdav+Cheb`	0	3.09E−11	1.64E−11	8.50E−10	–	–
	`bdav+CR`	3.09E−11	0	2.91E−11	8.41E−10	–	–
	`ARPACK`	1.64E−11	2.91E−11	0	8.50E−10	–	–
	`PROPACK`	8.50E−10	8.41E−10	8.50E−10	0	–	–
`na_MNIST` (eigenvalue)	`bdav+Cheb`	0	7.09E−12	8.05E−12	3.52E−11	–	–
	`bdav+CR`	7.09E−12	0	6.66E−12	3.46E−11	–	–
	`ARPACK`	8.05E−12	6.66E−12	0	3.51E−11	–	–
	`PROPACK`	3.52E−11	3.46E−11	3.51E−11	0	–	–

methods. The memory size used by either `LanPRO` or `LanBPRO` grows as the computation proceeds, the total memory is unknown in advance, since it depends on the number of iterations to reach convergence. If `LanPRO` (`LanBPRO`) cannot converge all the k_{want} eigenpairs (singular triplets) when the memory used by the projection subspace exceeds the preset limit of 60 GB, the computation is terminated intentionally and the partially converged eigenpairs (singular triplets) are returned. The `LMSVD` solver is not listed in this comparison because it requires a large amount of memory throughout the iteration, which exceeds the preset 60 GB memory limit for all of the test cases in Table 5.

When used to compute the PSVDs of term-document matrices `td_NYTimes` and `td_PubMed`, `bdav+Cheb` is the fastest while `svds(ARPACK)` is the slowest, consistent with the observation on the moderately sized

Table 9 Cross-validation on the accuracy of the eigenvalues or singular values. The maximum relative errors between the sets of eigenvalues or singular values computed by different methods are calculated according to (7.1). The '–' marker means the related solvers are not for PEVD calculations. (The $k_{\text{want}} = 2000$ case)

Matrix	Method	`bdav+Cheb`	`bdav+CR`	`ARPACK`	`PROPACK`	`RandSVD`	`LMSVD`
`td_News20` (singular value)	`bdav+Cheb`	0	8.08E−06	7.53E−06	7.53E−06	1.13E−04	7.00E−05
	`bdav+CR`	8.08E−06	0	7.55E−06	7.55E−06	1.11E−04	6.75E−05
	`ARPACK`	7.53E−06	7.55E−06	0	9.72E−14	1.12E−04	6.91E−05
	`PROPACK`	7.53E−06	7.55E−06	9.72E−14	0	1.12E−04	6.91E−05
	`RandSVD`	1.13E−04	1.11E−04	1.12E−04	1.12E−04	0	9.05E−05
	`LMSVD`	7.00E−05	6.75E−05	6.91E−05	6.91E−05	9.05E−05	0
`td_Enron` (singular value)	`bdav+Cheb`	0	2.11E−05	1.21E−05	1.21E−05	1.11E−04	2.82E−04
	`bdav+CR`	2.11E−05	0	1.90E−05	1.90E−05	1.11E−04	2.86E−04
	`ARPACK`	1.21E−05	1.90E−05	0	1.10E−13	1.14E−04	2.85E−04
	`PROPACK`	1.21E−05	1.90E−05	1.10E−13	0	1.14E−04	2.85E−04
	`RandSVD`	1.11E−04	1.11E−04	1.14E−04	1.14E−04	0	1.81E−04
	`LMSVD`	2.82E−04	2.86E−04	2.85E−04	2.85E−04	1.81E−04	0
`na_Siam` (eigenvalue)	`bdav+Cheb`	0	8.47E−11	9.87E−11	5.11E−10	–	–
	`bdav+CR`	8.47E−11	0	7.31E−11	5.01E−10	–	–
	`ARPACK`	9.87E−11	7.31E−11	0	5.10E−10	–	–
	`PROPACK`	5.11E−10	5.01E−10	5.10E−10	0	–	–
`na_MNIST` (eigenvalue)	`bdav+Cheb`	0	2.21E−11	2.14E−11	4.53E−10	–	–
	`bdav+CR`	2.21E−11	0	2.29E−11	4.48E−10	–	–
	`ARPACK`	2.14E−11	2.29E−11	0	4.54E−10	–	–
	`PROPACK`	4.53E−10	4.48E−10	4.54E−10	0	–	–

matrices as in Figure 2 (for the same reason of the enforced augmentation of a matrix in `svds`, which should actually be avoided for the PSVD calculation of large matrices).

