Numerical Algorithms
https://doi.org/10.1007/s11075-024-01906-9

RESEARCH

A hybrid algorithm for computing a partial singular value decomposition satisfying a given threshold

James Baglama[1] · Jonathan A. Chávez-Casillas[1] · Vasilije Perović[1]

Received: 8 July 2024 / Accepted: 25 July 2024

Abstract
In this paper, we describe a new hybrid algorithm for computing all singular triplets above a given threshold and provide its implementation in MATLAB/Octave and R. The high performance of our codes and ease at which they can be used, either independently or within a larger numerical scheme, are illustrated through several numerical examples with applications to matrix completion and image compression. Well-documented MATLAB and R codes are provided for public use.

Keywords Partial singular value decomposition · `svds` · MATLAB · R · Matrix completion · Singular value thresholding

1 Introduction

A fundamental tool arising in diverse areas of applications, e.g., matrix completion [1–5], imaging [6–9], principal component analysis [10–12], machine learning [13], and genomics [14, 15], is the computation of a *partial singular value decomposition (PSVD)* of a large sparse matrix $A \in \mathbb{R}^{m\times n}$,

$$A\,V_k \;=\; U_k\,\Sigma_k \;+\; \mathcal{E}_1, \qquad A^T U_k \;=\; V_k\,\Sigma_k, + \mathcal{E}_2, \qquad A_k \;=\; U_k\,\Sigma_k V_k^T, \tag{1}$$

where $\Sigma_k = \mathrm{diag}(\sigma_1, \ldots, \sigma_k) \in \mathbb{R}^{k\times k}$ with singular values $\sigma_1 \geq \ldots \geq \sigma_k > 0$, $U_k \in \mathbb{R}^{m\times k}$, $V_k \in \mathbb{R}^{n\times k}$ have orthonormal columns of associated singular vectors, A_k

James Baglama, Jonathan A. Chávez-Casillas, and Vasilije Perović contributed equally to this work.

✉ Vasilije Perović
perovic@uri.edu

James Baglama
jbaglama@uri.edu

Jonathan A. Chávez-Casillas
jchavezc@uri.edu

[1] Department of Mathematics and Applied Mathematical Sciences, University of Rhode Island, Kingston, RI 02881, USA

Published online: 06 August 2024

is a rank k approximation of A, and $\mathcal{E}_1 \in \mathbb{R}^{m\times k}$, $\mathcal{E}_2 \in \mathbb{R}^{n\times k}$ are approximation errors. A singular value with its associated left and right singular vectors is referred to as a *singular triplet*. In this paper we investigate various computational aspects of determining *all* singular triplets corresponding to singular values of A above a user-specified threshold parameter *sigma*, or in other words, determining a k–PSVD of A such that $\sigma_k \geq sigma$ and $\sigma_{k+1} < sigma$. When A is of modest size, a k-PSVD of A can be obtained by truncating its full SVD which ultimately determines *all* singular triplets of A (see [16]). However, we do not consider this approach since the full SVD is designed for dense matrices and is not scalable to large sparse matrices, becoming computationally and memory intensive. Furthermore, we may assume A and A^T are only accessible via matrix–vector product routines which can be optimized for respective sparsity structure.

Over the last few decades, various numerical schemes with publicly available software have been developed for finding an *approximate k–PSVD* of a large sparse matrix A, e.g., PROPACK [17], `irlba` [18, 19], `primme_svds` [20], `svdifp` [21], `rsvd` [7], `svds-C` [22], RSpectra [23], `dashSVD` [24], Octave's `svds`[25], and commercially available MATLAB's `svds` [26]. A shared feature by all these methods is that they all require knowing the number of desired singular triplets, k, in advance, thus significantly limiting their use in applications where k is not readily available. For example, in the now classical work on matrix completion by Cai, Candès, and Shen [1], the authors developed a method for the problem of recovering an approximately low-rank matrix from a sampling of its entries. The main computational kernel in their proposed method [1, Alg. 1] consists of having to *repeatedly* compute larger and larger PSVDs corresponding to singular values above a certain threshold (see SVT-MC Algorithm (3) in Example 2). A similar kernel also appears in an approach for solving large-scale *linear discrete ill-posed* problems, when A is very ill-conditioned [27], which relies on solving a smaller least-squares problem associated with a k-PSVD of A, where k is unknown a priori and is determined *adaptively* so it satisfies the discrepancy principle. The most direct approach for these problems would be to compute some predetermined number t of largest singular triplets by using one of the previously listed routines and check whether the threshold is met. If the threshold is not met, then current limitations of these routines necessitate having to *recompute* $\widetilde{t} > t$ of largest singular triplets. This can be unnecessarily expensive and it completely dismisses the knowledge of the previously computed t singular triplets. Our proposed algorithm completely circumvents this problem.

In this paper we have developed a hybrid algorithm that efficiently and systematically computes (an unknown number of) all singular triplets above a user-specified threshold. So far, to the best of our knowledge, this problem was only addressed either by adapting algorithms that work on a larger symmetric eigenproblem associated with $C = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$ [4] or by adaptation of various randomization techniques (e.g., [8, 28, 29]) that utilize a different stopping criteria (see Remark 1). Our proposed algorithm is based on the *explicit deflation procedure applied to Golub-Kahan-Lanczos Bidiagonalization (GKLB)-based methods* [30] and it greatly extends the functionality of the highly popular MATLAB routine `svds` [26]. What differentiates this algorithm from

the others is that it works directly on matrix A (no need to consider $C = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$) and it maintains the overall high accuracy one expects from `svds`. Furthermore, our results can also be easily adapted to various existing `svds`-like implementations in programming languages such as C (`svds-C` and `irlb.c`), C++ (`CppIrlba`), Python (`irlb.py`), R (`irlba.R`), and Julia (`svdl.jl`).[1]

With respect to GKLB-based methods, the first observation relevant to this paper is that they can be thought of as a one–sided PSVD approximations of A where either $\mathcal{E}_1$ or $\mathcal{E}_2$ in (1) is exactly zero while the other value contains the approximation error [17, 18]. Furthermore, the structure of GKLB-based methods have the added advantage of allowing for the explicit deflation technique to be applied only to either the left or right singular vectors of A, depending on its dimensions, and thus significantly reducing the overall computational cost. Due to our iterative deflation process, a modest non–zero error, that is orthogonal to computed basis vectors, is added to the side in the GKLB method that had theoretically zero error [30].