The CPU time of `bdav+CR`, `LanBPRO(PROPACK)` and `RandSVD` are comparable, but `bdav+CR` provides more accurate results, as seen in Table 10 in the next subsection.

Moreover, our `bdav+CR` method requires much less memory as illustrated in Figure 4. When used to solve the PEVD of normalized adjacency matrices `na_Epsilon`, `na_Youtube`, and `na_LiveJournal`, our methods `bdav+Cheb` and `bdav+CR` outperform `eigs` (`ARPACK`) and `LanPRO` (`PROPACK`) in terms of CPU time cost as well as memory usage, especially for the larger matrices `na_Youtube` and `na_LiveJournal`.

We also report a detailed comparison between `bdav+Cheb` and `bdav+CR` in Table 7. The results in Table 7 indicate that when tested on `td_NYTimes`, `td_PubMed` and `na_Epsilon`, the `bdav+Cheb` method clearly outperforms `bdav+CR` in terms of CPU time and number of iterations. However, when compared on `na_Youtube` and `na_LiveJournal`, the number of iterations taken by `bdav+CR` to convergence is approximately 3/4 of that needed by `bdav+Cheb`.

The performance difference between our two methods for these tests can be explained by the eigenvalue distributions of the test matrices: The eigenvalues of `na_Youtube` and `na_LiveJournal` are more clustered than the eigenvalues of the other test matrices. While the Chebyshev polynomial filter can enlarge the gaps among the exterior eigenvalues efficiently (see Theorem 3.1), its power in introducing favorable gaps for the interior eigenvalues is limited, especially when the interior eigenvalues are clustered. The shift-invert filter, however, is better suited for magnifying the gaps among interior and clustered eigenvalues when proper shifts are used.

7.3.3 Cross validation on the accuracy of the solvers

Besides the CPU time and memory consumption, an important measure of a solver is the accuracy of the solutions it produces.

Table 10 Cross-validation on the accuracy of the eigenvalues or singular values. The maximum relative errors between the sets of eigenvalues or singular values computed by different methods are calculated according to (7.1). The '–' marker means `RandSVD` is not for PEVD calculations, and the '`x`' marker indicates `PROPACK` does not get full convergence thus not suitable for cross-validation

matrix	method	`bdav+Cheb`	`bdav+CR`	`ARPACK`	`PROPACK`	`RandSVD`
`td_NYTimes` (singular value)	`bdav+Cheb`	0	1.41E−08	1.52E−08	4.52E−07	1.42E−02
	`bdav+CR`	1.41E−08	0	1.36E−08	4.50E−07	1.42E−02
	`ARPACK`	1.52E−08	1.36E−08	0	4.52E−07	1.42E−02
	`PROPACK`	4.52E−07	4.50E−07	4.52E−07	0	1.42E−02
	`RandSVD`	1.40E−02	1.40E−02	1.40E−02	1.40E−02	0
`td_PubMed` (singular value)	`bdav+Cheb`	0	6.58E−10	9.94E−10	1.61E−06	2.95E−02
	`bdav+CR`	6.58E−10	0	5.75E−10	1.61E−06	2.95E−02
	`ARPACK`	9.94E−10	5.75E−10	0	1.61E−06	2.95E−02
	`PROPACK`	1.61E−06	1.61E−06	1.61E−06	0	2.95E−02
	`RandSVD`	2.86E−02	2.86E−02	2.86E−02	2.86E−02	0
`na_Epsilon` (eigenvalue)	`bdav+Cheb`	0	4.43E−11	2.76E−11	5.84E−10	–
	`bdav+CR`	4.43E−11	0	4.62E−11	5.69E−10	–
	`ARPACK`	2.76E−11	4.62E−11	0	5.81E−10	–
	`PROPACK`	5.84E−10	5.69E−10	5.81E−10	0	–
`na_Youtube` (eigenvalue)	`bdav+Cheb`	0	2.25E−10	4.44E−11	`x`	–
	`bdav+CR`	2.25E−10	0	2.24E−10	`x`	–
	`ARPACK`	4.44E−11	2.24E−10	0	`x`	–
`na_LiveJournal` (eigenvalue)	`bdav+Cheb`	0	3.87E−10	9.44E−11	`x`	–
	`bdav+CR`	3.87E−10	0	3.93E−10	`x`	–
	`ARPACK`	9.44E−11	3.93E−10	0	`x`	–

To examine the accuracy of the compared methods, we cross-validate the numerical results obtained by the six methods for each of the PEVD or PSVD test problem, obtaining 6 sets of pairs (eigen-pairs, or SVD pairs using a left/right singular vector),

$$\{(\lambda_i^{(j)}, \boldsymbol{v}_i^{(j)})\}_{i=1}^{k_{\text{want}}}, \quad j = 1, \ldots, 6.$$

Then we use each set of pairs as the 'benchmark' to compute the cross validation errors. The maximum relative error between $\{\lambda_i^{(p)}\}_{i=1}^{k_{\text{want}}}$ computed by the p-th method and those computed by other methods is defined as

$$E_j^{(p)} = \max_{1\leqslant i\leqslant k_{\text{want}}} \frac{|\lambda_i^{(p)} - \lambda_i^{(j)}|}{|\lambda_i^{(p)}|}, \quad j = 1, \ldots, 6. \tag{7.1}$$

For the purpose of cross-validating the eigenvalues computed by different methods, we calculate $E_j^{(p)}$ for all $j = 1, \ldots, 6$.

To cross-validate the computed eigenvectors, we calculate the subspace angle θ_{pq} between any two eigenspaces with orthonormal basis $\boldsymbol{V}_p$ and $\boldsymbol{V}_q$ using the Matlab® built-in function `subspace()`. This function returns the value $\sin(\theta_{pq}) = (\boldsymbol{I} - \boldsymbol{V}_p\boldsymbol{V}_p^{\mathrm{T}})\boldsymbol{V}_q$.

For brevity of the presentation, we only report the cross validated errors of the computed eigenvalues and singular values. For the moderately sized Group (I) matrices, we only report the $k_{\text{want}} = 400$ and $k_{\text{want}} = 2000$ cases, in Tables 8 and 9, respectively. For the larger Group (II) matrices, we report the maximum cross validated errors in Table 10.

In all these tables, the results obtained by the method, whose name is listed on top of each data column, serve as the 'benchmark' for comparison with results by the other methods. The maximum relative errors is computed according to (7.1).

Cross validation errors for the other values of k_{want} are similar to the ones reported here and thus omitted for the sake of brevity.

As seen from Tables 8–10, the results obtained by `RandSVD` are most different from any of the other methods, implying that `RandSVD` is less accurate comparing with other methods. This is mainly due to the fact that `RandSVD` does not use a tolerance to measure convergence. We mention that the accuracy of `RandSVD` may be improved to be comparable with the other methods by increasing the number of iterations, although this will incur increased CPU time cost for `RandSVD`.

As for the other methods, their accuracy are comparable with the ARPACK solver. Since ARPACK is generally recognized as providing industrial standard accuracy, we conclude that $\texttt{bdav}_{+}\texttt{Cheb}$ and $\texttt{bdav}_{+}\texttt{CR}$, as well as PROPACK and `LMSVD` (when applicable), all can provide accuracy within a specified tolerance for PEVD/PSVD calculations.

8 Concluding remarks

We develop two algorithms based on filter-accelerated block Davidson method for large PSVD and symmetric PEVD calculations. One algorithm applies Chebyshev polynomial filtering, which utilizes the fastest growth of the Chebyshev polynomial among same degree polynomials that are bounded by 1 on $[-1, 1]$. The other algorithm applies rational-function filtering by solving linear equations. Since the shifted linear equation is solved by a finite-step conjugate residual (CR) method, essentially the rational-function filter is approximated by a polynomial filter. We numerically compare four types of automatic choices of shifts and find that using multiple shifts is better than using a single shift in our block Davidson method. The two filters usually generate high-quality basis vectors to augment the projection subspace at each Davidson iteration step. This allows a restart scheme using a projection subspace of small dimension. This feature makes our algorithms memory-economical, thus practical for large PEVD/PSVD calculations, especially when a large number of eigenpairs or singular triplets are needed.