Although the error growth is theoretically manageable, our experiences while developing our public software on a wide–range of problems with varying singular value distributions showed that not to be the case and thus highlighting that even a seemingly straightforward explicit deflation as in [30] requires special care when implementing. This led us to develop a new, robust hybrid scheme that combines a GKLB-based method [30] together with a variant of the block SVD power method [31], the latter being solely used to restore the one–sided GKLB structure as needed. This helped with maintaining strong orthogonality among the deflated and newly computed singular vectors, computing all multiple singular values, and avoiding the pitfall of recomputing mapped singular values – see Section 2 and Example 1.

To make our algorithm more accessible to a wider audience, we developed three standalone and well-documented computer codes which can be run either independently or as a part of a larger routine. All three codes repeatedly call a GKLB-based method based on [18] to compute a PSVD of A while applying an explicit deflation technique. In other words, they only differ in which `psvd` method is being called and the programming environment. Specifically, our codes `svt_svds.m` (MATLAB), `svt_irlba.m` (MATLAB/Octave), and `svt_irlba.R` (R), call MATLAB native function `svds`, a MATLAB/Octave internal implementation of `irlba` [18], and the R package `irlba.R` [19], respectively. All codes and documentation are available on GitHub https://github.com/jbaglama/svt. It is important to note here that while both codes `svds.m` and `irlba.m` are based on the thick–restarted GKLB method [18], there are subtle differences in the performance of `svt_svds.m` and `svt_irlba.m` (see Section 4). Furthermore, since `svt_irlba.m` provides its own standalone implementation of `irlba` it can also run in the freely available Octave, whereas `svt_svds.m` cannot. This is particularly relevant since Octave's `svds` is *not* a GKLB-based method and thus it cannot be used in our proposed framework. In fact, Octave's `svds` creates the matrix `C =[0 A;A' 0]` and calls Octave's `eigs` function to compute the associated eigenpairs before translating them to the singular triplets of A. This implementation was also used by MATLAB's `svds` at the time

[1] All are public codes that can be found on GitHub: https://github.com.

when [1] first appeared (2010) and was the rationale for the authors to use PROPACK (GKLB-based method) in their SVT-MC Algorithm instead of that version of `svds` citing a speedup of factor of ten [1, Sec. 5.1.1].

We are only aware of one comparable MATLAB code that computes all singular triplets above a given user-inputted singular value threshold, `svt.m` from [4]. Similar to Octave's `svds`, the code `svt.m` uses `C =[0 A;A' 0]` and calls MATLAB's `eigs` function. As we will see in Section 4, this is very inefficient and our methods consistently outperform `svt.m`. With respect to `svt_irlba.R`, to the best of our knowledge this is the only R implementation that computes all singular values above a specific threshold thus making it very valuable to diverse communities that are heavy R users.

An added feature that further enhances the overall functionality of our codes is that instead of providing a thresholding value that singular triplets must exceed users may opt to specify a desired energy level of a PSVD, where $energy = \|A_k\|_F^2/\|A\|_F^2 = \Sigma_{i=1}^k \sigma_i^2/\|A\|_F^2$, with applications including imaging and nonlinear dimensionality reduction, see e.g. [8, 28, 32] and the references therein. We note that under some mild conditions (Proposition 1) the notion of *energy* is also closely related to another popular error measurement based on the Frobenius norm, the normalized root mean squared error, $nrmse = \|A - A_k\|_F/\|A\|_F$, [7, 32].

Proposition 1 *Given A_k with either $\mathcal{E}_1$ or $\mathcal{E}_2$ equal to zero (1), then $\|A - A_k\|_F^2 = \|A\|_F^2 - \|A_k\|_F^2$.*

Proof Without loss of generality, assume $\mathcal{E}_1 = 0$ in (1). Recall the matrix–Pythogoras Lemma [33, Lemma 2.1] which states that if $X, Y \in \mathbb{R}^{m\times n}$ and $Y^T X = 0$, then $\|X+Y\|_F^2 = \|X\|_F^2 + \|Y\|_F^2$. The desired conclusion follows by setting $X = A^T - A_k^T$, $Y^T = A_k$, and using (1) with $\mathcal{E}_1 = 0$ to show that $Y^T X = 0$. □

From Proposition 1, it now follows that if $\mathcal{E}_1 = 0$ and/or $\mathcal{E}_2 = 0$ in (1), then

$$(nrmse)^2 = \|A-A_k\|_F^2/\|A\|_F^2 = \|A\|_F^2/\|A\|_F^2 - \|A_k\|_F^2/\|A\|_F^2 = 1 - energy\,. \quad (2)$$

Remark 1 *It is worth highlighting that for low-rank approximations obtained via randomization, such as in* [2], Proposition 1 *corresponds to* [2, Thm. 1]. *Furthermore, it is worthwhile emphasizing that the user-inputted threshold condition (singular value or energy percentage, or normalized root mean squared error)* is not *used as stopping criteria within* `svds.m or irlba.m` *methods in the proposed algorithm in this paper, but rather as an exit criterion for the wrapper codes* `svt_svds.m`, `svt_irlba.m`, *and* `svt_irlba.R`. *That is, the k–PSVD of A is computed to within a prescribed* `svds.m or irlba.m` *method's tolerance* ('tol' in Table 1) *and then tested against the user-inputted threshold condition. This differs from other routines, e.g.* `farPCA`[2] [28], *that use the energy percentage or normalized root mean squared error as a stopping criteria, and ultimately makes any comparison with respect to timing and accuracy between those methods and ours difficult to interpret* (see comparison of one-sided errors in Example 3). *We note that in applications such as imaging where*

[2] Date: June 28, 2024: GitHib: https://github.com/THU-numbda/farPCA

the use of (2) *as an stopping criteria to compute a PSVD of a matrix is appropriate, randomized SVD methods like* `farPCA` [28] *are competitively fast and ought to be considered.*

The paper is organized as follows. The next section contains our main contribution in the form of a hybrid algorithm (detailed codes are avilable at GitHub https://github.com/jbaglama/svt), while in Section 3 we provide a detailed description of all parameters and include a sample of one-line computer code function calls. Finally, in Section 4 we present several numerical experiments with application to matrix completion and image compression, while Section 5 contains concluding remarks.