We compare our filtered Davidson methods with representative algorithms, including the well-established ARPACK [30,60], the frequently utilized PROPACK [27] in data science, and two recent methods—the randomized SVD method [21, 48] and the limited memory SVD method [34]. Numerical tests on representative datasets demonstrate the advantage of our methods in terms of CPU time cost and memory usage. In general, our methods have similar or faster convergence speed in terms of CPU time, while requiring much lower memory comparing with other algorithms. The comparatively much lower memory requirement makes our methods more practical for large-scale PEVD/PSVD computations.

A natural extension of our methods is to combine Chebyshev filtering with the rational filtering, using the former for exterior eigenvalues, and then switching to the latter when Chebyshev filtering becomes less effective for the eigenvalues located in the more interior part of the spectrum. This provides a promising way to combine the strength of both filters and apply them to where they work best. However, designing an effective way to switch automatically from Chebyshev to the rational filtering appears to be not straightforward and consists part of our further studies.

Acknowledgements This work was supported by National Science Foundation of USA (Grant Nos. DMS-1228271 and DMS-1522587), National Natural Science Foundation of China for Creative Research Groups (Grant No. 11321061), the National Basic Research Program of China (Grant No. 2011CB309703), and the National Center for Mathematics and Interdisciplinary Sciences, Chinese Academy of Sciences. The authors thank the anonymous referees for constructive comments.

References

1 Bache K, Lichman M. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2013

2 Baer R, Head-Gordon M. Chebyshev expansion methods for electronic structure calculations on large molecular systems. J Chem Phys, 1997, 107: 10003–10013

3 Baglama J, Calvetti D, Reichel L. IRBL: An implicitly restarted block-Lanczos method for large-scale Hermitian eigenproblems. SIAM J Sci Comput, 2003, 24: 1650–1677

4 Bai Z, Demmel J, Dongarra J, et al. Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. Philadelphia: SIAM, 2000

5 Belabbas M-A, Wolfe P J. Spectral methods in machine learning and new strategies for very large datasets. Proc Natl Acad Sci USA, 2009, 106: 369–374

6 Berns-Müller J, Graham I G, Spence A. Inexact inverse iteration for symmetric matrices. Linear Algebra Appl, 2006, 416: 389–413

7 Cai D, He X, Han J. Spectral regression: A unified subspace learning framework for content-based image retrieval. In: Proceedings of the 15th International Conference on Multimedia. New York: ACM, 2007, 403–412

8 Cai J F, Candès E J, Shen Z. A singular value thresholding algorithm for matrix completion. SIAM J Optim, 2010, 20: 1956–1982

9 Calvetti D, Reichel L, Sorensen D C. An implicit restarted Lanczos method for large symmetric eigenvalue problem. Elec Trans Numer Anal, 1994, 1: 237–263

10 Chebyshev P L. Sur les fonctions qui s'écartent peu de zéro pour certaines valeurs de la variable. Oeuvreas, 1881, 2: 335–356

11 Chen J, Saad Y. Lanczos vectors versus singular vectors for effective dimension reduction. IEEE Trans Knowl Data Eng, 2009, 21: 1091–1103

12 Davidson E R. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. J Comput Phys, 1975, 17: 87–94

13 Deerwester S, Dumais S T, Furnas G W, et al. Indexing by latent semantic analysis. J Amer Soc Inform Sci, 1990, 41: 391–407

14 Fan R-E, Lin C-J. Libsvm data: Classification, regression, and multi-label. http://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/

15 Fokkema D R, Sleijpen G L G, van der Vorst H A. Jacobi-Davidson style QR and QZ algorithms for the reduction of matrix pencils. SIAM J Sci Comput, 1998, 20: 94–125

16 Freitag M A. Inner-outer Iterative Methods for Eigenvalue Problems—Convergence and Preconditioning. PhD Thesis. Bath: University of Bath, 2007

17 Gleich D F, Gray A P, Greif C, et al. An inner-outer iteration for computing PageRank. SIAM J Sci Comput, 2010, 32: 349–371

18 Golub G H, Kahan W. Calculating the singular values and pseudo-inverse of a matrix. SIAM J Numer Anal, 1965, 2: 205–224

19 Golub G H, Van Loan C F. Matrix Computations. Baltimore: The Johns Hopkins University Press, 1996

20 Golub G H, Ye Q. Inexact preconditioned conjugate gradient method with inner-outer iteration. SIAM J Sci Comput, 1999, 21: 1305–1320

21 Halko N, Martinsson P-G, Tropp J A. Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions. SIAM Rev, 2009, 53: 217–288

22 Knyazev A V. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. SIAM J Sci Comput, 2001, 23: 517–541

23 Ko Y. A study of term weighting schemes using class information for text classification. In: SIGIR' 12 Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval. New York: ACM, 2012, 1029–1030

24 Kohanoff J. Electronic Structure Calculations for Solids and Molecules: Theory and Computational Methods. Cambridge: Cambridge University Press, 2006

25 Kunegis J, Lommatzsch A. Learning spectral graph transformations for link prediction. In: Proceedings of the 26th International Conference on Machine Learning. New York: ACM, 2009, 561–568

26 Lanczos C. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. J Res Nat Bur Standards, 1950, 45: 255–282

27 Larsen R M. Lanczos bidiagonalization with partial reorthogonalization. Technical Report DAIMI PB-357. Aarhus: Aarhus University, 1998

28 LeCun Y, Cortes C, Burges C J C. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/

29 Lehoucq R B, Meerbergen K. Using generalized Cayley transformations within an inexact rational Krylov sequence method. SIAM J Matrix Anal Appl, 1998, 20: 131–148

30 Lehoucq R B, Sorensen D C, Yang C. ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods. Phildelphia: SIAM, 1998

31 Leskovec J, Krevl A. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014

32 Li J, White J. Low rank solution of Lyapunov equations. SIAM J Matrix Anal Appl, 2002, 24: 260–280

33 Liang Q, Ye Q. Computing singular values of large matrices with inverse free preconditioned krylov subspace method.

Electron Trans Numer Anal, 2014, 42: 197–221
34 Liu X, Wen Z, Zhang Y. Limited memory block Krylov subspace optimization for computing dominant singular value decompositions. SIAM J Sci Comput, 2013, 35: A1641–A1668
35 Ma S, Goldfarb D, Chen L. Fixed point and Bregman iterative methods for matrix rank minimization. Math Program, 2011, 128: 321–353
36 Martin R M. Electronic Structure: Basic Theory and Practical Methods. Cambridge: Cambridge University Press, 2004
37 Mazumder R, Hastie T, Tibshirani R. Spectral regularization algorithms for learning large incomplete matrices. J Mach Learn Res, 2010, 11: 2287–2322
38 Morgan R B, Scott D S. Generalizations of Davidson's method for computing eigenvalues of sparse symmetric matrices. SIAM J Sci Stat Comput, 1986, 7: 817–825
39 Ng A Y, Jordan M I, Weiss Y. On spectral clustering: Analysis and an algorithm. Adv Neural Inf Process Syst, 2002, 14: 849–856
40 Notay Y. Combination of Jacobi-Davidson and conjugate gradients for the partial symmetric eigenproblem. Numer Linear Algebra Appl, 2002, 9: 21–44
41 Ovtchinnikov E E. Jacobi correction equation, line search, and conjugate gradients in hermitian eigenvalue computation II: Computing several extreme eigenvalues. SIAM J Numer Anal, 2008, 46: 2593–2619
42 Parlett B N. The Symmetric Eigenvalue Problem. Philadelphia: SIAM, 1998
43 Penzl T. A cyclic low-rank Smith method for large sparse Lyapunov equations. SIAM J Sci Comput, 2000, 21: 1401–1418
44 Peters G, Wilkinson J H. Inverse iteration, ill-conditioned equations, and Newton's method. SIAM Rev, 1979, 21: 339–360
45 Prateek J, Meka R, Dhillon I S. Guaranteed rank minimization via singular value projection. Adv Neural Inf Process Syst, 2010, 23: 937–945
46 Rennie J. 20 newsgroups. http://qwone.com/∼jason/20Newsgroups/, 2008
47 Rivlin T J. Chebyshev Polynomials. New York: John Wiley & Sons, 1974
48 Rokhlin V, Szlam A, Tygert M. A randomized algorithm for principal component analysis. SIAM J Matrix Anal Appl, 2009, 31: 1100–1124