2 Algorithms

In this section we only describe simplifications of our codes and refer the reader to publicly available documentation on GitHub for further details. Our Algorithm 1 is a hybrid scheme that repeatedly calls a GKLB-based method and based on certain criteria calls a block SVD power (`blksvdpwr`) method: Algorithm 2. We first discuss the GKLB-based method, the criteria which connects the methods, and then finally the `blksvdpwr` method.

At the center of our Algorithm 1 is the ability to repeatedly call a GKLB-based method (from now on generically referred to as a `psvd` method) to compute a PSVD of a matrix while taking the advantage of explicit deflation techniques [30]. Our codes `svt_svds.m`, `svt_irlba.m`. and `svt_irlba.R` are virtually the same modulo the choice of the `psvd` method and programming environment, i.e., `psvd` $\in$ {`svds.m`, `irlba.m`, `irlba.R`}.

We postpone providing detailed descriptions for all input *parameters* until Section 3 (see Table 1) – for now, we assume that they are all set to their default values other than the possible input of an optional initial PSVD which inherently alters the start of Algorithm 1 (lines 3–7). The variable ℓ holds the current size of the PSVD of A during an iteration, while the parameter structure *opts* (Algorithm 1: line 17) possesses the tolerance, maximum iteration, subspace dimension, and a starting vector for a `psvd` method. Given these parameters a `psvd` method may not converge for all requested k singular triplets. If some number of singular triplets have converged, then Algorithm 1 readjusts k to that number and continues. In the rare event that a `psvd` method returns with *no* converged singular triplets, we increase the maximum number of iterations and the size of the subspace dimension before recalling a `psvd` method once again. Through numerous experiments we have found that only one additional recall of a `psvd` method is required to get at least one singular triplet to converge. However, if after that one additional iteration a `psvd` method still produces no converged singular vectors, then Algorithm 1 exits at line 17 indicating `psvd` method failed (see Table 1).

The deflation schemes from [30] (Algorithm 1: lines 13 and 15) are not explicitly computed, but rather set–up as internal matrix–vector product routines. For `svds.m` or `irlba.m` this is done via an internally written function handle, while for `irlba.R` we provide two options, the `irlba.R` parameter *mult* and a custom matrix multipli-

Algorithm 1 Hybrid Singular Value Threshold.

1: **Input:** A, threshold condition (*sigma* or *energy*); *parameters* (see Section 3).
2: **Output:** U; S; V.
3: **if** nonempty($[U_0, S_0, V_0]$) **then**
4: $V = V_0$; $U = U_0$; $S = S_0$; $\ell = k_0$;
5: **if** $pwrsvd > 0$ **then**
6: $[U, S, V] =$ `blksvdpwr`$(A, V, U, pwrsvd)$;
7: **end if**
8: **else**
9: $A_d = A$; $V = [\,]$; $U = [\,]$; $S = [\,]$; $\ell = 0$;
10: **end if**
11: **while** $\ell \leq \min(m, n)$ **do**
12: **if** $\ell > 0$ and $m \leq n$ **then**
13: $A_d = A - U(U^T A)$
14: **else if** $\ell > 0$ and $m > n$ **then**
15: $A_d^T = A^T - V(AV)^T$
16: **end if**
17: $[U_1, S_1, V_1] =$ `psvd`$(A_d, k,$ opts$)$; % `psvd` ∈ {`svds.m`, `irlba.m`, `irlba.R`}
18: C1: $\ell > 0$ and $\max(|V_1^T V|$ *or* $|U_1^T U|) > \sqrt{\epsilon}/(\ell + k)$;
19: C2: $\min(S_1) < \max(S)\sqrt{\epsilon}$;
20: C3: `psvd` returned > 0 and $< k$ values
21: **if** (C1 or C2 or C3 or $pwrsvd > 0$) **then**
22: $[U, S, V] =$`blksvdpwr`$(A, [V\ V_1], [U\ U_1], \max(1, pwrsvd))$;
23: **else**
24: $U = [U\ U_1]$; $V = [V\ V_1]$; $S =$ `blkdiag`(S, S_1);
25: **end if**
26: $\ell = \ell +$ size(S_1);
27: **if** $\min(S) <$*sigma* or $\Sigma_{i=1}^{k} \sigma_i^2 / \|A\|_F^2 \geq energy$ **then**
28: truncate and return $[U, S, V]$;
29: **else**
30: $k = k + incre$; $incre = 2 \cdot incre$;
31: **end if**
32: **end while**

cation function (*cmmf*) for the user to choose (see Table 1). We refer to the `irlba.R` documentation for details on *mult* and *cmmf* [19].

We now turn our attention to the *hybrid* aspect of our proposed method, namely the use of a block SVD power method, `blksvdpwr` from Algorithm 2, within Algorithm 1. Use of `blksvdpwr` method is triggered either via the user-inputted parameter *pwrsvd* which forces its use on every iteration of Algorithm 1 or when a certain criteria is met (see three cases in Algorithm 1, lines 18–20). Case C1 computes the orthogonality "level" of the large basis vectors that are implicitly held orthogonal during the deflation process. From our experiments, we have noticed that the large basis vectors rarely lose orthogonality, though in some isolated cases this can happen, e.g., when A is very ill–conditioned, a "large" tolerance is used, or if A has a significant number of multiple interior singular values (see Example 1). The choice of required orthogonality "level" before `blksvdpwr` gets called follows [17, Thm. 5]. A more challenging scenario is the case C2 largely due to the fact that the deflation scheme (Algorithm 1: lines 13 and 15) maps the large singular values to "zero" which under certain numerical conditions can reappear, i.e., be recomputed by a `psvd` method (see

Table 1 Parameters and output for `svt_svds.m`, `svt_irlba.m`, and `svt_irlba.R`

Parameters for `svt_svds.m`, `svt_irlba.m`, and `svt_irlba.R`	
sigma	threshold – if missing returns top k singular triplets
energy	energy percentage (≤ 1) (2) (cannot be combined with *sigma*)
tol	tolerance used for convergence in the `psvd` method (default: $\sqrt{eps}$)
k	initial k used in the `psvd` method (default: 6)
incre	initial increment added to k (default: 5)
kmax	maximum value k can reach (default: $\min\{0.1 \cdot \min(m, n), 100\}$)
p0	initial vector in the `psvd` method (default: *p0* = *randn*)
psvdmax	max. dim. of output (default: $\max(\min(100 + \text{size}(S0), \min(n, m)), k)$)
pwrsvd	performs $pwrsvd$ iterations of Algorithm 2 (default: 0)
display	if set to 1, then display some diagnostic information (default: 0)
Parameters for `svt_svds.m` and `svt_irlba.m`	
m	number of rows of A – required if A is a function handle
n	number of columns of A – required if A is a function handle
U0	left singular vectors of a previous PSVD of A (default: [ ])
V0	right singular vectors of a previous PSVD of A (default: [ ])
S0	diagonal matrix of singular values of a previous PSVD of A (default: [ ])
Parameters for `svt_irlba.R`	
psvd0	a list of a previous PSVD of A (default: NULL)
cmmf	if TRUE uses an internal custom matrix multiplication function to compute the matrix–vector product deflation technique (Algorithm 1, lines 15, 17). If FALSE, then uses an internal `mult` parameter from `irlba.R` to perform the deflation technique (default: FALSE)
Output for `svt_svds.m` and `svt_irlba.m`	
U, S, V	left singular vectors, diagonal matrix of singular values, right singular vectors
FLAG	0 – successful output – either threshold or energy percentage satisfied
	1 – `psvd` fails to compute any singular triplets – output last values of U, S, V
	2 – *psvdmax* is reached – output last values of U, S, V
	3 – no singular values above threshold *sigma* – output $U = [\,], V = [\,], S = [\,]$
Output for `svt_irlba.R`	
Returns a list `psvdR` with U as `psvdR$u`, V as `psvdR$v`, S as a vector `psvdR$d`, and *FLAG* as `psvdR$flag` with same values as above.	

Example 1). Case C1 generally catches the recomputing of already mapped singular values, though numerous experiments showed that this may still occur when too many singular triplets are requested with respect to the rank of A or when A has a very large number of interior multiple singular values. Finally, case C3 is more of a precautionary case which typically indicates that the GKLB process struggled computing interior values and therefore, securing a more stringent orthogonality level among basis vectors helps with the overall convergence of a `psvd` method on the next iteration.

With respect to `blksvdpwr` itself, we note that our Algorithm 2 is based on the routine described in [31] with slight differences in its implementation when it comes to lines 9 and 16 of Algorithm 2. We compute the full SVD of the upper triangular matrix R, whereas the authors in [31] use R as the approximate diagonal matrix S. For a large matrix R, this computation can appear to be expensive, however the computation of the full SVD of R is not anymore expensive in the overall scheme of Algorithm 1 where a `psvd` method, which depends on a GKLB-based method, repetitively computes the full SVD of a projected matrix of a similar size [18]. It is worth highlighting here that while it is theoretically possible to use Algorithm 2 to compute a PSVD of A that is not our objective here and no comparisons are made with respect to this aspect. In our scenario, the sole purpose of Algorithm 2 within Algorithm 1 is simply to ensure the overall orthogonality of the basis vectors, check the reliability of a PSVD of A, and restore the one–sided GKLB structure. Moreover, an arbitrary SVD algorithm can not be used in place of Algorithm 2 as accuracy of Algorithm 1 heavily relies on having the one-sided GKLB structure. Therefore, the output from Algorithm 2 is of the form (1) with $\mathcal{E}_1 = 0$ and $\mathcal{E}_2 \neq 0$ when $m \leq n$ or with $\mathcal{E}_2 = 0$ and $\mathcal{E}_1 \neq 0$ in case when $m > n$.

Finally, the determination of next k value and $incre$ (Algorithm 1: line 30) follows the discussion in [4], where $incre$ is doubled on each iteration – the user can only preset the initial values of k and $incre$ on input (see Table 1). It is worth noting here that a priori knowledge of an initial k can enhance overall convergence. Some promising ideas on how to estimate an initial k for a `psvd` method are based on using harmonic Ritz values [34] or eigenvalue estimates of $A^T A$ [35]. These are beyond the scope of this paper and at this time are an ongoing area of research. The algorithm terminates (Algorithm 1: line 27) if it successfully computed all singular triplets above the user-inputted threshold *sigma* or if $\Sigma_{i=1}^{k} \sigma_i^2 / \|A\|_F^2 \geq energy$ for the user-inputted $0 < energy \leq 1$.

Remark 2 *When $m > n$ and for the* R *code* `svt_irlba.R` *one iteration of* Algorithm 2 *must run each iteration of* Algorithm 1 *to safely still use the deflation theory in* [30] *applied to the smaller dimension side. This is because of how the* R *package* `irlba.R` [19] *handles matrix–vector products with the transpose of A.*

3 Input, parameters, and output

All our codes share a common set of key input parameters, though some parameters are language specific, e.g., MATLAB/Octave vs R (see Table 1). Of particular note are the parameters *kmax* and *psvdmax* whose purpose is to prevent excess computational

Algorithm 2 Block SVD Power Method (`blksvdpwr`).

1: **Input:** $A \in \mathbb{R}^{m \times n}$; V; U; $iter$
2: **Output:** U; S; V.
3: **if** $m \leq n$ **then**
4: $[U, R] = \texttt{qr}(U)$;
5: **for** $j = 1 : iter$ **do**
6: $[V, R] = \texttt{qr}(A^T U)$;
7: $[U, R] = \texttt{qr}(AV)$;
8: **end for**
9: $[u_r, S, v_r] = \texttt{svd}(R)$;
10: **else**
11: $[V, R] = \texttt{qr}(V)$;
12: **for** $j = 1 : iter$ **do**
13: $[U, R] = \texttt{qr}(AV)$;
14: $[V, R] = \texttt{qr}(A^T U)$;
15: **end for**
16: $[v_r, S, u_r] = \texttt{svd}(R)$;
17: **end if**
18: return $V = Vv_r$; $U = Uu_r$; S;
(`qr` is the economy-size QR decomposition)
(`svd` is the full SVD method)

time by limiting the number of requested singular values to a `psvd` method (*kmax*) and the size of the output $[U, S, V]$ (*psvdmax*). The user can maintain very strong orthogonality among basis vectors and reset the one–sided GKLB structure by setting $pwrsvd > 0$, otherwise the error growth is monitored (Algorithm 1: lines 18–20) and reset as needed. The output and variable *FLAG* that alerts the user to method's convergence are also given in Table 1.