49 Roman J E, Campos C, Romero E, et al. SLEPc users manual. Technical Report DSIC-II/24/02-Revision 3.6. València: Universitat Politècnica de València, 2015
50 Ruhe A, Skoogh D. Rational Krylov algorithm for eigenvalue computation and model reduction. In: Proceedings of the 4th International Workshop on Applied Parallel Computing. Large Scale Scientific and Industrial Problems. New York: Springer-Verlag, 1998: 49150-2
51 Ruhe A, Wiberg T. The method of conjugate gradients used in inverse iteration. BIT, 1972, 12: 543–554
52 Saad Y. A flexible inner-outer preconditioned GMRES algorithm. SIAM J Sci Comput, 1993, 14: 461–469
53 Saad Y. Iterative Methods for Sparse Linear Systems. Philadelphia: SIAM, 2003
54 Saad Y. Numerical Methods for Large Eigenvalue Problems. Philadelphia: SIAM, 2011
55 Salton G, Buckley C. Term-weighting approaches in automatic text retrieval. Inform Process Manag, 1988, 24: 513–523
56 Sankey O F, Drabold D A, Gibson A. Projected random vectors and the recursion method in the electronic-structure problem. Phys Rev B, 1994, 50: 1376–1381
57 Simon H D. The Lanczos algorithm with partial reorthogonalization. Math Comput, 1984, 42: 115–142
58 Simoncini V, Eldén L. Inexact Rayleigh quotient-type methods for eigenvalue computations. BIT, 2002, 42: 159–182
59 Sleijpen G L G, van der Vorst H A. A Jacobi-Davidson iteration method for linear eigenvalue problems. SIAM J Matrix Anal Appl, 1996, 17: 401–425
60 Sorensen D C. Implicit application of polynomial filters in a k-step Arnoldi method. SIAM J Matrix Anal Appl, 1992, 13: 357–385
61 Sorensen D C. Implicitly restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations. Technical Report TR-96-40. Houston: Rice University, 1996
62 Stathopoulos A, McCombs J R. Nearly optimal preconditioned methods for hermitian eigenproblems under limited memory, part II: Seeking many eigenvalues. SIAM J Sci Comput, 2007, 29: 2162–2188
63 Stathopoulos A, McCombs J R. PRIMME: PReconditioned Iterative MultiMethod Eigensolver: Methods and software description. ACM Trans Math Software, 2010, 37: 21–30
64 Toh K C, Yun S. An accelerated proximal gradient algorithm for nuclear norm regularized linear least squares problems. Pac J Optim, 2010, 6: 615–640
65 von Luxburg U. A tutorial on spectral clustering. Stat Comput, 2007, 17: 395–416
66 Wu K, Simon H D. A parallel Lanczos method for symmetric generalized eigenvalue problems. Report 41284. Lawrence: Lawrence Berkeley National Laboratory, 1997
67 Wu L, Stathopoulos A. PRIMME SVDS: A preconditioned SVD solver for computing accurately singular triplets of

large matrices based on the PRIMME eigensolver. ArXiv:1408.5535, 2014

68 Xue F, Elman H C. Fast inexact subspace iteration for generalized eigenvalue problems with spectral transformation. Linear Algebra Appl, 2011, 435: 601–622

69 Zhou Y. Studies on Jacobi-Davidson, rayleigh quotient iteration, inverse iteration generalized Davidson and Newton updates. Numer Linear Algebra Appl, 2006, 13: 621–642

70 Zhou Y. A block Chebyshev-Davidson method with inner-outer restart for large eigenvalue problems. J Comput Phys, 2010, 229: 9188–9200

71 Zhou Y. Practical acceleration for computing the HITS ExpertRank vectors. J Comput Appl Math, 2012, 236: 4398–4409

72 Zhou Y, Chelikowsky J R, Saad Y. Chebyshev-filtered subspace iteration method free of sparse diagonalization for solving the Kohn-Sham equation. J Comput Phys, 2014, 274: 770–782

73 Zhou Y, Li R-C. Bounding the spectrum of large Hermitian matrices. Linear Algebra Appl, 2011, 435: 480–493

74 Zhou Y, Saad Y. A Chebyshev-Davidson algorithm for large symmetric eigenvalue problems. SIAM J Matrix Anal Appl, 2007, 29: 954–971

75 Zhu X, Kandola J, Lafferty J, et al. Graph kernels by spectral transforms. In: Chapelle O, Scholkopf B, Zien A, eds. Semi-Supervised Learning. Cambridge: MIT Press, 2006, 277–291