To aid the reader, and also quickly demonstrate simplicity and versatility of our codes, we now provide several sample command line calls for each of them. However, in the publicly available repository, we exemplify how to call each routine using different combinations of available parameters and include demo files that reproduce all numerical results from Section 4 and some additional ones. Since MATLAB `svt_svds.m` and MATLAB/Octave `svt_irlba.m` codes have *identical* command line calls, we only include one for `svt_svds.m`. Matrices U, S, V, and variable $FLAG$ (Table 1) are returned directly from our MATLAB/Octave routines, whereas `svt_irlba.R` returns a *list* `psvdR` which can then be used to access U as `psvdR$u`, V as `psvdR$v`, diagonal matrix S as a vector `psvdR$d`, and *FLAG* as `psvdR$flag`. For example, MATLAB and R commands, denoted by `M»` and `R>`, respectively, used to compute all singular triplets with singular values exceeding 1.1 are given by:

```
M» [U,S,V,FLAG] = svt_svds(A, 'sigma', 1.1);
R>  psvdR <- svt_irlba(A, sigma = 1.1)
```

Analogously, command lines for computing all singular triplets with singular values exceeding 1.1 given initial PSVD are as follows:

```
M» [U, S, V, FLAG] = svt_svds(A, 'sigma', 1.1, 'U0', U, 'V0'
, V, 'S0', S);
R>  psvdR <- svt_irlba(A, sigma=1.1,  psvd0 = psvdR)
```

Finally, for sample calls where energy percentage is used for thresholding purposes see Example 3.

4 Numerical examples

All examples were carried out on a MacBook Pro (M1 Max) with 32GB of memory and macOS Sonoma 14.4.1. Computations were performed using MATLAB R2024a (Examples 1-2) or Rstudio-2023.06.1+524 (v4.3.2) (Example 3), both with machine

epsilon $\epsilon = 2.2 \cdot 10^{-16}$. The recorded total cpu times (in seconds), denoted by $\mathcal{T}_{tot}^{cpu}$, were done either with MATLAB `tic-toc` or R `proc.time` commands. For Example 1 multiple runs were used before the final results were recorded, while for Examples 2 and 3 only a single run was performed. The authors noticed that when the main comparison of methods was timing, very little differences were observed between the runs and the reported results are typical.

Example 1 Here, we compare performance of our two codes `svt_svds.m` and `svt_irlba.m` with the only other currently available comparable MATLAB alternative, `svt.m`[3] [4]. For comparison we used 12 sparse matrices listed in Table 2 from the SuiteSparse Matrix Collection [36] which vary in size, structure, and range/magnitude of singular values. The first six matrices, along with the corresponding choice of threshold values *sigma*, are exactly the same as used in [4] thus allowing for a fair comparison. The low-rank perturbation LR^T of the first five matrices, with L and R being random matrices with 10 columns, is consistent with examples in [4] – same RandStream used for `svt.m`[3].

Each matrix problem was run 10 times. In all three codes, the common parameters *sigma*, *tol*, *k*, and *incre*, were uniformly set, while all other parameters were set to default with the exception of $tol = 10^{-8}$, $kmax = 100$, and $psvdmax = 800$ (see Table 1). In Table 2 we also recorded the average cpu times for $\mathcal{T}_{tot}^{cpu}$ and maximum values for errors $\mathcal{E}_{tot}^{err} = \sqrt{\|\mathcal{E}_1\|^2 + \|\mathcal{E}_2\|^2}$ cf. (1) and $\mathcal{UV}^{err} = \sqrt{\|V^TV - I\|^2 + \|U^TU - I\|^2}$.

From Table 2 it is evident that with respect to cpu timings `svt_svds.m` clearly outperforms the other two codes for all test matrices, especially when compared to `svt.m` which is significantly slower. On the other hand, only a minor variation in timings between our two codes is not surprising, given their main difference is the choice of a `psvd` method in Algorithm 1. These two `psvd` methods have the same underlying structure, and so the slight difference in timings in Table 2 is accounted by differences in their implementation, e.g., reorthogonalization of basis vectors, heuristics used for convergence, detection of multiple singular values, and selection of vectors to use for restarting [17, 18].

Turning our attention to errors $\mathcal{E}_{tot}^{err}$ and $\mathcal{UV}^{err}$, Table 2 shows that both of our codes maintained strong orthogonality among both basis vectors, U and V, which was not always the case for `svt.m`. Furthermore, `svt.m` failed to compute all of the singular values above the prescribed threshold for three of the test matrices. In the case of the matrices *maragal_2* and *well1850,* the number of needed singular values was very close to their ranks thus resulting in `svt.m` recomputing mapped values – the hybrid structure and use of `blksvdpwr` routine prevented our codes succumbing to the same fate, emphasizing the need for such a check. Finally, for the matrix *illc1033*, which has large number of multiple singular values clustered around 1.0, `svt.m` struggled and was only able to compute 128 out of the 197 singular values above $sigma = 0.9$. Setting $tol = 10^{-16}$ for `svt.m` also did not result in success.

[3] Codes available at: https://github.com/Hua-Zhou/svt, retrieved on March 30, 2024.

Table 2 Example 1: Errors $\mathcal{E}_{tot}^{err}$ and $\mathcal{UV}^{err}$ and the total cpu timing $\mathcal{T}_{tot}^{cpu}$ in seconds, for each of the codes that successfully computed all singular values above *sigma*

matrix	*sigma*	#	`svt_svds.m`			`svt_irlba.m`			`svt.m`		
(size)		SVs	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{E}_{tot}^{err}$	$\mathcal{UV}^{err}$	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{E}_{tot}^{err}$	$\mathcal{UV}^{err}$	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{E}_{tot}^{err}$	$\mathcal{UV}^{err}$
1. bibd_20_10+LR^T (190×184756)	466	20	2.1s	10^{-11}	10^{-14}	2.3s	10^{-11}	10^{-14}	6.6s	10^{-5}	10^{-8}
2. bibd_22_8+LR^T (231×319770)	435.79	20	4.7s	10^{-10}	10^{-14}	4.8s	10^{-10}	10^{-14}	31.2s	10^{-5}	10^{-9}
3. bfwb398+LR^T (398×398)	$2.2\cdot10^{-5}$	50	0.06s	10^{-12}	10^{-15}	0.08s	10^{-12}	10^{-15}	0.54s	10^{-12}	10^{-10}
4. mhd4800b+LR^T (4800×4800)	0.122	50	2.5s	10^{-8}	10^{-11}	2.2s	10^{-8}	10^{-11}	15.5s	10^{-8}	10^{-9}
5. cryg10000+LR^T (10000×10000)	21757	50	16.2s	10^{-4}	10^{-14}	10.7s	10^{-4}	10^{-14}	21.8s	10^{-4}	10^{-9}
6. stormG2_1000(528185×1377306)	632.46	50	25.9s	10^{-6}	10^{-14}	72.1s	10^{-6}	10^{-14}	183.6s	10^{-11}	10^{-14}
7. maragal_2 (555×350)	10^{-10}	171	0.34s	10^{-14}	10^{-15}	0.45s	10^{-14}	10^{-15}	$\star$	$\star$	$\star$
8. illc1033(1033×320)	0.9	197	0.76s	10^{-8}	10^{-13}	1.1s	10^{-9}	10^{-10}	$\star$	$\star$	$\star$
9. well1850(1850×712)	0	712	5.2s	10^{-8}	10^{-9}	6.3s	10^{-8}	10^{-10}	$\star$	$\star$	$\star$
10. JP(87616×67320)	1500	118	31.1s	10^{-5}	10^{-15}	32.4s	10^{-7}	10^{-15}	55.5s	10^{-5}	10^{-9}
11. Rel8(345688×12347)	12.5	47	16.5s	10^{-7}	10^{-14}	29.0s	10^{-7}	10^{-14}	22.2s	10^{-7}	10^{-9}
12. Rucci1(1977885×109900)	6.5	33	55.1s	10^{-7}	10^{-14}	114.6s	10^{-8}	10^{-14}	75.2s	10^{-8}	10^{-9}

Value "$\star$" indicates the code didn't successfully compute all the singular values

Example 2 In this example we continue to compare our two MATLAB thresholding codes with `svt.m` [4], but this time through the lens of an application to *matrix completion* briefly discussed in Section 1. We emphasize here that our objective is *not* to devise a new competitive algorithm for matrix completion, but instead to illustrate the ease at which our codes can be incorporated into larger numerical schemes and have a meaningful impact. More specifically, we replace the main computational kernel, the PSVD section given by lines 5–8 in (3), in the original SVT-MC implementation [1, Alg. 1] with `svt_svds.m`, `svt_irlba.m`, and `svt.m`. We do not compare here against alternatives that either modify other parts of SVT-MC Algorithm or ones that simply replace line 6 in (3). For example, in [2] it appears that the authors utilize a randomized SVD method solely for line 6 in (3) (publicly available code `fastsvt`[4]) and do illustrate an impressive performance increase over when only `svds` is used. But, as far as we can tell from the publicly available codes[4], `fastsvt` constantly recomputes the PSVD until the desired threshold is met, i.e., dismisses the knowledge of the previously computed PSVDs, thus making the comparison with our routines which are based on a completely different premise beyond the scope of this paper.

While several versions of the original SVT-MC Algorithm are publicly available (MATLAB, C, and FORTRAN),[5] we only consider its MATLAB implementation. As stated in Section 1, at the time of publication [1], the authors used PROPACK [17] as a method of choice for the PSVD computation (line 6 in (3)) instead of the built–in MATLAB routine `svds`. Since then, the `svds` has changed and is now based on PROPACK with thick–restarting [26], and so, for the purposes of comparison with our implementations of `svt_svds`, `svt_irlba`, and `svt`, we use MATLAB's `svds` for line 6 in (3).

$$
\begin{array}{l}
\vdots \\
\text{5. } \textbf{repeat} \\
\text{6. } \quad \text{Compute } [U^{k-1}, S^{k-1}, V^{k-1}]_{s_k} \\
\text{7. } \quad \text{Set } s_k = s_k + \ell \\
\text{8. } \textbf{until } \sigma_{sk-\ell}^{k-1} \leq \tau \\
\vdots
\end{array}
\left.\begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array}\right\}
\begin{array}{l}
\text{PSVD section} \\
\text{of } \textbf{SVT} - \textbf{MC} \\
\text{Algorithm} \\
\text{[1, Alg.1]}
\end{array}
\left.\begin{array}{l} \\ \\ \\ \\ \\ \\ \end{array}\right\}
\begin{array}{l}
\text{replace with :} \\
\texttt{svt\_svds} \\
\texttt{svt\_irlba} \text{ or} \\
\texttt{svt}
\end{array}
\qquad (3)
$$

In order to analyze, within a larger matrix completion framework, the performance of our proposed alternatives for the PSVD section in (3), we closely follow the recommendations from [1] when it comes to selecting test matrices and all of the parameters. We do so since an exhaustive presentation with all of the choices for different parameters is simply infeasible, e.g., τ (line 8 in (3)) was chosen heuristically in [1], though different values for it could lead to more/less iterations of the SVT-MC Algorithm or even no convergence, see [1, Table 5.2]. We chose our test matrix M to have the same structure (distributed Gaussian entries) as in [1] and made it rectangular instead of

[4] Date: June 28, 2024. GitHub: https://github.com/XuFengthucs/fSVT

[5] SVT–MC codes: https://www.convexoptimization.com/wikimization/index.php/Matrix_Completion.m
SVT–MC codes: https://github.com/stephenbeckr/SVT/blob/master/SVT.m

square, i.e., $M = M_L M_R$, where $M_L \in \mathbb{R}^{2000\times r}$, $M_R \in \mathbb{R}^{r\times 20000}$, and r is a chosen rank (ranging from 1% to 15%) with an oversampling ratio of 4 (see [1, Table 5.1]).

All four runs of SVT-MC Algorithm had successfully recovered the rank – we used 10^{-3} for an overall convergence tolerance and for all PSVD methods tolerance was set to 10^{-8} and initial $k = s_k$. For each of the implementations, in Table 3 we recorded two cpu timings – $\mathcal{T}_{tot}^{cpu}$ for the overall convergence and $\mathcal{T}_{psvd}^{cpu}$ for just the PSVD computations (lines 5–8 in (3)). Easy analysis shows that in virtually all of the cases roughly 90%-95% of overall cpu timing is due to its respective PSVD section, thus clearly indicating the importance of developing flexible portable routines that can be easily integrated as computational kernels in other algorithms, such as the ones presented here. For all choices of parameters *rank* and *sampling* in Table 3 our code `svt_irlba.m` significantly outperformed others, especially as rank increased.

Example 3 In our final example, we investigate performance of the R-implementation of our Algorithm 1, `svt_irlba.R`, and demonstrate its versatility of using energy percentage (2) as a thresholding condition. For that purpose, we consider the classical problem of image compression and analyze `svt_irlba.R`'s performance against a comparable and highly popular R package `rsvd` [7].

For the sake of fair comparison, we consider the 1600×1200 grayscale image `tiger`[6] that was used in [7] and readily available from the `rsvd` package:

```
R> data("tiger", package = "rsvd")
```

When initializing our `svt_irlba.R` code all parameters were kept at default with the exception of $tol = 10^{-5}$ and $psvdmax = 1200$. Based on the discussion in [7], we consider $nrmse = 0.12081$ as a thresholding consideration, which together with (2) in turn corresponds to energy percentage 98.54% and is called via the following R command:

```
R>psvdR <- svt_irlba(tiger, tol = 1e-5, energy = 0.9854, psv
dmax = 1200)
```

The compressed image can be reconstructed with the output from `svt_irlba.R` and displayed with the R command `image`, see [7] for details. In this case `svt_irlba.R` successfully returned an output with $k = 100$ singular triplets (same k value as in [7]) and the desired $nrmse = 0.12081$ while requiring $\mathcal{T}_{tot}^{cpu} = 6.2$s with errors $\mathcal{E}_{tot}^{err} = 10^{-13}$ and $\mathcal{UV}^{err} = 10^{-14}$. We compared our results against `rsvd` using its default parameter values though with one major distinction – `rsvd` requires a priori knowledge of the number of desired singular triplets, i.e., it required setting $k = 100$. In this case, `rsvd` clocked total cpu timing of $\mathcal{T}_{tot}^{cpu} = 1.2$s being several times faster than our method `svt_irlba.R`. While the `rsvd` outputted a PSVD with $\mathcal{UV}^{err} = 10^{-14}$, it only achieved $nrmse = 0.12238$. This is the same output displayed within three significant digits as in [7, Table 1] for `rsvd` $q = 2$ (default value). It is important to highlight that $\mathcal{E}_{tot}^{err}$ is *not* an appropriate error measure for `rsvd` since $\|\mathcal{E}_1\| = 10^{-13}$ and $\|\mathcal{E}_2\| = 10^{0}$. Furthermore, `rsvd` does not have an input parameter for tolerance or error measurement – this is checked outside the function. This observation also tends to plague other numerical schemes that use energy as a stopping criteria, whereas in our case the energy level *is not* used to regulate the

[6] Image is also freely available: https://en.wikipedia.org/wiki/File:Siberischer_tiger_de_edit02.jpg.

Table 3 Example 2: Comparison of the PSVD codes used in SVT–MC Algorithm for matrix $M = M_L M_R$, where $M_L \in \mathbb{R}^{2000\times r}$ and $M_R \in \mathbb{R}^{r\times 20000}$

rank (%)	`svds.m`		`svt_svds.m`		`svt_irlba.m`		`svt.m`	
sample (%)	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{T}_{psvd}^{cpu}$	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{T}_{psvd}^{cpu}$	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{T}_{psvd}^{cpu}$	$\mathcal{T}_{tot}^{cpu}$	$\mathcal{T}_{psvd}^{cpu}$
20 (1%) 4%	1453.1s	1305.7s	1509.3s	1360.3s	902.1s	753.1s	1740.1s	1593.6s
50 (2.5%) 11%	147.6s	133.03s	143.3s	128.2s	132.5s	118.0s	222.6s	208.1s
100 (5%) 22%	371.1s	345.3s	333.0s	307.1s	311.1s	285.5s	647.5s	621.8s
200 (10%) 44%	1341.4s	1296.9s	1020.4s	976.4s	981.3s	937.4s	1842.5s	1799.4s
300 (15%) 65%	2933.5s	2877.2s	1930.5s	1874.5s	1867.9s	1812.1s	3276.5s	3220.7s

$\mathcal{T}_{tot}^{cpu}$ is overall timing and $\mathcal{T}_{psvd}^{cpu}$ timing just for the PSVD method (lines 5–8 in (3))

underlying `irlba.R`, `irlba.m`, and `svds.m`, but rather as an exit thresholding condition for the provided wrapper codes (see Remark 1).

As a possible strategy to achieve a desired $nrmse$ value, as suggested in [7, Table 1], one may try to increase the number of additional power iterations, q. On our first try, we set $q = 3$ and obtained an error of $nrmse = 0.12145$ which when rounded to three significant digits matches the error of $nrmse = 0.121$ reported in [7, Table 1]. However, if we wish to achieve the same accuracy in this error measurement ($nrmse$) that meant setting $q = 15$ which resulted in $\mathcal{T}_{tot}^{cpu} = 6.3$s and $\mathcal{UV}^{err} = 10^{-14}$ with $nrmse = 0.12082$. In other words, with respect to cpu timings `svt_irlba.R` performed at least as good as `rsvd`, but with better errors and no required a priori knowledge on the number of needed singular values.

Now, one might take the output from the first calculation and compute the energy percentage 99%:

```
R > psvdR <- svt_irlba(tiger, tol = 1e-5, energy = 0.99, psvdmax = 1200, psvd0 = psvdR)
```

For `svt_irlba.R` this required additional 6.3s, computing 155 singular triplets with $nrmse = 0.09991$, $\mathcal{E}_{tot}^{err} = 10^{-7}$ and $\mathcal{UV}^{err} = 10^{-14}$. Here the default value of $k = 6$ is used for `svt_irlba.R` which may not always be the most efficient and can be improved upon if there is a sensible guess for the number of additional initial number of singular triplets needed. Finally, we conclude that was not even an option with the `rsvd` package.

5 Concluding remarks

We presented several publicly available well-documented software implementations (MATLAB and R) of a new hybrid algorithm demonstrating the ability to successfully and efficiently compute all of the singular values of A above either a given threshold or a given energy percentage. Our codes consistently outperform comparable implementations tackling the same problem and in many cases the difference is quite substantial. Finally, the robustness and flexibility of the codes, as demonstrated with applications to matrix completion and image compression in Examples 2-3, highlight their ease of use either as a standalone program or as a part of a more complex numerical routine.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s11075-024-01906-9.

Author Contributions All authors contributed equally to this work.

Funding The work of James Baglama was partially supported by the Faculty Career Enhancement Grant 2023-2024 through the University of Rhode Island.

Data Availability No datasets were generated or analysed during the current study.

Code availability The software is available as `svt_svds.m`, `svt_irlba.m`, and `svt_irlba.R` from the author's GitHub account (https://github.com/jbaglama/svt).

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Competing interests The authors declare no competing interests.

References

1. Cai, J.-F., Candès, E.J., Shen, Z.: A singular value thresholding algorithm for matrix completion. SIAM J. Optim. **20**(4), 1956–1982 (2010)
2. Feng, X., Yu, W., Li, Y.: Faster matrix completion using randomized svd. In: 2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 608–615 (2018). IEEE
3. Kumar, R., Patbhaje, U., Kumar, A.: An efficient technique for image compression and quality retrieval using matrix completion. J. King Saud Univ. Comput. Inf. Sci. **34**(4), 1231–1239 (2022)
4. Li, C., Zhou, H.: svt: Singular value thresholding in MATLAB. J. Stat. Softw. **81**(2) (2017)
5. Li, Y., Yu, W.: A fast implementation of singular value thresholding algorithm using recycling rank revealing randomized singular value decomposition. Preprint at arXiv:1704.05528 (2017)
6. Asnaoui, K.E.: Image compression based on block svd power method. J. Intell. Syst. **29**(1), 1345–1359 (2019)
7. Erichson, N.B., Voronin, S., Brunton, S.L., Kutz, J.N.: Randomized matrix decompositions using R. J. Stat. Softw. 89(11) (2019)
8. Ji, H., Yu, W., Li, Y.: A rank revealing randomized singular value decomposition (R3SVD) algorithm for low-rank matrix approximations. Preprint at arXiv:1605.08134 (2016)
9. Narwaria, M., Lin, W.: Svd-based quality metric for image and video using machine learning. IEEE Trans. Syst. Man Cybern. Part B (Cybernetics) **42**(2), 347–364 (2011)
10. Cherapanamjeri, Y., Jain, P., Netrapalli, P.: Thresholding based outlier robust PCA. In: Conference on Learning Theory, pp. 593–628 (2017)
11. Jolliffe, I.T.: Principal Component Analysis, 2nd edn. Springer Series in Statistics. Springer, New York (2010)
12. Xu, H., Caramanis, C., Sanghavi, S.: Robust PCA via outlier pursuit. Advances in neural information processing systems, p. 23 (2010)
13. Eldén, L.: Matrix Methods in Data Mining and Pattern Recognition. SIAM, Philadelphia (2019)
14. Alter, O., Brown, P.O., Botstein, D.: Singular value decomposition for genomewide expression data processing and modeling. Proc. Nat. Acad. Sci. **97**(18), 10101–10106 (2000)
15. Tsuyuzaki, K., Sato, H., Sato, K., Nikaido, I.: Benchmarking principal component analysis for large-scale single-cell RNA-sequencing. Genome Biol. **21**, 1–17 (2020)
16. Golub, G., Kahan, W.: Calculating the singular values and pseudo-inverse of a matrix. J. Soc. Ind. Appl. Math. Ser. B: Numer. Anal. **2**(2), 205–224 (1965)
17. Larsen, R.M.: Lanczos bidiagonalization with partial reorthogonalization. DAIMI Rep. Ser. **27**, 537 (1998)
18. Baglama, J., Reichel, L.: Augmented implicitly restarted Lanczos bidiagonalization methods. SIAM J. Sci. Comput. **27**(1), 19–42 (2005)
19. Lewis, B.W., Baglama, J., Reichel, L.: The irlba Package. https://cran.r-project.org/web/packages/irlba (2021)
20. Wu, L., Romero, E., Stathopoulos, A.: Primme svds: A high-performance pre-conditioned svd solver for accurate large-scale computations. SIAM J. Sci. Comput. **39**(5), 248–271 (2017)
21. Liang, Q., Ye, Q.: Computing singular values of large matrices with an inverse-free preconditioned Krylov subspace method. Electron. Trans. Numer. Anal. **42**, 197 (2014)
22. Feng, X., Yu, W., Xie, Y.: svds-c: A multi-thread C code for computing truncated singular value decomposition. SoftwareX **27**, 101781 (2024)
23. Qiu, Y., Mei, J., Guennebaud, G., Niesen, J.: Rspectra: Solvers for large scale eigenvalue and svd problems. R version 0.16-0 (2019)
24. Feng, X., Yu, W., Xie, Y., Tang, J.: Algorithm xxx: Faster randomized SVD with dynamic shifts. ACM Trans. Math. Softw. (TOMS) (2024). https://doi.org/10.1145/3660629. Just Accepted

25. Eaton, J.W., Bateman, D., Hauberg, S., Wehbring, R.: GNU Octave Version 8.4.0 Manual: A High-level Interactive Language for Numerical Computations. (2023). https://www.gnu.org/software/octave/doc/v8.4.0/
26. The Mathworks, I.: MATLAB (R2024a) SVDS. Natick, Massachusetts
27. Onunwor, E., Reichel, L.: On the computation of a truncated SVD of a large linear discrete ill-posed problem. Numer. Algorithms **75**(2), 359–380 (2017)
28. Feng, X., Yu, W.: A fast adaptive randomized pca algorithm. In: Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, pp. 3695–3704 (2023)
29. Yu, W., Gu, Y., Li, Y.: Efficient randomized algorithms for the fixed-precision low-rank matrix approximation. SIAM J. Matrix Anal. Appl. **39**(3), 1339–1359 (2018)
30. Baglama, J., Perović, V.: Explicit deflation in Golub-Kahan-Lanczos bidiagonalization methods. Electron. Trans. Numer. Anal. **58**, 164–176 (2023)
31. Bentbib, A., Kanber, A.: Block power method for SVD decomposition. Analele ştiinţifice ale Universităţii" Ovidius" Constanţa. Ser. Matematică **23**(2), 45–58 (2015)
32. Sadek, R.A.: SVD based image processing applications: State of the art, contributions and research challenges. Int. J. Adv. Comput. Sci. Appl. **3**(7) (2012)
33. Boutsidis, C., Drineas, P., Magdon-Ismail, M.: Near-optimal column-based matrix reconstruction. SIAM J. Sci. Comput. **43**(2), 687–717 (2014)
34. Baglama, J., Reichel, L.: An implicitly restarted block Lanczos bidiagonalization method using Leja shifts. BIT **53**, 285–310 (2013)
35. Di Napoli, E., Polizzi, E., Saad, Y.: Efficient estimation of eigenvalue counts in an interval. Numer. Linear Algebra Appl. **23**(4), 674–692 (2016)
36. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. (TOMS) **38**(1), 1–25 (2011)